Sources des systèmes d'exploitation et programmation système

Patrick Cégielski
Université Paris XII

Résumé : Le programmeur système n'a pas besoin en théorie d'étudier les sources du système d'exploitation pour lequel il conçoit des applications, et heureusement. Une bonne connaissance des appels système est suffisante. Nous nous proposons de montrer en quoi l'étude des sources peut être profitable.

Modèles en couches

Les informaticiens ont pris l'habitude de représenter les objets complexes par une **architecture en couches,** rendue célèbre par le modèle OSI de l'ISO concernant les réseaux informatiques et présentant sept couches. Le principe en est le suivant :

- Les couches sont numérotées de 1 à N, la couche 1 étant toujours la plus proche des contingences matérielles et la couche N la plus proche de l'intervenant humain.
- La règle fondamentale est que la couche de rang i ne peut communiquer qu'avec les couches i-1 et i+1. Les règles qui régissent les échanges entre une couche donnée et la couche immédiatement inférieure constituent une **interface**.

Rappelons au passage que le fameux modèle OSI à sept couches n'a jamais été réellement implémenté, relégué aux oubliettes par un modèle plus pratique à quatre couches : l'architecture TCP/IP.

Les couches d'un système d'exploitation

On peut distinguer trois couches principales en ce qui concerne l'approche d'un ordinateur par un utilisateur, ce qui se fait toujours à travers son système d'exploitation :

- La **couche utilisation** permet à l'*utilisateur final* (*utilisateur lambda* disent certains) de charger les logiciels (ou *applications*) dont il a besoin et de manipuler quelque peu les fichiers ;
- La **couche programmation système** a pour but de concevoir les applications dont on vient de parler sur un système informatique donné constitué d'un ordinateur et du noyau d'un système d'exploitation ;
- La **couche conception du système** concerne la mise au point du noyau et de l'implémentation de l'interface avec la couche précédente.

Voyons maintenant ce qu'il en est des interfaces :

- L'interface entre l'utilisateur final et la couche système s'effectue grâce à l'*interpréteur de commandes*, celui-ci pouvant être soit sous forme console (avec ses commandes telles que *copy*, *rename*...) ou comme interface graphique (**GUI** pour *Graphical User Interface*) ;
- L'interface entre la couche système et la couche conception s'effectue à travers un langage de programmation, le langage C étant devenu le langage de référence absolu, augmenté par des **API** (*Application Programmer Interface*), constituées d'*appels système*.

Remarquons qu'une application à proprement parlé est indépendante d'un système informatique donné : le même traitement de texte *Word* de Microsoft se décline en version pour Windows (du même Microsoft) mais également pour MacOS d'Apple ; de même pour le système de gestion des bases de données Oracle ou d'autres grandes applications. Par contre le cycle de conception de l'application devra tenir compte des couches indiquées ci-dessus, la diversité des systèmes d'exploitation présentant un grave inconvénient pour les concepteurs d'applications.

Notons, pour être complet, qu'il existe en fait deux autres couches, accessoires pour notre propos, mais évidemment non dénuées d'importance par ailleurs :

- La ***couche administration*** permet de paramétrer le système et à le tenir à jour ; elle est indispensable pour les systèmes d'exploitation qui permettent plusieurs utilisateurs ;
- La **couche écriture de scripts** permet d'automatiser certaines séquences répétitives de commandes (en mode console uniquement).

Il existe, à l'heure actuelle, trois grands systèmes d'exploitation. Tout d'abord Unix, devenu en fait le nom déposé d'un noyau parmi d'autres : BSD, Linux, Solaris et autres. Ensuite Windows, qui ne concerne que les micro-ordinateurs de type PC, mais ce sont de loin les plus nombreux. Enfin MacOs d'Apple. Depuis sa version X, MacOS est en fait un Unix avec un système de gestionnaire de fenêtres original (comparable à KDE ou à Gnome pour Linux). De même Windows va certainement passer sous un noyau Unix dans une de ses prochaines versions, tout en conservant son système de gestionnaire de fenêtres original, mais la date n'en est pas encore annoncée.

Cycle de conception d'une application

Le cycle de conception d'une application comprend l'élaboration d'un cahier des charges (70% du temps total, rappelons-le), mise au point de l'algorithme, codage (on dit aussi programmation) d'un prototype, traitement des exceptions, codage final (de l'application consolidée), écriture de la documentation, localisation éventuelle (pour différents pays et différents publics) et enfin maintenance.

