

Systemes informatiques et applications concurrentes

Partie 1 - Introduction et rappels

Ivan KURZWEG

24 février 2010

Systemes informatiques et applications concurrentes

by Ivan KURZWEG

Copyright © 2010 Ivan KURZWEG, ivan.kurzweg@gmail.com, 2010

Permission to use, copy, modify, and distribute this documentation *for any purpose with or without fee* is here by granted, provided that *the above copyright notice and this permission notice appear in all copies*.

The documentation is provided "as is" and the author disclaims all warranties with regard to this documentation including all implied warranties of merchantability and fitness. In no event shall the author be liable for any special, direct, indirect, or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this documentation.

Table des matières

1 Objectifs généraux du cours	1
1.1 Cadre pédagogique	1
1.2 Plan	1
1.3 Environnement de travail	1
1.4 Prérequis	1
2 Rappels sur la structure d'un ordinateur	1
2.1 Architecture générale d'un système informatique	2
2.2 Le processeur	2
2.3 Le système mémoire	3
2.4 Le système d'entrées sorties	4
2.4.1 Le mode programmé	4
2.4.2 Accès direct à la mémoire	4
2.4.3 Processeur spécialisé	5
2.4.4 Les interruptions	5
2.5 Exécution multimode	5
2.6 Quelques questions pour aller un peu plus loin	5
3 Principes généraux d'un système d'exploitation	6
3.1 Fonctions d'un système d'exploitation	6
3.2 Classes de systèmes	7
3.3 Unix et <i>FreeBSD</i>	8
3.3.1 Philosophie	8
3.3.1.1 Tout est fichier	8
3.3.1.2 Les données sont du texte	8
3.3.1.3 KISS - <i>Keep it simple, stupid!</i>	8
3.3.1.4 Faire une chose et le faire bien	8
3.3.2 Un peu d'histoire	8
3.3.3 La licence <i>BSD</i>	9
3.3.4 Sources complémentaires	9
4 Rappels de programmation en Langage C	9
4.1 Chaîne de production des exécutables	9
4.2 Illustration	10
4.2.1 Compilation	11
4.2.2 L'édition de liens	12
4.2.3 Exécution	13
4.3 Exercices de révision	14
5 Références	14
5.1 Références	14

Table des figures

1 Structure d'un ordinateur	2
2 Diagramme de bloc d'un processeur	3
3 DMA	4
4 Fonctions d'un OS	7
5 Histoire des Unices - Source Wikipedia	9
6 Chaîne de production d'un exécutable	10

Résumé

Partie 1 du cours "Systèmes informatiques et applications concurrentes". Cette première partie du cours présente les objectifs du cours, puis les grands principes des systèmes d'exploitation. Nous y rappellerons quelques concepts sur la structure de l'ordinateur et sur la programmation en C

1 Objectifs généraux du cours

1.1 Cadre pédagogique

Ce cours a été réalisé dans le cadre de l'enseignement de l'UE SMB137 du *Cnam*, cours dispensé en Licence 3 STIC. Il peut également être une introduction à la programmation système.

Les objectifs visés sont ceux fournis par la fiche d'information du *Cnam* :

- Permettre de comprendre les principaux paradigmes et "patterns" de la programmation concurrente
- Obtenir des bases pour des compétences en compréhension du fonctionnement des systèmes informatiques
- Obtenir des bases pour des compétences en compréhension de la programmation d'applications comportant des processus coopérants, locaux ou distants

Il s'agit donc d'un cours plutôt théorique, mais il sera proposé dans la mesure du possible des exemples de mises en oeuvre des concepts sous forme de programmes en langage C.

1.2 Plan

Le programme de l'ensemble du cours est le suivant :

1. Présentation des systèmes d'exploitation, rappels sur la programmation C en environnement Unix
2. Processus et Threads, gestion et ordonnancement
3. Gestion de la mémoire
4. Pagination de la mémoire virtuelle
5. Gestion de fichiers
6. Exclusion mutuelle
7. Interblocage
8. Moniteurs

1.3 Environnement de travail

Il semble donc intéressant de proposer une implémentation des concepts en environnement Unix.

Les différents exemples fournis ont été écrits, compilés et exécutés (dans la mesure du possible et sauf indications contraires) sur un système *FreeBSD* 8.0. En passant, et si ce n'est déjà fait, je vous invite à utiliser ce système d'exploitation, "Unix-like" libre (*Open Source*), dont la robustesse, la fiabilité et les performances en ont fait un des systèmes majeurs dans le domaine des serveurs Internet (Web, mail, ...).

Si vous souhaitez vous lancer dans l'aventure *FreeBSD*, il va de soit que vous pourrez me solliciter en cas de besoin sur l'installation et la configuration de ce système d'exploitation.

1.4 Prérequis

Pour aborder ce cours, vous devez idéalement avoir les compétences et connaissances suivantes :

- connaissance de l'architecture des ordinateurs (nous en rappelons quelques concepts dans le chapitre suivant)
- maîtrise du langage C (quelques rappels sur la syntaxe, et la compilation sont en chapitre 4)
- maîtrise d'un système *Unix-like* (BSD, Linux, ...). Vous trouverez de nombreuses ressources sur le Web.

2 Rappels sur la structure d'un ordinateur

Ce chapitre rappelle les bases de l'architecture matérielle des ordinateurs, exposant ainsi les problématiques auxquelles les systèmes d'exploitation modernes doivent faire face.

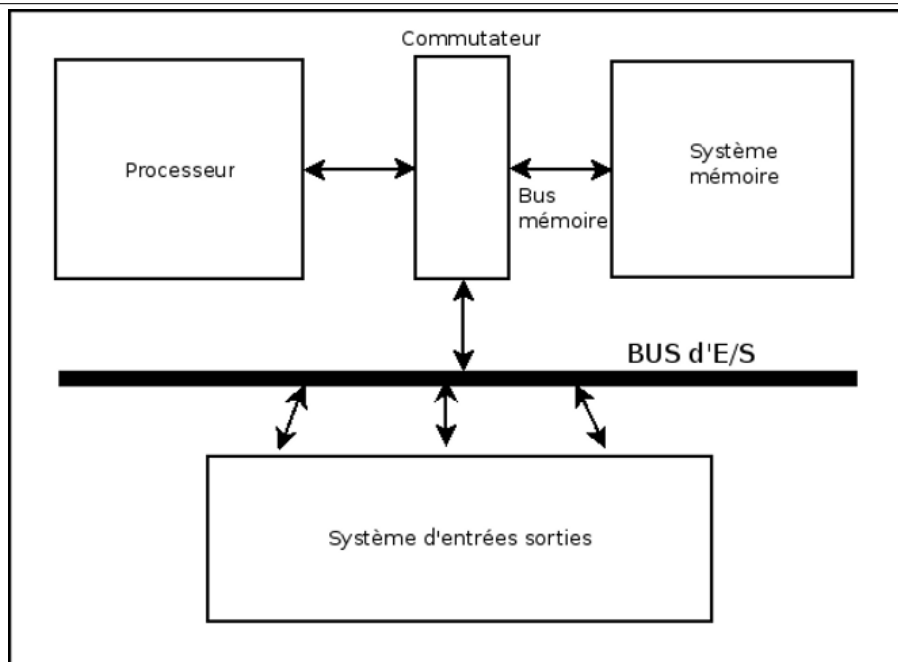
2.1 Architecture générale d'un système informatique

