X Window, connu également sous l'appellation **X11**, et produit au MIT. Ce système prend en charge les manipulations de base des fenêtres et permet aux utilisateurs de créer, détruire, déplacer ou retailler des fenêtres à l'aide de la souris. Le plus souvent, une IHM complète, comme **Gnome** ou **KDE**, est présente au-dessus de X11 pour donner à UNIX un aspect proche de celui du Macintosh ou de Windows, si l'utilisateur le désire.

Un développement intéressant a débuté vers le milieu des années 1980 : la croissance des réseaux d'ordinateurs personnels fonctionnant sous des **systèmes d'exploitation en réseau** ou des **systèmes d'exploitation distribués.** Dans un système d'exploitation en réseau, les utilisateurs connaissent l'existence individuelle des multiples machines mises à contribution, et peuvent se connecter sur des machines distantes et transférer des fichiers d'une machine à une autre du réseau. Chaque machine fonctionne sous son propre système d'exploitation et gère ses propres utilisateurs.

Les systèmes d'exploitation en réseau ne diffèrent pas en profondeur des systèmes d'exploitation classiques, pour un processeur unique. Ils requièrent la présence d'une interface réseau et de logiciels de bas niveau pour la piloter, ainsi que de programmes permettant la connexion et/ou l'accès aux fichiers à distance, mais cela ne remet pas en cause en profondeur la structure du système.

En revanche, un système d'exploitation distribué apparaît à l'utilisateur identique à un système classique à un processeur. L'utilisateur ne sait pas sur quelle machine s'exécute son programme, ni où se trouvent ses fichiers. Ces détails sont cachés (et gérés efficacement) par le système d'exploitation.

Les véritables systèmes distribués ne se réduisent pas à l'ajout de code à un système classique, car les différences avec les systèmes centralisés sont profondes. Par exemple, un système distribué permet à une application de s'exécuter sur plusieurs processeurs en parallèle, ce qui nécessite des algorithmes d'ordonnancement plus complexes, ne serait-ce que pour optimiser le degré de parallélisme.

Les délais de communication sur le réseau impliquent que ces algorithmes doivent souvent fonctionner avec des informations obsolètes, erronées ou même fausses. Ce contexte est radicalement différent de celui des systèmes monoprocesseurs, où le contrôle de l'état du système est fiable en permanence.

1.3 La structure matérielle d'un ordinateur

Le système d'exploitation est étroitement lié au matériel de l'ordinateur qu'il fait fonctionner. Il en étend le jeu d'instructions et gère ses ressources. Pour cela, il doit connaître en détail ce matériel, ou tout au moins l'aspect qu'il présente au programmeur. Pour cette raison, il est bon de passer en revue les principaux éléments matériels des ordinateurs personnels. Nous pourrons ensuite commencer plus efficacement l'étude des systèmes d'exploitation : ce qu'ils font et comment ils fonctionnent.

Conceptuellement, un ordinateur personnel simple peut être symboliquement schématisé par la description de la figure 1.6. Le processeur ou l'UC, la mémoire et les périphériques d'E/S sont connectés par un bus système et communiquent entre eux *via* ce bus. Les machines modernes ont une structure plus sophistiquée, composée de plusieurs bus, que nous détaillerons plus loin. Pour l'instant, ce modèle simple est suffisant. Dans les sections qui suivent, nous passons ces différents composants en revue en les montrant sous l'angle d'approche du concepteur de système d'exploitation. Il est utile de dire qu'il ne s'agira que d'un résumé sommaire. De nombreux ouvrages ont été écrits sur le sujet, dont *Architecture de l'ordinateur*, publié par Pearson Education France en 2006.

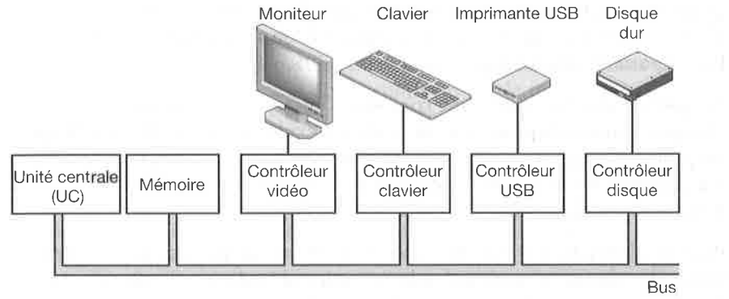

Figure 1.6 • Quelques-uns des composants d'un ordinateur personnel simple.

