



Cours de système

Qu'est-ce qu'un système d'exploitation ?
Introduction à Linux

Sébastien Paumier



Objectifs du cours

- savoir ce qu'est un système d'exploitation (OS)
- connaître les grandes questions à se poser pour en concevoir un
- connaître le modèle de Linux
- savoir programmer des applications système



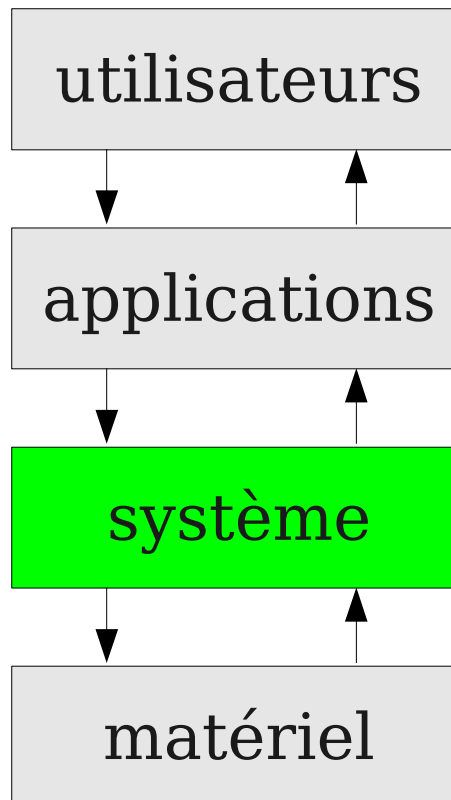
Ressources

- "Systèmes d'exploitation", d'Andrew Tanenbaum
- poly du cours de système de D. Revuz:
 - <http://igm.univ-mlv.fr/~dr>
- exemples du cours et code du premier noyau:
 - <http://igm.univ-mlv.fr/~paumier/systeme/>



Qu'est-ce qu'un OS ?

- interface entre les applications et le matériel physique





Missions

- permettre d'accéder au matériel de façon transparente
 - un programme n'a pas à savoir s'il écrit sur un disque ext3 ou une clé USB fat32
- gérer les ressources (accès physiques, mémoire, CPU)
 - optimiser l'usage de la machine (taux d'occupation du CPU, minimisation des mouvements des têtes de lecture des disques, minimiser le swap, gestion de l'énergie sur les systèmes portables, etc)



Missions

- veiller à la sécurité des applications et des données
- fournir une qualité de service
 - éviter les famines, garantir un accès prioritaire à root, temps de réponse sur un OS temps réel, etc
- être robuste
 - éviter de planter !
 - tolérance à l'erreur (blocs disques défectueux, reboot sauvage, etc)



Les types de système

- mono vs multi-utilisateurs
 - un OS de téléphone peut être mono-utilisateur
- mono vs multi-tâches
 - DOS était monotâche
- mono vs multi-processeurs
- temps réel: garantit un délai maximal d'exécution quelles que soient les conditions



Le multi-utilisateurs

- suppose de pouvoir protéger les données de chacun sur les supports de stockage
- nécessite la notion de droits d'accès
- protège les utilisateurs entre eux
- protège le système:
 - impossible de détruire le système en supprimant par accident une DLL :)



Le multi-tâches

- suppose de pouvoir protéger les processus les uns des autres
- nécessite la notion de protection de la mémoire
- rendu possible par le mode protégé et la mémoire virtuelle paginée
 - seul le noyau peut accéder à tout
 - les processus ne peuvent physiquement pas sortir de leurs pages mémoire



Le noyau

- espace mémoire protégé+ensemble de programmes qui forment la base minimale de l'OS
- tout ce qui n'est pas un appel système fonctionnera dans l'espace utilisateur
- exemple de choix de conception:
l'interface graphique
 - dans le système sous Windows
 - programme utilisateur (serveur X) sous Linux



Types de noyau

- monolithique: tout est dans le noyau (système de fichiers, pilotes, etc)
 - Linux, FreeBSD
- micro-noyau: seulement le strict minimum (ordonnanceur+mémoire virtuelle)
 - Minix, Mac OS X
- hybride (Windows NT)
- exo-noyau: rien n'est protégé



Pourquoi étudier Linux ?

