Systèmes d'exploitation

Stéphane Bortzmeyer
<stephane@bortzmeyer.org>

23 janvier 2008

Un petit programme tout seul

```
import urllib2
import time

results_filename = "test.out"
url = "http://www.pasteur.fr/"
n = 100 # Read only the first n characters
results_file = open(results_filename, 'w')
try:
    web_data = urllib2.urlopen(url).read(n)
except Exception:
    web_data = None
results_file.write("Result for \"%s\" at %s is \"%s\"\n" % \
                   (url, time.strftime("%H:%M:%S",
              time.localtime(time.time())),
                      web_data))
results_file.close()
```

Que fait ce programme ?

1. Il ouvre en écriture un fichier nommé "test.out" (`open()` est une fonction pré-définie en Python),
2. Il récupère une page Web, "http ://www.pasteur.fr/" (le module urllib2 fait partie de la bibliothèque standard),
3. Il stocke les 100 premiers caractères de cette page dans le fichier (`write()` est une méthode des objets Fichier).

Qui évalue ce programme ?

L'interpréteur Python fait une partie du travail (affectation aux variables, gestion de l'exception, ...).

Certaines tâches (open(), le réseau, l'horloge) sont déléguées au **Système d'exploitation**.

Pourquoi un système d'exploitation ?

- Pour aider (pour éviter aux applications de tout réimplémenter, certaines tâches sont faites par le SE et mises à la disposition de toutes),

Pourquoi un système d'exploitation ?

- Pour aider (pour éviter aux applications de tout réimplémenter, certaines tâches sont faites par le SE et mises à la disposition de toutes),
- Pour gêner et contrarier (si c'était uniquement pour aider, le SE pourrait se réduire à une **bibliothèque** d'utilitaires ; mais il faut aussi protéger les applications les unes contre les autres, par exemple en vérifiant les permissions au moment du open()).

Les tâches du système d'exploitation

- Fournir des abstractions de niveau élevé, par exemple l'application doit voir des **fichiers** plutôt que des blocs sur un disque dur. Elle doit voir la mémoire comme un grand tableau, et pas comme un mélange de barettes et d'espace sur le disque dur,
- Garder le temps et l'indiquer aux applications qui en ont besoin.

Les tâches du système d'exploitation, suite

- Gérer le matériel, les accès concurrents (sur un quadri-processeur, un maximum de quatre tâches peuvent être actives en même temps), les pannes et la reprise,
- Protéger les tâches les unes contre les autres (mémoires distinctes, par exemple),
- Protéger les utilisateurs les uns contre les autres (sur un SE multi-utilisateurs), par exemple en faisant respecter les permissions des fichiers,

Les tâches du système d'exploitation, suite

- Gérer le matériel, les accès concurrents (sur un quadri-processeur, un maximum de quatre tâches peuvent être actives en même temps), les pannes et la reprise,
- Protéger les tâches les unes contre les autres (mémoires distinctes, par exemple),
- Protéger les utilisateurs les uns contre les autres (sur un SE multi-utilisateurs), par exemple en faisant respecter les permissions des fichiers,
- etc (il n'y a pas de limites absolues à cette liste, certains SE en font plus que d'autres).

En quoi consiste un système d'exploitation ?

Deux parties (attention, c'est une vision simplifiée) :

- Un ensemble de bibliothèques et de programmes utilitaires,
- Le **noyau** du SE, la partie qui parle au matériel.

En quoi consiste un système d'exploitation ?

Deux parties (attention, c'est une vision simplifiée) :

- Un ensemble de bibliothèques et de programmes utilitaires,
- Le **noyau** du SE, la partie qui parle au matériel.

Et entre les deux ?

On trouve les **appels système**, seul moyen de communication.

Appels système de notre petit programme

```
% strace -f ./test.py |& more
...
open("test.out", O_WRONLY|O_CREAT|O_TRUNC|O_LARGEFILE, 0666) = 3
...
```

(open = ouverture d'un fichier, distinct du open Python)

Ce petit programme fait en tout 2980 appels système...

Beaucoup sont des chargements de bibliothèques Python.

Petite introduction aux modèles en couches

Pour aider à découper le problème, on l'organise souvent en couches successives.

Les couches les plus hautes sont celles qui touchent à l'utilisateur. Les couches les plus basses sont celles qui touchent au matériel.

Le SE et les autres couches nous permettent de monter plus haut (de faire d'avantage de travail en moins de temps).

Exemple de couche supplémentaire

Python, en tant que langage de haut niveau, nous fournit plein de services au dessus du SE, services qu'on n'aura pas à développer.