1.3.1 Le processeur

Le « cerveau » de l'ordinateur est le processeur ou UC. Il extrait des instructions de la mémoire et les exécute. Le cycle de base de tout processeur est d'extraire la première instruction de la mémoire, la décoder pour connaître son type et ses opérandes, l'exécuter, puis recommencer pour les instructions qui suivent. C'est ainsi qu'un programme s'exécute.

Chaque processeur possède un ensemble spécifique d'instructions exécutables. C'est pourquoi un Pentium ne peut pas exécuter un programme destiné à un SPARC, et *vice versa*. Comme le temps d'accès à la mémoire est très largement supérieur à celui nécessaire pour exécuter une instruction, tous les processeurs contiennent des registres permettant de stocker des variables importantes et des résultats temporaires. L'ensemble des instructions classiques comprend des instructions pour charger un mot mémoire depuis la mémoire dans un registre et pour stocker le contenu d'un registre dans la mémoire. D'autres instructions combinent deux paramètres provenant de la mémoire, de registres ou des deux, et stockent le résultat obtenu en mémoire ou dans un registre.

Outre les registres ordinaires décrits précédemment, la plupart des ordinateurs disposent de registres spéciaux accessibles par le programmeur. L'un d'eux est le **compteur**

ordinal (*program counter*), qui contient l'adresse de la prochaine instruction à extraire de la mémoire. Ce compteur est mis à jour à chaque exécution d'instruction.

Dans un autre registre, on trouve le **pointeur de pile** (*stack pointer*), qui contient l'adresse courante du sommet de pile en mémoire. Cette pile contient un enregistrement par procédure dans laquelle le programme est entré et d'où il n'est pas encore ressorti. Chaque enregistrement contient les paramètres d'entrée, variables locales et temporaires de la procédure qui ne sont pas stockées dans des registres.

Un autre registre contient le **mot d'état** du programme (**PSW**, *Program Status Word*). Il comprend des bits de condition (positionnés par des instructions de comparaison), la priorité de l'UC, le mode (utilisateur ou noyau) et d'autres bits de contrôle. Les programmes utilisateur peuvent en principe lire le contenu entier de ce mot, mais n'ont accès en écriture qu'à un sous-ensemble de ses champs. Le PSW joue un rôle très important dans les appels système et les E/S.

Le système d'exploitation doit connaître l'ensemble des registres. Lors du multiplexage temporel du processeur, le système d'exploitation interrompra fréquemment un programme pour en démarrer un autre. À chaque interruption, le système doit sauvegarder tous les registres afin qu'ils soient restaurés quand le programme pourra à nouveau reprendre son exécution.

Pour améliorer les performances, les concepteurs de processeurs ont depuis longtemps abandonné ce modèle simple extraction-décodage-exécution en séquence. De nombreux processeurs modernes ont des caractéristiques qui leur permettent d'exécuter plus d'une instruction à la fois. Ainsi, une même UC peut posséder des unités séparées pour l'extraction, le décodage et l'exécution, de façon à pouvoir décoder l'instruction n+1 et extraire l'instruction n+2 pendant l'exécution de l'instruction n. Une telle structure est nommée **pipeline** et est décrite à la figure 1.7(a) dans le cas d'un pipeline à 3 niveaux ou étages. De plus longs pipelines ne sont pas rares. Dans la plupart des conceptions de pipelines, une fois l'instruction chargée dans le pipeline, elle est exécutée même si l'instruction précédente était un branchement conditionnel. La présence de pipelines rend la conception des compilateurs et des systèmes d'exploitation beaucoup plus compliquée.