La plupart des systèmes informatiques peuvent être divisés en trois sous-systèmes :

- Processeur
- mémoire
- sous-système d'E/S

La figure ci-dessous présente les relations entre ces sous-systèmes :

FIG. 1 Structure d'un ordinateur

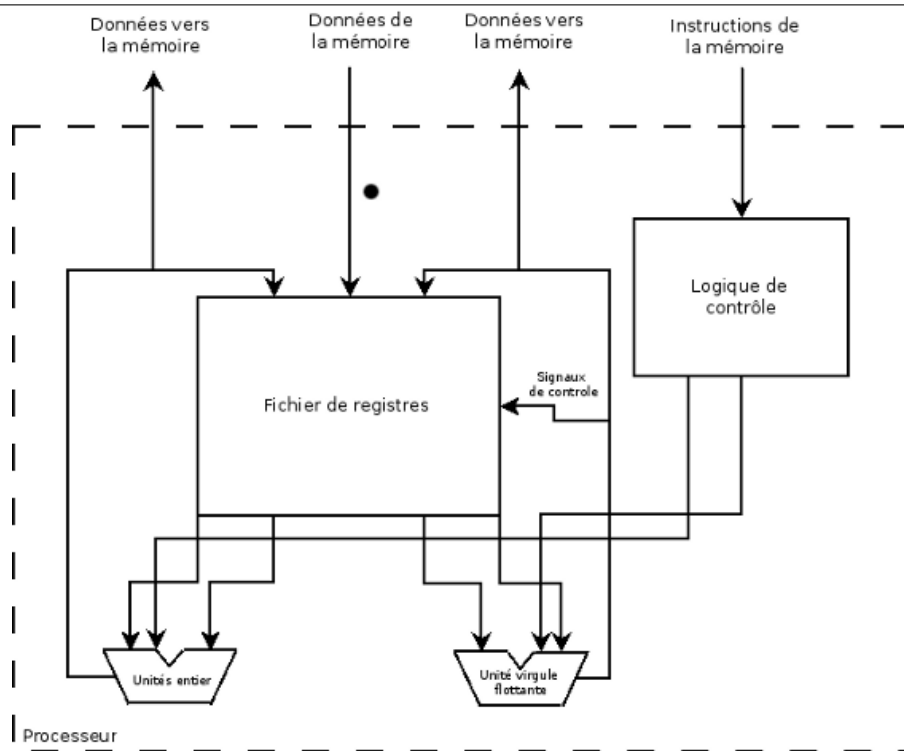


Le processeur a pour charge d'exécuter les programmes, la mémoire fournit l'espace de stockage, tandis que le sous-système d'E/S permet le contrôle des appareils tels que disques durs, lecteurs optiques, cartes vidéos, ... Le commutateur (il peut être intégré au processeur) permet les échanges de données via les bus d'E/S et mémoire. Cette structure est globalement la même depuis sa définition par **John Von Neuman** en 1945.

2.2 Le processeur

Les processeurs sont constitués de plusieurs blocs : unités d'exécutions, fichiers de registres et logique de contrôle. Les unités d'exécution contiennent le dispositif matériel qui exécute les instructions : les *Unités Arithmétiques et Logiques (UAL)*, capables d'effectuer les opérations élémentaires habituelles sur des valeurs binaires (addition, soustraction, ET, OU, ...)

FIG. 2 Diagramme de bloc d'un processeur



Le *fichier de registres* est une zone de stockage que le processeur utilise pour stocker des données, de manière plus efficace que le stockage en mémoire centrale. Les registres sont ainsi généralement utilisés pour mémoriser des résultats intermédiaires ou des états particuliers du processeur. La *logique de contrôle* contrôle le reste du processeur, en déterminant le moment auquel les instructions sont exécutées et quelles opérations doivent être exécutées pour chaque instruction.

Les instructions sont en général assez rudimentaires : ce sont essentiellement des opérations de transfert de données entre les registres et l'extérieur du processeur (mémoire ou périphérique), ou des opérations arithmétiques ou logiques avec un ou deux opérandes. Pour ces dernières opérations un registre particulier, l'accumulateur, est souvent utilisé implicitement comme l'un des opérandes et comme résultat. En général le déroulement de l'instruction entraîne l'incréméntation du compteur ordinal, et donc l'exécution de l'instruction qui suit.

La tendance naturelle a été de construire des processeurs avec un jeu d'instructions de plus en plus large; on pensait alors que le programmeur utiliserait les instructions ainsi disponibles pour améliorer l'efficacité de ses programmes. Ceci a conduit à ce que l'on a appelé l'architecture CISC (*Complex Instruction Set Computer*). Cependant on a constaté que les programmes contenaient toujours les mêmes instructions, une partie importante du jeu d'instructions n'étant utilisée que très rarement. Une nouvelle famille de processeurs a alors été construite, l'architecture RISC (*Reduced Instruction Set Computer*), qui offre un jeu réduit d'instructions simples mais très rapides.

2.3 Le système mémoire

Le système mémoire agit à la manière d'un réceptacle de stockage pour les données et les programmes. La mémoire centrale (*mémoire vive*, ou encore *mémoire à accès aléatoire*, RAM) est divisée en une série d'emplacements de stockage, chacun étant numéroté (son *adresse*). L'une des caractéristiques des systèmes informatiques est ainsi la largeur des adresses qu'ils utilisent, car elle limite la quantité de mémoire utilisable (on parle de systèmes 32 bits, ou 64 bits).