La programmation système concerne les trois points liés au codage, à savoir la programmation d'un prototype, le traitement des exceptions et le codage final.

Prototypage

La programmation système devrait toujours commencer par l'élaboration d'un **prototype**. Le langage le plus utilisé à l'heure actuelle pour le prototypage est certainement Java. Notons au passage que ce langage de programmation ne devrait d'ailleurs n'être utilisé que pour cela, à cause de la lenteur de l'exécution des programmes fournis (cette lenteur n'est pas intrinsèque, bien entendu, mais due à la philosophie Java qui invite à

éviter les *compilateurs natifs*, qui traduiraient le programme directement en langage machine et non dans un langage intermédiaire qu'il faut ensuite interpréter, le ByteCode dans le cas de Java). Curieusement cependant, à la grande surprise générale, il existe de nombreuses applications finales écrites en Java : soit la lenteur n'est pas véritablement un problème, soit l'effort supplémentaire pour porter l'application dans un langage plus approprié est trop important. Une fois un tel prototype terminé, on le présente au client et on l'améliore.

Remarquons que le prototypage est malheureusement souvent remplacé par une application écrite en langage RAD (*Rapid Application Development*). Pendant longtemps, Microsoft a préconisé d'utiliser *Visual Basic* à cet effet. L'inconvénient est qu'il est difficile de passer d'une application RAD à l'application consolidée. Microsoft a changé d'avis et propose maintenant sa technologie .NET qui permet d'écrire une application pour Windows en C#, une extension du langage C proche de Java, mais également dans une nouvelle version de Visual Basic que l'on peut traduire dans le même langage intermédiaire que C#.

Traitement des exceptions

Pour concevoir une application, on doit d'abord se concentrer, et uniquement se concentrer, sur le but principal de celle-ci, en ne tenant pas compte des erreurs éventuelles qui pourraient être dues à des données erronées fournies par l'utilisateur final. Par contre, une fois le cœur de l'application mise au point, on doit revenir sur celle-ci et essayer de tenir compte le plus possible des erreurs qui pourraient intervenir : c'est le **traitement des exceptions.**

Rappelons qu'on ne peut pas traiter toutes les erreurs possibles (il existe un théorème mathématique à ce propos) et qu'on appelle *exceptions* les erreurs prises en charge.

Un langage tel que Java permet de bien distinguer syntaxiquement, par son jeu de « *try – catch* » le traitement des exceptions du cœur du programme, mais là encore au prix d'un certain ralentissement. Rien de tel n'existe en langage C et cela peut rendre les grands projets d'une lecture particulièrement ardue. Dès sa toute première version de Linux, la 0.01, Linus Torvalds consacre à peu près la moitié du code au traitement des exceptions, qui plus est sans le dire explicitement. C'est une des raisons qui rend la lecture du source de Linux, pourtant d'une clarté rarement vue ailleurs, rébarbative au premier abord.

Application consolidée

Une fois le prototype considéré comme acceptable (l'expérience montre qu'on obtiendra rarement un prototype parfait), on écrit l'**application consolidée** en langage natif pour le système d'exploitation. Le plus souvent il s'agit du langage C augmenté des API fournies par les concepteurs du système d'exploitation. L'intérêt d'un langage de prototypage tel que Java est qu'il est indépendant de la plate-forme considérée. Ce n'est plus vrai en général pour l'application consolidée.

À la recherche de documentation

La programmation système s'effectuant en langage C augmenté des API, il va falloir se documenter sur le langage C, sur le compilateur C utilisé et sur les dites API.

Pour le noyau du langage C, il n'y a pas de problème puisqu'il n'en existe que trois versions : le C Kernigham et Ritchie d'origine (1978), le C standard (ou ANSI, 1988) et enfin le C ISO (1999).

Le compilateur, ou l'environnement de développement intégré (IDE), fait en général l'objet d'une bonne documentation de la part de ses concepteurs. On peut juste regretter qu'il varie d'une version à l'autre.

Pour les API, c'est en général un peu plus difficile. On commence en général par étudier de ci, de là des applications dont on arrive à se procurer les sources. Les universités (et, exception culturelle oblige, en France les Écoles) sont les détenteurs de sources, mâchées et remâchées.