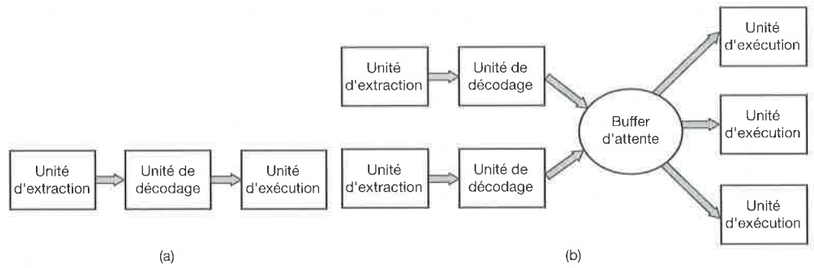

Figure 1.7 • (a) Un pipeline à trois niveaux. (b) Un processeur superscalaire.

Autre degré de sophistication au-dessus du pipe-line : les processeurs **superscalaires**, décrits à la figure 1.7(b). Dans cette approche, on dispose de plusieurs unités d'exécution, par exemple une pour l'arithmétique entière, une pour l'arithmétique flottante et une pour les opérations logiques. Deux ou plusieurs opérations sont extraites à la fois, décodées, et placées dans un buffer d'attente (une mémoire tampon) jusqu'au moment de leur exécution. Dès qu'une unité d'exécution est libre, elle consulte le buffer pour voir si une instruction de type correspondant est en attente ; si c'est le cas, elle l'extrait du buffer et l'exécute. La conséquence de cette approche est que les instructions sont souvent exécutées dans le désordre. La plupart du temps, il revient principalement au matériel de vérifier que le résultat obtenu est équivalent à celui fourni par une architecture séquentielle, mais une partie non négligeable (et complexe) du travail échoit tout de même au système d'exploitation, comme nous le verrons par la suite.

La plupart des processeurs, mis à part les plus simples d'entre eux, utilisés dans les systèmes embarqués ont deux modes de fonctionnement : le mode utilisateur et le mode noyau. Ce mode est en général mémorisé dans un des bits du PSW. En mode noyau, le processeur peut exécuter n'importe quelle instruction de son jeu et utiliser toutes les caractéristiques du matériel sous-jacent. Le système d'exploitation tourne en mode noyau, ce qui lui donne accès à l'ensemble du matériel.

En revanche, les programmes utilisateur tournent en mode utilisateur, qui ne permet l'accès qu'à un sous-ensemble des instructions et des ressources de la machine. En général, toutes les instructions relatives aux E/S et à la protection mémoire sont inaccessibles en mode utilisateur. Il est aussi évidemment interdit de basculer sur « noyau » le bit de mode du PSW.

Pour accéder aux services offerts par le système d'exploitation, un programme utilisateur doit effectuer un **appel système**, qui bascule en mode noyau et invoque le système d'exploitation. L'instruction *trap* effectue cette bascule. Quand le travail du système d'exploitation est terminé, le contrôle est rendu au programme utilisateur pour l'instruction suivante. Nous expliquerons les détails de gestion des appels système plus loin dans ce chapitre. Détail typographique : nous utiliserons une police à espacement fixe pour les appels système, comme dans le cas de `read`.

Il faut noter que les ordinateurs ont d'autres déroutements que l'instruction `trap` permettant d'effectuer un appel système. La plupart d'entre eux sont déclenchés par le matériel pour avertir d'une situation imprévue, comme par exemple une tentative de division par 0 ou un débordement de pile. Dans tous les cas, le système d'exploitation prend le contrôle et décide de la conduite à adopter. Parfois, le programme se termine sur une erreur. D'autres fois, l'erreur peut être ignorée (par exemple, un nombre flottant inférieur à la précision minimale peut être mis à 0). Enfin, quand le programme a annoncé au départ qu'il souhaitait prendre en charge certaines conditions exceptionnelles, le contrôle peut lui être donné le cas échéant.