- logiciel libre (ouvert et gratuit)
- très portable (PC, MacBook, smartphones, Xbox, etc)
- interface simple et élégante (les appels systèmes)
- met en œuvre beaucoup de notions intéressantes: processus, droits d'accès, mémoire virtuelle, journalisation, temps réel, modules dynamiques, etc
- formidable mutualisation d'expertise



Un peu d'histoire

- 1969: UNIX (Thompson et Ritchie)
- 1987: Minix (Tanenbaum)
- 1991: premier noyau Linux (Torvalds)
- 1994: v1.0 qui donne le premier système GNU/Linux
- depuis, de nombreuses distributions:
 - Debian, Ubuntu, Fedora, SuSE, Red Hat, Mandriva, Slackware, ArchLinux, Gentoo, etc



Architecture

- noyau monolithique qui gère le matériel, la mémoire, l'ordonnancement des tâches
- séparation en couches (noyau, libs, applications)
- séparation en deux espaces distincts:
 - espace utilisateur (user land)
 - espace noyau (kernel land)
- basée sur un contrôle matériel: le mode protégé du processeur



Un modèle en couches

- la portabilité est assurée par les couches basses
- possible grâce un langage puissant inventé pour écrire UNIX: le C
- un binaire n'est pas portable, mais il suffit de recompiler une application pour la porter sur un OS compatible
- pas besoin de toucher au code!
- concept ancêtre de l'idée de machine virtuelle



Un modèle en couches

- du très bon génie logiciel, car on a des interfaces propres qui permettent de faire abstraction des couches plus basses
- polymorphisme qui facilite la portabilité et l'extensibilité:
 - pour un nouveau processeur, il suffit de changer les codes assembleur bas niveau
 - pour un nouveau système de fichier, il suffit d'implémenter les variantes de **read, write, open...**

fs/



Un modèle en couches

- on obtient des codes plus sûrs, car pour court-circuiter une couche, il faudrait la maîtriser parfaitement
- exemple de bug dû à un mélange entre la couche système et la libc:

```
int main(int argc, char* argv[]) {  
    if (argc != 2) return 1;  
    int fd = open(argv[1], O_CREAT | O_TRUNC | O_WRONLY, 0700);  
    FILE* f = fdopen(fd, "w");  
    close(fd);  
    fprintf(f, "0ooops\n");  
    fclose(f);  
    return 0;  
}
```

bug_FILE.cpp



Modes réel et protégé

- mode réel (par défaut au boot):
 - on peut exécuter n'importe quelle instruction
 - accès total à la mémoire physique
- mode protégé basé sur un contrôle matériel (utilisé dans les OS modernes):
 - protection de la mémoire, ce qui permet la mémoire virtuelle et la pagination
 - commutation de contexte
 - 4 niveaux de privilèges

boot/boot.s



Protection de la mémoire

- idée: interdire à un programme d'accéder à de la mémoire qui ne lui appartient pas, s'il n'a pas le bon niveau de privilège
- en mode protégé, tous les accès mémoire sont contrôlés par le processeur par rapport à une zone autorisée
- en cas de violation, Linux tue le programme fautif avec un signal **SIGSEGV: *segmentation fault***

kernel/traps.c



Commutation de contexte

- changement de contexte:
 - sauver l'état du processus courant (valeurs des registres)
 - restaurer l'état d'un autre processus P
 - reprendre l'exécution de P où elle s'était arrêtée
- permet de donner l'illusion du multi-tâches
- opération critique si trop lente ou trop fréquente

```
kernel/sched.c  
include/linux/sched.h
```



Bascule noyau/utilisateur

- quand un processus utilisateur doit accéder au système physique, pour des raisons de protection, il doit demander au noyau de le faire pour lui
- d'où l'obligation de changer de contexte en copiant des choses entre user land et kernel land
- très coûteux, donc on préfère bufferiser autant que possible
- exemple: copie d'un fichier