L'apprentissage par l'exemple, c'est bien mais cela devient très vite insuffisant. On va donc se précipiter vers de gros livres qui ont l'air de tout renfermer. Tout un chacun connaît les Petzold pour la description de l'API des différentes versions des Windows, en particulier de Win32 pour Windows 95, NT, 98, 2000, Me et les suivants. Pour Linux, il existe d'excellents ouvrages, y compris en français, comme le livre de Christophe Blaess. Il existe également des livres sur des aspects particuliers, par exemple celui de Gray sur la programmation réseau. Il existe également la documentation Unix en son fameux manuel en ligne ***man*** et Linux sous la forme des HOWTO ou d'un projet de documentation plus globale, non encore finalisé.

Il s'agit d'introduction à un niveau plus élevé mais, curieusement, il n'existe pas de documentation complète. Rappelons que la documentation officielle de la part de Microsoft sur l'API Windows était si incomplète au début des années 1990 que le livre d'Andrew Schulman sur Windows non documenté connut un énorme succès.

L'arbitre final : les sources

C'est devant cet état de fait que l'étude intelligentes des sources du système d'exploitation devient indispensable :

- Les sources contiennent tout d'abord des commentaires qui éclairent quelquefois l'utilisation de tel ou tel appel système.
- Une étude plus approfondie des sources montre ensuite le rôle exact de ces appels système. Le livre de Card, Dumas et Mevel, par exemple, commente les appels système en les mettant en regard avec l'implémentation, sans entrer dans le détail de celle-ci.
- Bien entendu, les sources apparaissent au premier abord comme extrêmement techniques et lourdes. Une étude complète de celles-ci cependant ne peut que nous laisser coi sur ce qu'il faudrait programmer soi-même s'il n'y avait pas de système d'exploitation et cela les rend très vite beaucoup plus sympathiques.

Cela ne doit pas nous inciter à court-circuiter les appels système. Nous avons vu qu'ils constituent l'interface entre la couche conception et la couche programmation système. :

- Un système d'exploitation tel que Linux prend ses précautions : on doit recompiler tout le noyau si on essaie de court-circuiter. Imaginer la tête du client à qui vous devriez annoncer : l'application fonctionnera après recompilation de tous les noyaux sur chaque poste de travail.
- Cette interface est, dans sa plus grande partie, standardisée grâce à Posix. Même Windows, dont on ne considère pas que le noyau actuel soit un Unix, se veut compatible avec les standards Posix.

- Il suffit de regarder les changements dans la définition du descripteur de processus *task_struct* de Linux pour s'apercevoir que votre application ne connaîtra un cycle de vie que de quelques mois.
- Rappelons ce qui s'est passé avec MS-DOS. Les programmeurs système essayaient de trouver des astuces, utilisant des emplacements réservés (annoncés comme tels très longtemps à l'avance). Dès sa version suivante, l'utilisation de l'application faisait planter tout le système. Notons qu'on retrouve encore ce phénomène pour Windows : Microsoft a même dû donner des certifications pour les applications lors du passage à la version XP.

Comment se procurer ces sources ?

J'espère avoir éveillé suffisamment la curiosité du lecteur pour l'inciter à consulter les sources du système d'exploitation sur lequel il a à concevoir des applications, ne serait-ce que grossièrement. La première étape consiste alors à se procurer ces sources. Ceci n'est pas toujours évident car, jusqu'il y a peu, les concepteurs de systèmes d'exploitation propriétaires ne les diffusaient pas, espérant ainsi sauvegarder leurs droits et éviter d'être pillés. Jusqu'aux années 1980, chaque constructeur de matériel avait sa ligne de produit, et le client était dans une très large mesure dépendant de ce constructeur pour tout ce qui concernait le logiciel et le matériel. Certains constructeurs profitaient de cette situation pour pratiquer des tarifs prohibitifs ou pour imposer certaines configurations à leurs clients. Le ressentiment des clients prit une telle ampleur que ces derniers finirent par imposer leurs souhaits. IBM commença par diffuser le code du BIOS de ses micro-ordinateurs en 1981, ce qui leur permit de devenir prédominants (mais IBM ne fut plus le seul constructeur). DEC (*Digital Equipment Corporation*) passa d'un système d'exploitation propriétaire, VMS, sur ses mini-ordinateurs à un système d'exploitation ouvert de type Unix. Ses clients lui en furent gré et l'entreprise vendit plus de machines. Microsoft essaie de faire passer sa technologie .NET comme un standard ECMA (*European Computer Manufacturer Association*), sans totalement réussir pour l'instant (mi 2004).