Technique du multithread et circuits intégrés multicœurs

La **loi de Moore** affirme que le nombre de transistors d'un circuit intégré double tous les 18 mois. Cette loi n'est pas une loi physique comme la conservation de la quantité de mouvement, mais une loi qui résulte de l'observation faite par Gordon Moore, le cofondateur de la société Intel. Moore a observé de combien les ingénieurs concepteurs des circuits intégrés étaient capables de réduire la taille des transistors et ainsi d'en augmenter leur nombre sur une même surface de silicium. La loi de Moore a été vérifiée pendant plus de 30 ans. Elle devrait encore tenir pendant une dizaine d'années.

L'abondance de transistors est source de problème : que faire avec eux ? Nous avons vu ci-dessus une approche qui peut être un élément de réponse : l'architecture superscalaire dotée de plusieurs unités fonctionnelles. D'autres réponses sont possibles, comme créer de volumineux caches dans les processeurs. C'est ce qui s'est passé, mais finalement on a atteint le point de rendement décroissant. Alors que faire de plus ?

Une autre approche est la duplication ; non seulement d'unités fonctionnelles mais également de l'unité de commande. Le Pentium 4 et d'autres processeurs ont mis en place cette technique qui a donné lieu au **multithreading** ou **hyperthreading** (nom donné par Intel à cette technique). En première approximation, cette technique consiste à permettre à l'UC de maintenir les états de deux threads différents et ainsi de pouvoir passer de l'un à l'autre en un temps très court, de l'ordre de la nanoseconde. Un **thread** est une sorte de processus léger, qui correspond à un programme exécutable par le processeur. (Nous étudions en détail les threads au chapitre 2.) Par exemple si l'un des processus (ou thread) a besoin de lire un mot mémoire (ce qui peut prendre quelques cycles d'horloge), une UC dotée de la technique de multithreading peut passer très rapidement à l'autre processus pour éviter l'attente. Ce qui a pour effet d'améliorer les performances. Même si elle s'en approche, il ne faut toutefois pas confondre cette technique avec le parallélisme car en multithreading un seul processus est exécuté à la fois.

Le multithreading a une forte conséquence sur le système d'exploitation, car chaque thread lui apparaît comme une UC séparée. Considérons par exemple un système qui comprend deux UC, chacune traitant deux threads. Le système d'exploitation voit cela comme quatre UC. S'il y a suffisamment de travail pour maintenir les deux UC occupées à un instant donné, le système d'exploitation peut, par mégarde, faire exécuter les deux threads sur la même UC, laissant la seconde UC oisive. Ce choix est bien moins efficace que de faire exécuter un thread par chaque UC. Le successeur du Pentium 4, le Core 2, ne pratique pas la technique de l'hyperthreading, mais Intel a annoncé que le successeur du Core 2 la mettrait en œuvre.

Au-delà du multithreading on trouve des UC qui disposent de deux ou quatre processeurs intégrés complets, voire plus. On dit de ces circuits qu'ils sont **multicœurs**. Le circuit multicœur de la figure 1.8 comprend quatre processeurs, chacun disposant d'une UC indépendante. (Nous expliquons le rôle des caches ci-dessous.) L'utilisation de ces circuits multicœurs nécessite impérativement un système d'exploitation multiprocesseur.

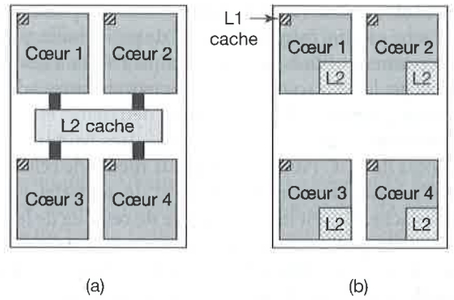

Figure 1.8 • (a) Un circuit à quatre cœurs avec un cache L2 partagé. (b) Un circuit à quatre cœurs avec des caches L2 indépendants.

1.3.2 La mémoire

Le second constituant fondamental de tout ordinateur est la mémoire. Dans l'idéal, la mémoire devrait être extrêmement rapide (plus rapide que le temps d'exécution d'une instruction, de façon à ce que l'UC ne soit pas freinée par les accès mémoire), disponible en grande quantité et peu onéreuse. Aucune des technologies actuelles ne satisfait ces trois critères, il faut donc biaiser. La mémoire est construite comme une hiérarchie de couches, comme le montre la figure 1.9. Les mémoires de la couche supérieure sont très rapides, de faible capacité et présentent des coûts par bit bien plus élevés que celles des couches inférieures.