En programmant en C, le petit programme utilisé comme exemple serait bien plus long.

La partie haute du SE

C'est un ensemble d'utilitaires et de bibliothèques considérées comme indispensables au fonctionnement des applications.

Il y a une part de flou dans cette définition

Exemples :

- Sur MS-Windows, l'interface graphique est considérée comme faisant partie du SE. Pas sur Unix.
- Sur FreeBSD, il existe une claire séparation entre le **système de base** (« la partie haute du SE ») et applications. Pas sur Gentoo ou Debian.
- Certains spécialistes réservent le terme de SE au noyau (la partie basse du SE).

Ce qu'on trouve dans la partie haute

- Sur Unix, des programmes comme ls ou rm,
- Sur Unix, la bibliothèque C standard (libc) qui fait des choses comme la résolution de noms de machine en adresses IP,
- Sur Unix, des démons de base comme cron.

Points communs : sur un PC/Unix, un utilisateur ordinaire, sans privilège, **peut** remplacer complètement cette partie.

La partie haute et l'utilisateur

C'est cette partie haute qui est vue par l'utilisateur, pas le noyau.

Par exemple, pour l'utilisateur ordinaire, il y a une très grosse différence entre MacOS X et FreeBSD, alors que leurs noyaux sont presque les mêmes.

Les bibliothèques

Sur Unix, on les trouve typiquement dans `/lib` et `/usr/lib`.

La plus connue est la libc, dont dépendent presque tous les programmes.

Une bibliothèque peut être **statique** (incluse au programme lors de l'édition de liens) ou **dynamique** (chargée au moment de l'exécution).

Trouver les bibliothèques

Sur Unix, on peut afficher la liste des bibliotèques dynamiques chargées par un programme avec ldd :

```
% ldd /usr/bin/echoping
        libpopt.so.0 => /lib/libpopt.so.0 (0xb7ec9000)
        libm.so.6 => /lib/tls/i686/cmov/libm.so.6 (0xb7ea3000)
        libidn.so.11 => /usr/lib/libidn.so.11 (0xb7e72000)
        libgnutls.so.13 => /usr/lib/libgnutls.so.13 (0xb7e02000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7c25000)
...
```

Ici, ce programme utilise la libc, mais aussi la bibliothèque de cryptographie GNU TLS, la bibliothèque mathématique libm, etc.

Bibliothèques sur Windows

On les nomme DLL pour *Dynamic Linking Library*.

Comme elles sont chargées dynamiquement, le changement de la DLL modifie le comportement d'un programme, même si celui-ci n'a pas été touché.

Internet Explorer, par exemple, n'est qu'un petit programme, presque tout est fait par des DLL (c'est pour cela qu'il est difficile d'avoir deux versions d'IE sur la même machine).

Pourquoi les problèmes récurrents avec les DLL

Les DLL sont une bonne chose. . . sauf si on a modifié une DLL importante sans faire attention (ce que font parfois les scripts d'installation).

Se méfier à chaque nouveau jeu à installer :-)

Le noyau

Le noyau est la partie basse du SE, celle qui est au contact direct du matériel.

Sur un système comme Unix ou MS-Windows, il s'exécute dans un mode particulier du processeur (*Ring-0* sur les Intel), pour assurer la sécurité.

Activités du noyau

- Arbitrer les accès au matériel (une seule tâche - on dit aussi processus - à la fois dans le processeur),
- Présenter une vision abstraite du matériel (l'appel système `write()` va se traduire par des instructions très différentes selon que le disque parle IDE ou SCSI),
- Présenter une version abstraite des données, c'est le **système de fichiers**, qui permet d'utiliser les mêmes applications, qu'on écrive sur le disque SCSI d'un serveur ou sur la carte SD de son appareil photo numérique,

Activités du noyau, suite

- Présenter une version abstraite de la mémoire, la **mémoire virtuelle**,
- Faire respecter les permissions, par exemple l'appel système `kill()` vérifie que le processus qu'on vise appartient au même utilisateur (pas le droit de tuer les processus des autres),
- Faire respecter la séparation des tâches et des utilisateurs (pas le droit d'écrire dans la mémoire des autres, que ce soit par méchanceté ou par accident).

Ces tâches sont typiquement couplées

La gestion de la mémoire virtuelle protège également contre les accès des autres processus.

Mémoire virtuelle

Une des tâches importantes d'un SE moderne est de gérer la **mémoire virtuelle**. Il s'agit de présenter aux programmes une mémoire linéaire (les **adresses** sont des nombres entiers) et de grande taille.