Les sources du tout premier noyau Unix furent mises à la disposition des universités qui le demandaient. Ceci conduisit à la mouture BSD (*Berkeley System Distribution*) dont la version 4.4BSD de 1993 demeure une référence.

Microsoft ne diffuse pas les sources de Windows à l'heure actuelle, y compris pour une somme rondelette pour les entreprises tiers qui conçoivent les applications phare. Le problème majeur de ce système d'exploitation, son instabilité, serait peut-être en passe d'être résolu s'il le faisait mais c'est comme ça. Il est vrai qu'il ne publie pas non plus les sources de MS-DOS pourtant abandonné depuis plusieurs années, mais on pouvait avoir une idée de son organisation interne grâce aux clones, FreeDos par exemple, qui étaient diffusés avec les sources. L'instabilité chronique de Windows et le succès croissant de Linux font que la conception de tels clones ne soit pas suffisamment intéressante pour qu'on s'y intéresse.

Il nous reste les sources de plusieurs Unix, principalement FreeBSD, NetBSD et Linux. Les sources sont accessibles dans toutes les distributions et, de toute façon, disponible sur Internet (faire « Linux Kernel cross-reference » par exemple sous Google pour obtenir de nombreux sites qui les diffusent et qui permettent de naviguer dans différentes versions). Les sources de MacOs X sont en partie publiques puisqu'il s'agit de BSD. Linux reste la référence quant à la diffusion de ses sources : c'est ce qui a fait son succès, à la fois par le nombre de ses contributeurs à travers le monde entier et par sa diffusion puisqu'il est en passe de devenir le premier système d'exploitation utilisé dans le monde et sans propriétaire.

Les sources sont-elles lisibles ?

Il est difficile de le savoir pour les sources qui ne sont pas du tout diffusées, à moins d'en être l'un des acteurs. Tout ce que nous savons c'est que les sources des systèmes d'exploitation propriétaires fournies aux entreprises tiers sont d'une lisibilité analogue à celles des systèmes d'exploitation dits libres.

Penchons-nous sur le cas Linux. Si on prend la dernière version en date, quelle qu'elle soit, on risque d'être très vite submergé car les sources en sont d'une très grande clarté mais d'un accès difficile. Je me permettrais donc deux conseils :

- commencer par étudier le plus ancien noyau puis explorer quelques noyaux intermédiaires qui contiennent des commentaires non nécessairement repris plus tard ;
- se faire aider par les articles et les livres qui paraissent à propos de ces noyaux.

C'est dans ce cadre que j'ai écris un livre sur le tout premier noyau de Linux qui commente de façon structuré l'intégralité du code.

Sources des pilotes de périphériques

Les sources du noyau d'un système d'exploitation sont intéressantes. L'accès aux sources d'un certain nombre de pilotes de périphériques est par contre absolument indispensable. Les constructeurs de périphériques fournissent des pilotes pour les systèmes d'exploitation dominants. Ce fut pendant longtemps Windows et MacOS. La situation a légèrement changé : plusieurs constructeurs commencent à proposer également les pilotes pour Linux et il y eut beaucoup de retard lors de la sortie de Windows XP. Il faut donc se lancer dans la conception de ces pilotes. Le livre de Rubini et Corbet est devenu la référence pour Linux.

Références

Christophe Blaess, **Programmation système en C sous Linux**, Eyrolles, 2000, XX + 932 p.

Rémy Card, Éric Dumas et Frank Mevel, **Programmation Linux 2.0 : API système et fonctionnement du noyau**, Eyrolles, XIII + 520 p., 1998 et rééditions.

Patrick Cégielski, **Conception de systèmes d'exploitation : le cas Linux**, Eyrolles, XIII + 595 p., octobre 2003.

Warren W. Gray, **Linux Socket Programming**, QUE, XV + 526 p., 2000.

Charles Petzold, **Programmation Windows**, Microsoft Press (existe pour Windows 3.11, le livre qui a rendu Petzold célèbre, puis pour Windows 95 [en fait Win32] ; la dernière version avec C# semble moins intéressante)

Alessandro Rubini et Jonathan Corbet, **Pilotes de périphériques sous Linux**, O'Reilly, deuxième édition, 600 p., 2002.

Andrew Schulman, Coauthor and editor, **Undocumented Windows: A Programmer's Guide to Reserved Microsoft Windows API functions**, Addison Wesley, 1992.

Patrick Cégielski

Professeur d'informatique à l'IUT de Fontainebleau de l'université Paris XII, auteur d'un ouvrage sur le noyau Linux.