La couche supérieure est constituée des registres internes à l'UC. Ils sont fabriqués dans les mêmes matériaux que le processeur et sont donc du même niveau de performance. Leur capacité de stockage est de l'ordre de 32×32 bits (sur un processeur 32 bits) ou 64×64 bits (sur un processeur 64 bits). Elle est en tout cas inférieure à 1 Ko. Les programmes gèrent les registres eux-mêmes (c'est-à-dire décident de ce qu'ils y stockent).

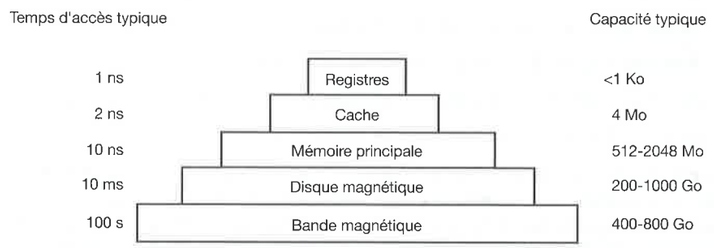

Figure 1.9 • Un découpage hiérarchique classique de la mémoire. Les valeurs mentionnées sont approximatives.

Vient ensuite la **mémoire cache** (ou simplement cache), principalement contrôlée par le matériel. Un cache est une mémoire rapide de petite taille qui contient les mots mémoire les plus récemment utilisés, accélérant ainsi l'accès à ces mots. Un cache est divisé en **lignes de cache**, le plus souvent de 64 octets chacune, adressées de manière contiguë (0 à 63 pour la première, 64 à 127 pour la deuxième, etc.). Les lignes les plus fréquemment utilisées sont stockées dans un cache à très hautes performances, généralement situé tout près de l'UC (voire à l'intérieur même de celle-ci). Quand le programme a besoin d'accéder à un mot mémoire, le matériel vérifie que la ligne demandée est dans le cache. Si c'est le cas, on parle de réussite de lecture ou **cache hit**. La requête étant satisfaite par le cache, aucun accès à la mémoire principale n'est effectué. Un cache hit prend en général deux cycles d'horloge. Si la ligne n'est pas dans le cache, on parle d'échec de lecture ou **cache miss** ; on doit alors accéder à la mémoire principale, au prix d'une perte de temps non négligeable. La taille du cache est limitée par le prix élevé de ce type de mémoire. Certaines machines ont deux, voire trois niveaux de cache, chaque niveau étant plus lent mais de taille plus importante que le précédent.

Les caches jouent un rôle essentiel dans les systèmes informatiques, et pas seulement pour améliorer les performances entre le processeur et la mémoire principale. Mais, par exemple, à chaque fois qu'on peut diviser une ressource en plusieurs éléments (ou blocs) et que certains éléments sont plus fréquemment utilisés que d'autres, le cache apparaît comme améliorant les performances. Les systèmes d'exploitation utilisent sans cesse des caches. C'est ainsi que la plupart conservent en mémoire centrale les blocs des fichiers les plus souvent référencés plutôt que de les extraire du disque à maintes reprises. C'est une technique de type cache. De façon similaire, le résultat de la conversion d'un long chemin d'accès comme

```
/home/ast/projects/minix3/src/kernel/clock.c
```

en une adresse disque où se trouve le fichier référencé peut être mise en cache pour éviter de refaire la conversion à chaque utilisation. De même, lorsque l'adresse d'une page Web (l'URL) est convertie en adresse IP, cette dernière peut être conservée en cache pour une utilisation future.

Dès qu'on utilise un système de cache, plusieurs questions se posent :

1. Quand mettre un nouvel item dans le cache ?
2. Dans quelle ligne de cache insérer le nouvel item ?
3. Quel item sortir du cache quand on a besoin de place pour en mettre un nouveau ?
4. Où ranger l'item expulsé du cache en mémoire ?