Un effet utile de la mémoire virtuelle est d'isoler complètement la mémoire de deux processus différents.

Et si la mémoire physique est limitée ?

La mise en œuvre de la mémoire virtuelle fait appel à une zone du disque dur, le *swap*.

Le SE déplace les données de la mémoire physique vers le *swap* si nécessaire (mais c'est lent).

```
% free
*                total        used        free      shared     buffers
Mem:          2066364     2014016       52348           0        7284
-/+ buffers/cache:       360624     1705740
Swap:         4586548        1900     4584648
```

Système de fichiers

Le système de fichiers présente une vue ordonnée des données : celles-ci sont dans des abstractions, nommées **fichiers**, ayant un nom, et organisés en **répertoires**.

Le système de fichiers garde également trace des **méta-données** (permissions, date de modification, type des données - sur BeOS).

Le système de fichiers fait respecter les permissions

```
% rm -f /etc/hosts
rm: cannot remove ‘/etc/hosts’: Permission denied
```

Ce n'est pas rm, une application du SE, mais qui tourne en mode utilisateur, qui fait respecter les permissions.

Notion d'utilisateurs

La plupart des SE sont **multi-utilisateurs**. Ils ont une base d'utilisateurs (sur Unix, `/etc/passwd`).

Tout objet (processus, fichier) a un propriétaire.

Le SE empêche de toucher aux objets des autres.

```
% id
uid=1000(stephane) ...

% ps auxww
root      4867  0.0  0.0  12560  2008 ?        Ssl  17:36   0:00

% kill 4867
kill: kill 4867 failed: operation not permitted
```

To be or not to be root

Ou pourquoi il n'y a guère de virus sur Unix

Sur Unix, un utilisateur est particulier, `root`. Il a tous les droits.

L'utilisateur ordinaire ne peut toucher qu'à ses objets ou à ce que root lui autorise.

Même si un virus est contenu dans une application, il n'aura que les permissions de l'utilisateur, il ne pourra pas affecter le système.

Notion de tâche et de processus

Un **processus** est un programme en train de s'exécuter.

On peut l'afficher avec la commande `ps`.

```
% ps
7010 pts/0    00:00:02 emacs
```

Ici, le processus de numéro 7010 exécute le programme emacs (un éditeur).

Le processus a sa propre mémoire virtuelle, il ne peut tout simplement pas voir celle des autres.

Tâche

Le terme de « tâche » est souvent utilisé pour une version légère du processus (qui partage sa mémoire avec d'autres tâches).

Gestion des E/S

Les E/S (**entrées-sorties**) sont une des tâches les plus complexes car le matériel est très varié... et parfois défaillant.

Pilote de périphérique

Le **pilote** est une partie du noyau qui gère un type de périphérique (disque Flash, carte Wifi, carte son, ...).

Sur un PC, le code de tous les pilotes fait la majorité du noyau...et des bogues.

C'est du code de bas niveau, difficile à écrire, d'autant plus que les constructeurs réservent en général leur documentation à Microsoft.

Le matériel

Continuant notre descente, nous arrivons au matériel, qui n'est pas couvert dans cet exposé.

L'aventure d'un write()

1. Le programmeur a écrit `myfile.write(mydata)`. Python l'exécute.
2. L'interpréteur Python, une application ordinaire ne tournant pas en mode noyau, cherche `myfile` et la traduit dans un identificateur compris par le SE. Il appelle l'appel système `write()` (qui est distinct du write Python) avec cet identificateur (**où** écrire) et les données (**quoi** écrire).
3. Le noyau a vérifié les permissions au moment de l'ouverture du fichier (il ne laisse pas les applications le faire).

L'aventure du write(), suite

1. Le noyau se souvient d'où on en était dans le fichier (depuis son ouverture),
2. Le noyau exécute le code du système de fichiers considéré (il y en a plusieurs), ce code traduit le nom de fichier en des emplacements sur le disque dur,

L'aventure du write(), suite encore

1. Le noyau envoie alors des instructions au **pilote** du disque concerné. Il existe plusieurs pilotes, selon le modèle de disque. Avec SCSI, les instructions sont simples (« écris ce bloc de données en 18234 »), avec IDE, bien plus complexe (« déplace ta tête sur le troisième cylindre, puis écris ce bloc de données sur le deuxième secteur du huitième plateau »).
2. Le matériel se met en mouvement et applique les ordres.

L'aventure du write(), fin

1. Le noyau attend une réponse positive du disque et libère alors l'application (ou bien signale une erreur).

Et, pendant ce temps, le noyau a géré les autres processus, la mémoire virtuelle du nôtre, etc. . .