La conception du système mémoire possède un impact déterminant sur la performance du système informatique, et se trouve souvent être le facteur limitant pour la vitesse d'exécution des applications. La bande passante et la latence sont capitales pour la performance des applications. Nous verrons également dans le chapitre qui lui est consacrée que la gestion de la mémoire implique également des notions de protection et de mécanisme de partage entre les processus.

2.4 Le système d'entrées sorties

Les entrées sorties permettent l'échange de données entre deux constituants physiques, et nécessitent trois liaisons :

- les données elles mêmes
- une liaison permettant à l'émetteur de signaler la présence de la donnée
- une liaison permettant au récepteur de signaler qu'il a lu la donnée.

Il existe trois manières d'implémenter ce principe : le mode programmé, le mode DMA, et les processeurs spécialisés.

2.4.1 Le mode programmé

Dans ce mode, l'unité d'échange qui interface le périphérique au bus et au processeur, ne sait pas délivrer d'informations sur son état. Pour savoir si l'unité d'échange est prête à recevoir ou délivrer une nouvelle donnée, le processeur doit régulièrement lire le contenu du registre d'état de l'unité d'échange. Le processeur est donc complètement occupé durant l'entrée sortie par la réalisation de cette boucle de scrutation dont la logique est donnée ci-dessous suivante en mémoire centrale. On ne peut donc concevoir qu'un système monoprogrammé dans ce cas.

```

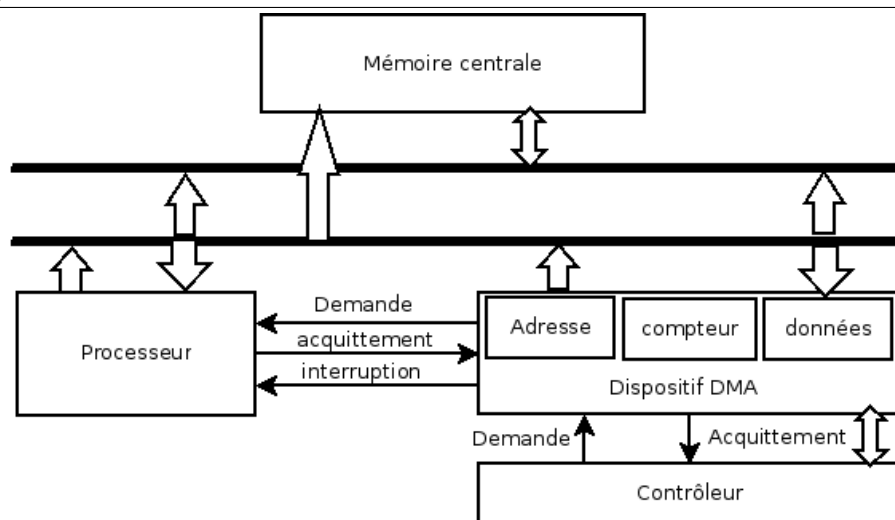
tantque il_y_a_des_données_à_lire faire
  tantque donnée_suivante_non_prête faire fait; { attente de la donnée}
  lire_la_donnée;
  traitement_de_la_donnée;
fait

```

2.4.2 Accès direct à la mémoire

Pour accroître le débit des entrées-sorties, et diminuer la monopolisation du processeur dont nous avons parlé ci-dessus, la première solution a été de déporter un peu de fonctionnalités dans le dispositif qui relie le périphérique au bus, de façon à lui permettre de ranger directement les données provenant du périphérique en mémoire dans le cas d'une lecture, ou d'extraire directement ces données de la mémoire dans le cas d'une écriture. C'est ce que l'on appelle *l'accès direct à la mémoire*, DMA (*Direct Memory Access*)

FIG. 3 DMA



L'exécution d'un transfert se déroule de la manière suivante :

1. Le processeur informe le dispositif d'accès direct à la mémoire de l'adresse en mémoire où commence le tampon qui recevra les données s'il s'agit d'une lecture, ou qui contient les données s'il s'agit d'une écriture. Il informe également le dispositif du nombre d'octets à transférer.

2. Le processeur informe le contrôleur des paramètres de l'opération proprement dite.
3. Pour chaque donnée élémentaire échangée avec le contrôleur, le dispositif demande au processeur le contrôle du bus, effectue la lecture ou l'écriture mémoire à l'adresse contenue dans son registre et libère le bus. Il incrémente ensuite cette adresse et décrémente le compteur.
4. Lorsque le compteur atteint zéro, le dispositif informe le processeur de la fin du transfert par une ligne d'interruption.

Pendant toute la durée du transfert, le processeur est libre d'effectuer une autre tâche.

2.4.3 Processeur spécialisé

La troisième façon de relier un périphérique avec la mémoire est d'utiliser un processeur spécialisé d'entrées-sorties : déléguer plus d'automatisme à ce niveau. Dans le schéma précédent, le processeur principal avait en charge la préparation de tous les dispositifs, accès direct à la mémoire, contrôleur, périphérique, etc..., pour la réalisation de l'opération. L'utilisation d'un processeur spécialisé a pour but de reporter dans ce processeur la prise en charge de cette préparation, mais aussi des opérations plus complexes, telles que par exemple le contrôle et la reprise d'erreurs. L'intérêt est de faire exécuter des tâches de bas niveau par un processeur moins performant, et donc moins coûteux à réaliser, tout en réservant le processeur principal à des tâches plus "nobles".

2.4.4 Les interruptions

Une interruption est un signal envoyé au processeur par un périphérique extérieur. Sorte de "bi-peur" du processeur, elle indique à ce dernier d'interrompre l'ensemble de ses activités en cours pour répondre aux besoins du périphérique à l'origine de l'interruption. Le processeur termine le traitement de l'instruction en cours, et contrôle les interruptions.

Le processeur répond à l'interruption en stockant la valeur en cours du compteur ordinal, qui lui permettra de reprendre l'exécution au point où l'interruption s'est produite. La valeur du compteur ordinal (et d'autres indicateurs complémentaires) est stockée dans un registre d'état, le PSW (*Program Status Word*). Le contenu de l'ensemble des registres est également sauvegardé, c'est ce que l'on appelle la sauvegarde du contexte.

Enfin, les interruptions peuvent être associées à un niveau de priorité matérielle, et masquées. Ceci permet de résoudre des cas d'interruptions survenant lors du traitement d'une précédente interruption. Effectivement, sans ces mécanismes, le contexte d'un programme pourrait être perdu.