Chaque question ne concerne pas forcément tous les types de caches. Dans le cache d'une UC, un item est systématiquement inséré dans la ligne du cache à chaque fois qu'une situation d'échec de lecture (*cache miss*) apparaît. La ligne de cache à utiliser est souvent calculée à partir de quelques bits de poids fort de l'adresse mémoire référencée. Par exemple avec un cache de 4 096 lignes de 64 octets et 32 bits d'adresse, les bits 6 à 17 de l'adresse peuvent être utilisés pour déterminer la ligne du cache, et les bits 0 à 5 pour la position de l'octet dans la ligne. Dans ce cas, l'item à

enlever est celui qui est rangé à la position que l'on va occuper, mais dans d'autres systèmes ce ne serait pas si simple. Cependant, quoi qu'il en soit, lorsqu'on récrit une ligne de cache en mémoire, l'emplacement mémoire de réécriture est uniquement déterminé par l'adresse en question.

La plupart des UC actuelles ont deux niveaux de caches. Le premier niveau, L1, est interne au processeur et comprend le plus souvent deux caches, un cache d'instructions et un cache de données. Chaque cache fait en général 16 Ko. Il y a, en outre, très souvent un cache de second niveau, L2, entre les caches de niveau 1 et la mémoire principale. Sa capacité est de l'ordre de quelques mégaoctets. Il contient les mots mémoire les plus récemment utilisés. La différence ente les deux niveaux tient au facteur temps : avec L1 les accès se font sans délai alors qu'il faut un ou deux cycles d'horloge avec L2.

À la figure 1.8(a), chaque cœur compte un cache L1 privé et peut accéder à un cache L2 partagé par tous. C'est l'approche utilisée par les circuits multicœurs d'Intel. En revanche, à la figure 1.8(b), chaque cœur dispose de deux caches privés, L1 et L2. C'est l'approche utilisée par AMD. Chaque stratégie a ses partisans et ses détracteurs. L'approche d'Intel nécessite un contrôleur de cache L2 plus complexe que dans celle retenue par AMD, qui au contraire rend plus complexe la garantie de cohérence des caches L2.

On trouve ensuite sur la hiérarchie de la figure 1.9 la mémoire principale, point central du système de mémoire. Elle est souvent appelée mémoire **RAM** (*Random Access Memory*). Les anciens la nomment souvent *core memory* (mémoire noyau), car les ordinateurs des années 1950 et 1960 utilisaient de minuscules noyaux de ferrite magnétisable pour la mémoire principale. À l'heure actuelle, la capacité de ces mémoires s'étend de quelques centaines de mégaoctets à plusieurs gigaoctets et croît rapidement. Toutes les requêtes non satisfaites par les accès aux caches sont reportées ensuite vers la mémoire principale.

En plus de la mémoire RAM, qui est volatile, la plupart des ordinateurs disposent d'une quantité relativement faible de mémoire non volatile. Celle-ci, contrairement à la RAM, est une mémoire qui ne perd pas ses données lorsque l'alimentation électrique est coupée. Il s'agit de la mémoire **ROM** (*Read Only Memory*) qui ne peut être que lue. Programmé une fois pour toutes lors de sa fabrication en usine, le contenu d'une ROM ne peut pas être modifié. Comme la RAM, elle est très rapide et peu coûteuse. En général, on trouve dans ces mémoires ROM le *bootstrap* qui est le programme d'amorçage servant à démarrer la machine. De même, certains programmes de gestion de périphériques se trouvent eux aussi en mémoire ROM, sur des cartes d'entrées/sorties.

Il existe d'autres mémoires non volatiles : les **EEPROM** (*Electrically Erasable programmable ROM*, mémoire en lecture seule mais programmable après effacement électrique) et les **mémoires Flash** qui, contrairement aux ROM, peuvent être effacées et reprogrammées plusieurs fois. Le temps d'écriture dans ces mémoires étant beaucoup plus important que dans les mémoires RAM, on les utilise comme des mémoires ROM.