Bascule noyau/utilisateur

- version basique, utilisant la bufferisation par défaut de la libc:

```
int main(int argc, char* argv[]) {
    int c;
    while ((c=fgetc(stdin))!=EOF) {
        fputc(c, stdout);
    }
    return 0;
}
```

```
$>man gcc | time ./io_buf > err
```

```
0.01user 0.00system 0:00.57elapsed 2%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+1280outputs (0major+234minor)pagefaults 0swaps
```



Bascule noyau/utilisateur

- version non bufferisée, qui coûte très cher:

```
int main(int argc, char* argv[]) {
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);
    int c;
    while ((c=fgetc(stdin))!=EOF) {
        fputc(c, stdout);
    }
    return 0;
}
```

```
$>man gcc | time ./io_buf > err
0.01user 0.00system 0:00.57elapsed 2%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+1280outputs (0major+234minor)pagefaults 0swaps
```

```
$>man gcc | time ./io_unbuf > err
0.34user 1.62system 0:02.09elapsed 93%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+1280outputs (0major+228minor)pagefaults 0swaps
```



Mode noyau/utilisateur

- grâce au mode protégé, les programmes utilisateur n'ont physiquement pas accès à l'espace du noyau
- mais en mode noyau, on a tous les droits !
- le code du noyau étant critique, on met un maximum de choses en dehors
 - exemple: on peut redémarrer un serveur graphique planté sans devoir rebooter



Problème de maintenance

- si on ne peut pas toucher au noyau, on doit le changer ou le recompiler à chaque mise à jour de n'importe quel morceau!
 - il fût un temps où il fallait recompiler à chaque update d'un driver...
- pour simplifier cela, introduction de codes noyau chargeables dynamiquement: les *modules*



Problème de maintenance

- même philosophie avec les bibliothèques partagées et l'édition de liens dynamiques
- on peut changer un morceau de code sans devoir tout réinstaller ou recompiler, et bien souvent, sans rebooter
 - propriété vitale pour certains systèmes critiques (banques, hopitaux, etc)



Les appels système

- interface simple et élégante pour communiquer de façon homogène
- les diamants:

```
ssize_t read(int fd, void* buf, size_t n);
```

```
ssize_t write(int fd, const void* buf, size_t n);
```

- lire ou écrire des octets vers on ne sait pas quoi (et on ne veut surtout pas savoir!):
 - fichier ordinaire, fichier spécial, terminal, socket, tube, périphérique, etc



Les appels système

- peuvent (presque tous) être appelés comme des fonctions
- peuvent être invoqués par **syscall**, en utilisant les constantes définies dans **sys/syscall.h** de la forme **SYS_read**
- attention: certains appels sont wrappés par la libc
 - ne pas mélanger les appels wrappés et les appels directs avec **syscall** !



Bug

- un processus qui a le même pid que son père ?

```
int main(void) {
    getpid();
    int i=syscall(SYS_fork);
    if (0==i)
        /* Big problem: the son prints the same value
         * for getpid() and getppid() */
        printf("__son : %d    father : %d__\n", getpid(), getppid());
    else
        printf("<<father : %d    son : %d>>\n", getpid(), i);
    return 0;
}
```



Violation du modèle en couches

- explication dans la page man de **getpid**:

NOTES

Depuis la glibc version 2.3.4, la fonction enveloppe de la glibc pour `getpid()` faisait un cache des PID, de façon à éviter des appels système supplémentaires quand un processus appelle `getpid()` de façon répétée. Normalement, ce cache n'est pas visible, mais son fonctionnement correct repose sur la gestion du cache dans les fonctions enveloppes pour `fork(2)`, `vfork(2)` et `clone(2)` : **si une application se passe des enveloppes de la glibc pour ces appels système en appelant `syscall(2)`, alors un appel à `getpid()` dans le fils renverra la mauvaise valeur (pour être précis : il renverra le PID du processus père)**. Consultez également `clone(2)` pour une discussion sur un cas où `getpid()` peut renvoyer une mauvaise valeur quand `clone(2)` est appelé via la fonction enveloppe de la glibc.

- toujours lire les docs en entier!!!