2.5 Exécution multimode

Nous venons de voir qu'il est nécessaire de contrôler les actions d'un programme vis-à-vis de son environnement direct. On doit donc restreindre les privilèges des programmes, et confier au système d'exploitation des activités réservées (exécution d'instructions privilégiées, accès à certains registres, à certains périphériques d'E/S, ...). Les processeurs possèdent donc deux modes de fonctionnement, le *mode superviseur* (ou *maître*), et le *mode utilisateur* (ou *esclave*). Il existe trois manières de passer en mode superviseur :

- une instruction spéciale : l'appel système ou Supervisor Call (SVC)
- les interruptions
- les déroutements (conditions anormales détectés par le processeur)

Dans le mode utilisateur, chaque instruction est contrôlée avant son exécution, pour vérifier si elle est autorisée.

2.6 Quelques questions pour aller un peu plus loin ...

- Supposons que le système mémoire d'un ordinateur n'ait pas la propriété de l'accès aléatoire (RAM), c'est-à-dire que les références mémoires prennent un temps différents à s'accomplir selon l'adresse qu'elles référencent. Comment cela compliquerait-il le processus de développement des programmes ?
- Que fait le processeur quand il n'y a pas de programme à exécuter ?
- Parmi les instructions suivantes, lesquelles doivent être privilégiées ?

- a. Changement des registres de gestion de mémoire
 - b. Ecriture du compteur de programme
 - c. Lecture de l'horloge
 - d. Réglage de l'horloge
 - e. Changement de priorités du processeur
- Pourquoi un ordinateur doit-il démarrer en mode superviseur lors de sa première mise sous tension ?

3 Principes généraux d'un système d'exploitation

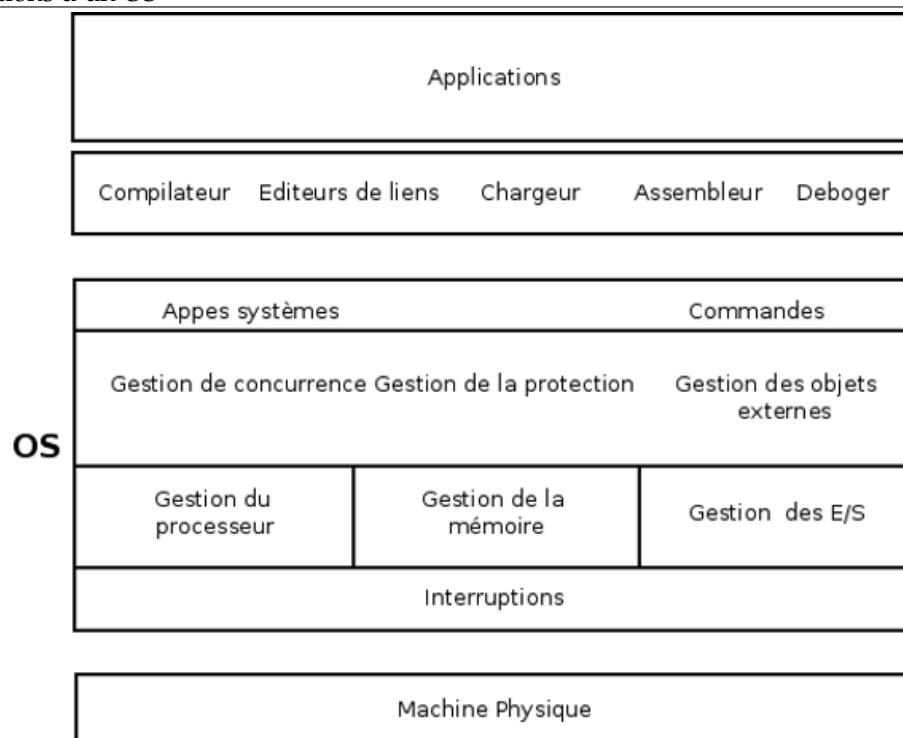
Le système d'exploitation est le logiciel qui prend en charge les fonctionnalités élémentaires du matériel et qui propose une plate-forme plus efficace en vue de l'exécution des programmes. Il gère les ressources matérielles, offre des services pour accéder à ces ressources, et crée des éléments abstraits de niveau supérieur, tels que des fichiers, des répertoires et des processus.

3.1 Fonctions d'un système d'exploitation

Le système d'exploitation se présente donc comme une couche logicielle placée entre la machine matérielle et les applications. Il s'interface avec la couche matérielle, notamment par le biais du mécanisme des interruptions. Il s'interface avec les applications par le biais des primitives qu'il offre : appels système et commandes. Le système d'exploitation peut être découpé en plusieurs grandes fonctions :

- *Gestion du processeur* : le système doit gérer l'allocation du processeur aux différents programmes pouvant s'exécuter. Cette allocation se fait par le biais d'un algorithme d'ordonnancement qui planifie l'exécution des programmes. Selon le type de système d'exploitation, l'algorithme d'ordonnancement répond à des objectifs différents
- *Gestion de la mémoire* : le système doit gérer l'allocation de la mémoire centrale entre les différents programmes pouvant s'exécuter. Comme la mémoire physique est parfois trop petite pour contenir la totalité des programmes, la gestion de la mémoire se fait selon le principe de la mémoire virtuelle : à un instant donné, seules sont chargées en mémoire centrale, les parties de code et données utiles à l'exécution
- *Gestion des entrées/sorties* : le système doit gérer l'accès aux périphériques, c'est-à-dire faire la liaison entre les appels de haut niveau des programmes utilisateurs et les opérations de bas niveau de l'unité d'échange responsable du périphérique (unité d'échange clavier) : c'est le pilote d'entrées/sorties (driver) qui assure cette correspondance
- *Gestion de la concurrence* : Comme plusieurs programmes coexistent en mémoire centrale, ceux-ci peuvent vouloir communiquer pour échanger des données. Par ailleurs, il faut synchroniser l'accès aux données partagées afin de maintenir leur cohérence. Le système offre des outils de communication et de synchronisation entre programmes
- *Gestion des objets externes* : La mémoire centrale est une mémoire volatile. Aussi, toutes les données devant être conservées au-delà de l'arrêt de la machine, doivent être stockées sur une mémoire de masse (disque dur, disquette, cédérom...). La gestion de l'allocation des mémoires de masse ainsi que l'accès aux données stockées s'appuient sur la notion de fichiers et de système de gestion de fichiers (SGF).
- *Gestion de la protection* : le système doit fournir des mécanismes garantissant que ses ressources (CPU, mémoire, fichiers) ne peuvent être utilisées que par les programmes auxquels les droits nécessaires ont été accordés. Il faut notamment protéger le système et la machine des programmes utilisateurs (mode d'exécution utilisateur et superviseur)

FIG. 4 Fonctions d'un OS



3.2 Classes de systèmes

- Les premiers temps de l'informatique ne permettaient guère aux utilisateurs d'avoir un accès direct aux ordinateurs. Les entrées (programmes et données) étaient préparées sur bande ou carte perforée. Les utilisateurs donnaient leur entrée à un opérateur, et récupéraient la sortie plus tard. L'opérateur assemblait les tâches similaires par lots, puis les exécutaient par le biais de l'ordinateur. Chaque tâche disposait d'un contrôle total de la machine pendant toute son exécution.

Le système d'exploitation de ce type de machine est en mode de *traitement par lots* (*batch mode*). Les fonctionnalités de ces systèmes sont souvent minimales, sauf quand ils offrent en plus un service de temps partagé :

- Les systèmes *par lots en multiprogrammation* (*multiprogram batch system*) lisent les tâches à exécuter sur le disque. Lorsqu'une tâche n'est pas en mesure de s'exécuter (attente d'une E/S), un autre tâche prend le relais.
- Les systèmes à *temps partagé* (*time-shared*), ou encore *systèmes multi-utilisateurs interactifs*, autorisent des interactions entre l'utilisateur et le processus. Dans les systèmes par lots, toutes les données sont fournies au moment où le programme est saisi. Lorsqu'une interaction avec l'utilisateur est nécessaire, le système d'exploitation doit gérer un environnement qui permet aux programmes de répondre aux saisies dans un laps de temps convenable. Le système d'exploitation doit non seulement partager des ressources, mais il doit agir comme si les processus s'exécutaient simultanément. Pour cela il bascule très rapidement d'un processus actif à un autre.
- La mise en réseau des ordinateurs implique aujourd'hui des communications via des protocoles de plus en plus complexes. On parle donc de système d'exploitation *en réseau* (*networked*). Le système d'exploitation universel d'aujourd'hui est donc un système à temps partagé en réseau.
- Un système d'exploitation en *temps réel* (*real-time*) est conçu pour prendre en charge l'exécution de tâches dans le cadre de contraintes liées à l'horloge. Le système d'exploitation doit ainsi garantir que la tâche peut-être exécutée dans un délai spécifié. Comme ces systèmes sont souvent interfacés à un environnement dynamique (procédé) délivrant des événements synchrones ou asynchrones auxquels ils doivent réagir, on parle aussi de systèmes réactifs. Ce sont des systèmes adaptés à la commande de procédé.
- Les systèmes d'exploitation *répartis* (*distributed*) gèrent l'ensemble des ressources des machines en réseau. Les système d'exploitation de toutes les machines travaillent donc en collaboration pour

gérer les ressources collectives. Un seul système d'exploitation gère les ressources du réseau, qui sont fournies par chaque ordinateur, ou *noeud*.

3.3 Unix et FreeBSD

Tout est fichier

3.3.1 Philosophie

3.3.1.1 Tout est fichier Dans le monde Unix, les périphériques sont des fichiers qui se trouvent dans `/dev`, les processus sont des fichiers qui se trouvent dans `/proc`, les paramètres et données du noyau sont des fichiers qui se trouvent dans `/sys` et accessoirement `/proc`, les exécutable sont des fichiers normaux. Une telle vision des choses apporte énormément de souplesse dans l'approche de la programmation.

3.3.1.2 Les données sont du texte Dans la philosophie unix, toutes les données doivent être stockées et transmises sous forme de texte, ce qui apporte de nombreux avantages :

- Un fichier texte peut être lu par les autres outils Unix, et donc respecte de principe faire une chose et le faire bien (cf. ci-dessous)
- Un fichier texte peut être lu par un être humain et donc respecte le principe KISS (cf. ci-dessous)
- Un fichier texte permet une interopérabilité avec d'autres systèmes
- Un fichier texte facilite le débogage

3.3.1.3 KISS - *Keep it simple, stupid!* Le principe est de disposer d'outils très simples, faciles à comprendre, à manipuler, et à maintenir. Le code source doit également rester simple, "stupide" ... Un tel code, même sans trop de commentaires, sera simple à lire, à maintenir et à corriger.

3.3.1.4 Faire une chose et le faire bien Unix a inventé le pipe, une méthode simple pour faire communiquer deux processus entre eux. Cette communication est unidirectionnelle et non formatée, seules des données brutes y passent.

S'y ajoute les sockets, qui permettent une communication bidirectionnelle, et d'autres moyens de communication inter-processus. Ces mécanismes permettent de développer des outils sachant facilement communiquer avec ceux existants, qui normalement savent faire UNE chose, mais bien ...

3.3.2 Un peu d'histoire

Tout commence dans les années 1970, avec le besoin d'AT&TTM de développer un grand nombre de logiciels. Ne pouvant les vendre (le groupe ne faisait pas partie de l'industrie du logiciel), AT&TTM délivra des licences et les codes sources à très bon prix aux universités. Ainsi, Unix fut pratiqué par toute une génération de chercheurs et d'étudiants, qui l'améliorèrent et le corrigèrent (*Job control, Fast File System, ...*).

Le groupe de recherche en informatique de l'université (*Computer Science Research Group, CSRG*) de Berkeley en Californie centralisa les différents apports de chaque université, les évaluant, puis les redistribuant gratuitement aux possesseurs de licences AT&TTM valides. Ainsi, cette collection de logiciel (comprenant TCP/IP par exemple) fût rapidement connue sous le nom de BSD (*Berkeley Software Distribution*).

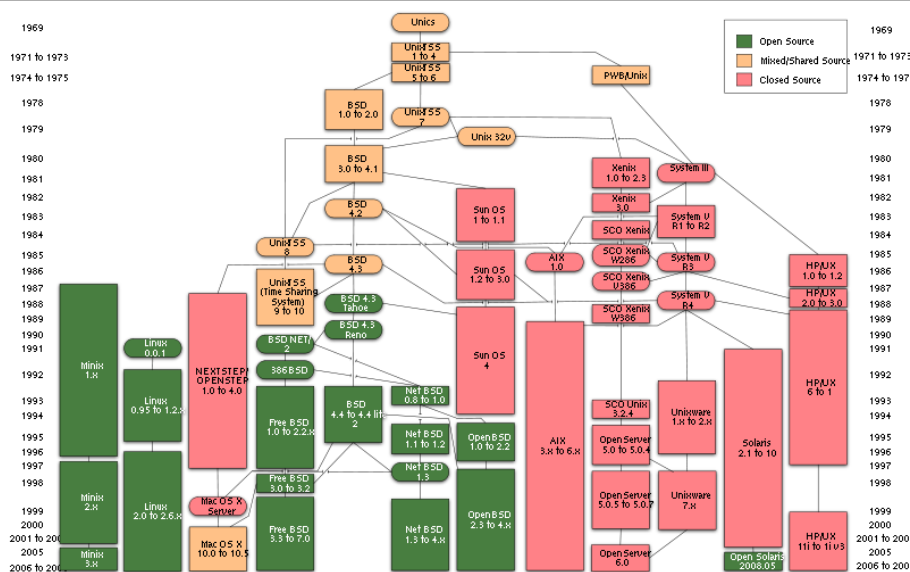
Pour différentes raisons (suivez les liens en fin de chapitre pour plus d'informations), en 1992, le code BSD avec toutes ses améliorations (la quasi totalité du code original avait été remplacé) fut publié pour le grand public, sous la licence BSD (voir paragraphe suivant).

A partir de ce code source, plusieurs projets vont débiter : FreeBSD et NetBSD entre autres. La qualité du système a rapidement fait ses preuves, et FreeBSD est aujourd'hui utilisé sur Internet par quelques-unes des sociétés les plus vitales et les plus visibles sur le net. Yahoo!TM par exemple repose quasiment entièrement sur FreeBSD. IBM, Nokia, Juniper, NetApp, ... utilisent FreeBSD sur des systèmes embarqués. Mac OSX utilise de larges portions de code dans son noyau.

Au moment de la rédaction de ce cours, FreeBSD en est à la version 8.0. Sa collection de logiciels n'a rien à envier au monde Linux (18000 applications portées sur FreeBSD), et dans tous les cas un mode de compatibilité Linux permet de faire tourner toutes les applications souhaitées.

Enfin, il existe d'autres distributions *BSD* : *NetBSD* (reconnue pour sa portabilité vers des architectures matérielles exotiques), *OpenBSD* (reconnue pour son niveau de sécurité), *DragonFlyBSD* (pour sa gestion des multiprocesseurs), etc ...

FIG. 5 Histoire des Unices - Source Wikipedia



3.3.3 La licence BSD

Cette licence, la plus libérale de toutes, peut se résumer comme ceci :

- Ne prétendez pas avoir écrit le code
- Ne nous rendez pas responsables si cela ne fonctionne pas
- N'utilisez pas notre nom pour vendre vos produits

Comparé à d'autres types de licences libres (GNU par exemple), la licence *BSD* permet l'intégration de parties de codes *BSD* dans des logiciels propriétaires, sans obligation ni de le préciser, ni même de republier les modifications éventuelles. Vous pouvez faire à peu près n'importe quoi du code source ...

3.3.4 Sources complémentaires

- <http://www.levenez.com/unix>
- http://en.wikipedia.org/wiki/Berkeley_Software_Distribution
- <http://www.kurzweg.info>
- <http://zero202.free.fr>
- <http://linux-attitude.fr/post/philosophie-unix>

4 Rappels de programmation en Langage C

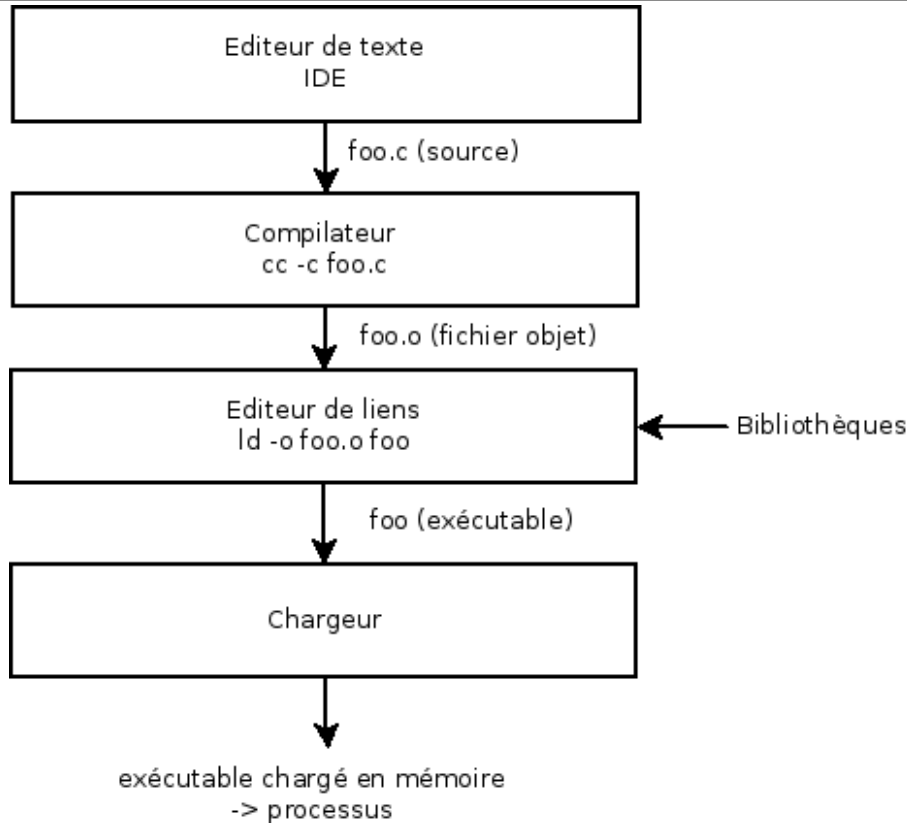
4.1 Chaîne de production des exécutables

La production de programmes exécutables sur un système d'exploitation passe par plusieurs étapes, et permet de transformer du code écrit dans un langage de haut niveau vers un programme écrit en code machine :

1. Saisie du programme dans le langage et enregistrement sur le disque : c'est le *fichier source*
2. Compilation avec le compilateur du langage (vérification syntaxique et lexicale) : on obtient le *fichier objet*
3. Edition de liens (résolution des références externes) : c'est le *fichier exécutable*

Lorsque l'utilisateur demande l'exécution de son programme, le fichier exécutable est alors monté en mémoire centrale : c'est l'étape de chargement. Le système alloue de la place mémoire pour placer le code et les données du programme. Nous précisons dans la partie suivante du cours cette dernière étape.

FIG. 6 Chaîne de production d'un exécutable



4.2 Illustration

Ces diverses manipulations sont inspirées des exercices donnés par Ivan Boule du CNAM, pour l'enseignement de la SMB137.

Soit le programme suivant écrit en langage C :

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 int TAILLE_ALLOC = 1024*1024*1024;
5 // =====A=====
6 int calcul_trigo( int n)
7 {
8     int i, res = 0;
9     for(i=0; i<=n; i++)
10         res+=3*sin(i);
11     return res;
12 }
13
14 // =====B=====
15 int calcul_carre( int n)
16 {
17     int i, res = 0;
18     for(i=0; i<=n; i++)
19         res+=i*i;
20     return res;
21 }
  
```

```

22
23 // =====C=====
24 int main(int argn, char *argv[], char *env[])
25 {
26     int nb_pas;
27     int i;
28     char *zone;
29
30     for (i=0; env[i] != NULL; i++)
31         printf("VARIABLE ENVIRONNEMENT:%s\n", env[i]);
32
33     zone = (char *) malloc(TAILLE_ALLOC);
34
35     if (argn != 2){
36         printf("il faut 1 argument : le nb de pas de calcul\n");
37         exit (-2);
38     }
39
40     sscanf(argv[1], "%d", &nb_pas);
41
42     printf("\n\n\nnombre de pas = %d \n", nb_pas);
43     printf("somme trigo = %d\n", calcul_trigo(nb_pas));
44     printf("somme carre = %d\n", calcul_carre(nb_pas));
45
46     printf("Adresse de     main         = %09lx\n", main);
47     printf("Adresse de     TAILLE_ALLOC = %09lx\n", &TAILLE_ALLOC);
48     printf("Adresse de     nb_pas        = %09lx\n", &nb_pas);
49     printf("Adresse de     zone          = %09lx\n", zone);
50     printf("Adresse de     argv[1]       = %09lx\n", argv[1]);
51
52     // =====D=====
53     sleep(5);
54     exit(nb_pas);
55 }

```

Ce programme contient quatre sections :

- A. Les directives d'inclusion qui permettent d'utiliser des codes existants, contenus dans d'autres fichiers.
- B. Les variables globales, visibles dans l'ensemble du programme
- C. Les variables locales, visibles dans le corps du programme où elles sont définies
- D. Les fonctions, unités d'exécution dans le langage C.

4.2.1 Compilation

La compilation de ce programme est illustrée par la séquence de commandes suivante :

```

ikare@kaitan:exemples > ls -l
total 2
-rw-r--r--  1 ikare  bsd  1393 Feb 18 11:31 exo1.c ❶
ikare@kaitan:exemples > gcc -c exo1.c ❷
ikare@kaitan:exemples > ls -l
total 6
-rw-r--r--  1 ikare  bsd  1393 Feb 18 11:31 exo1.c
-rw-r--r--  1 ikare  bsd  2088 Feb 22 15:23 exo1.o❸

```

1. Le fichier source
2. La compilation avec le compilateur gcc
3. Le fichier objet créé

Le fichier ainsi créé se compose des parties suivantes

- Text : ensemble du code, instructions machines.

- Data : données du programme.
- Rodata : données en lecture seule (Read Only).
- Symbol : table des symboles.
- Comment : informations sur le compilateur

Chaque fichier objet (ou exécutable) possède une table décrivant les différents symboles qu'il contient. La commande `nm [-options] [ref] ...` permet de visualiser la table des symboles, en fournissant : son nom, sa valeur, sa classe et son type.

```
ikare@kaitan:exemples > nm exo1.o
00000000 D TAILLE_ALLOC ❶
00000060 T calcul_carre ❷
00000000 T calcul_trigo
          U exit
000000a0 T main
          U malloc ❸
          U printf
          U puts
          U sin
          U sleep
          U sscanf
```

1. Le symbole est une variable
2. Le symbole est défini dans le code
3. Le symbole est indéfini

Certains symboles ne sont donc pas "résolus". Il s'agit dans notre cas de fonctions définies dans d'autres bibliothèques.

4.2.2 L'édition de liens

L'édition de lien permet de constituer un fichier exécutable à partir de bibliothèques et de modules objets compilés séparément. L'éditeur de lien doit donc résoudre pour chacun des modules les références externes non résolues, en extrayant de bibliothèques les modules correspondant à des fonctions effectivement utilisées.

Historiquement l'édition de lien se faisait de manière *statique* : chaque code d'une fonction utilisée est recopié dans le fichier binaire. L'exécutable est alors autonome, mais si plusieurs programmes utilisent la même fonction, cette portion de code sera alors autant de fois chargée en mémoire centrale. De plus, la taille de l'exécutable augmente en conséquence. Le changement de version de bibliothèque ne se fera également pas sans souci.

Dans le cas de *l'édition de liens dynamique*, les références à résoudre vont contenir des informations sur les objets partagés (fichiers `.so` sous BSD), qui seront chargés *si nécessaire* à l'exécution du programme.

```
ikare@kaitan:exemples > gcc -static exo1.c -lm -o static ❶
ikare@kaitan:exemples > gcc exo1.c -lm -o dynamic ❷
ikare@kaitan:exemples > ls -lh
total 300
-rwxr-xr-x  1 ikare  bsd      5.9K Feb 22 16:40 dynamic ❸
-rw-r--r--  1 ikare  bsd      1.4K Feb 18 11:31 exo1.c
-rw-r--r--  1 ikare  bsd      2.0K Feb 22 15:23 exo1.o
-rwxr-xr-x  1 ikare  bsd     265K Feb 22 16:40 static ❹
ikare@kaitan:exemples > nm dynamic
08049978 D TAILLE_ALLOC
08049980 D __DYNAMIC
08049a4c D __GLOBAL_OFFSET_TABLE__
          w __Jv_RegisterClasses
08049a3c d __CTOR_END__
08049a38 d __CTOR_LIST__
08049a44 d __DTOR_END__
08049a40 d __DTOR_LIST__
0804997c r __FRAME_END__
08049a48 d __JCR_END__
```

```

08049a48 d __JCR_LIST__
08049a7c A __bss_start
08048730 t __do_global_ctors_aux
080484c0 t __do_global_dtors_aux
08049970 D __dso_handle
0804996c D __progname
08049a7c A _edata
08049a84 A _end
0804875c T _fini
08048374 T _init
      U _init_tls@@FBSD_1.0
08048430 T _start
0804810c r abitag
      U atexit@@FBSD_1.0
08048580 T calcul_carre
08048520 T calcul_trigo
08049a7c b completed.4970
08049a80 B environ
      U exit@@FBSD_1.0
080484f0 t frame_dummy
080485c0 T main
      U malloc@@FBSD_1.0
08049974 d p.4968
      U printf@@FBSD_1.0
      U puts@@FBSD_1.0
      U sin@@FBSD_1.0
      U sleep@@FBSD_1.0
      U sscanf@@FBSD_1.0

```

1. Edition de lien en statique
2. Edition de lien en dynamique
3. Fichier exécutable (x) de taille 5.6Ko
4. Fichier exécutable (x) de taille 256Ko
5. Le lien dynamique

4.2.3 Exécution

Le fichier exécutable est maintenant créé. Pour le lancer il suffit sous *Unix* de faire la commande `./dynamic 4`, ce qui donne le résultat suivant :

```

VARIABLE ENVIRONNEMENT:TERM=rxvt
VARIABLE ENVIRONNEMENT:COLORTERM=rxvt-xpm
VARIABLE ENVIRONNEMENT:WINDOWID=44040196
VARIABLE ENVIRONNEMENT:DISPLAY=:0.0
VARIABLE ENVIRONNEMENT:COLORFGBG=15;0
VARIABLE ENVIRONNEMENT:SSH_AGENT_PID=1474
VARIABLE ENVIRONNEMENT:SSH_AUTH_SOCK=/tmp/ssh-3Vpzd7LUzB/agent.1457
VARIABLE ENVIRONNEMENT:USER=ikare
VARIABLE ENVIRONNEMENT:MACHTYPE=i386
VARIABLE ENVIRONNEMENT:MAIL=/var/mail/ikare
VARIABLE ENVIRONNEMENT:VENDOR=intel
VARIABLE ENVIRONNEMENT:SHLVL=3
VARIABLE ENVIRONNEMENT:HOME=/home/ikare
VARIABLE ENVIRONNEMENT:TEXEDIT=emacs
VARIABLE ENVIRONNEMENT:PAGER=more
VARIABLE ENVIRONNEMENT:LC_CTYPE=fr_FR.ISO8859-15
VARIABLE ENVIRONNEMENT:SSH_ASKPASS=/usr/local/bin/ssh-askpass
VARIABLE ENVIRONNEMENT:GROUP=bsd
VARIABLE ENVIRONNEMENT:MM_CHARSET=ISO-8859-15
VARIABLE ENVIRONNEMENT:LOGNAME=ikare
VARIABLE ENVIRONNEMENT:MOZILLA_HOME=/usr/local/bin/firefox3

```

```
VARIABLE ENVIRONNEMENT:RSYNC_RSH=/usr/bin/ssh
VARIABLE ENVIRONNEMENT:PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/games:/usr/local/ ↔
  sbin:/usr/local/bin:/home/ikare/bin
VARIABLE ENVIRONNEMENT:LANG=fr_FR.ISO-8859-15
VARIABLE ENVIRONNEMENT:SHELL=/bin/tcsh
VARIABLE ENVIRONNEMENT:HOST=kaitan.fremens
VARIABLE ENVIRONNEMENT:CPUTYPE=k7
VARIABLE ENVIRONNEMENT:OSTYPE=FreeBSD
VARIABLE ENVIRONNEMENT:PWD=/home/ikare/data/travail/Cours/Cnam/SMB137/Cours/ ↔
  Chapitre1/exemples
VARIABLE ENVIRONNEMENT:LC_ALL=en_US.ISO8859-1
VARIABLE ENVIRONNEMENT:HOSTTYPE=FreeBSD
VARIABLE ENVIRONNEMENT:EDITOR=vi
```

```
nombre de pas = 4
somme trigo = 1
somme carre = 30
Adresse de      main          = 0080485c0
Adresse de      TAILLE_ALLOC = 008049978
Adresse de      nb_pas       = 0bfbfe938
Adresse de      zone         = 028300000
Adresse de      argv[1]      = 0bfbfebf6
```

Le programme a simplement affiché les variables d'environnement de son contexte, et exécuté les deux fonctions. Pour son exécution, le système a chargé son code en mémoire, et exécuté les instructions sur le processeur, en mettant à jour les différents registres et le compteur ordinal.

4.3 Exercices de révision

Quelques petits exercices pour revoir les commandes *shells* et un petit peu de programmation. Vous pouvez consulter les cours disponibles sur www.kurzweg.info.

1. Soit un fichier `fic` contenant 12 lignes. Donnez la commande composée permettant d'afficher les lignes de 5 à 9.
2. Utilisez la commande `find` pour supprimer dans votre répertoire de domiciliation (`$HOME`) tous les fichiers dont le nom est `core`.
3. Ecrivez en C un programme affichant la liste des nombres entiers premiers (divisibles uniquement par 1 et par eux-mêmes).

5 Références

5.1 Références

- [1] Paolo ZanellaYves Ligier, *Architecture et technologie des ordinateurs*, Dunod, isdn : ISBN 2 10 003801 X, 199,2002.
- [2] Matt WelshKalle DalheimerLar Kaufman, *Le systeme Linux*, O'Reilly, 1995, 1997, 1999, 2000.
- [3] Joelle Delacroix, *Linux*, Dunod, 2003.
- [4] Jean-Marie Rifflet, *La programmation sous Unix*, EdiScience, 1986, 1989, 1993, 2002.
- [5] Michael W. Lucas, *Absolute FreeBSD: The Complete Guide to FreeBSD*, No Starch Press, 2007.
- [6] Michael W. Lucas, *FreeBSD 7.0*, No Starch Press, 2007.
- [7] Nicholas P. Carter, *Architecture de l'ordinateur*, EdiScience - Schaum's, 2002.
- [8] J. Archer Harris, *Systèmes d'exploitation*, EdiScience - Schaum's, 2002.
- [9] Eric Steven Raymond, *The art of Unix Programming*, Thyrsus Enterprises, 2003.