120 Appels système

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Plan

1. Mode noyau (dit privilégié)
2. Appels système
3. Enveloppes
4. Appels de bibliothèque
5. Compatibilité

Mécanismes matériels

Analogie

- « L'État est une communauté humaine qui, dans les limites d'un territoire, revendique avec succès le monopole de la violence physique légitime. » — Max Weber, *Le Savant et le politique* (1919)
- « Le système d'exploitation est une couche logicielle qui, dans les limites d'un ordinateur, revendique avec succès le monopole des mécanismes matériels. » — Analogie facile...

Mode noyau (dit privilégié)

Mode noyau : mécanisme

Objectif

S'assurer que certaines instructions machine sont réservées au système d'exploitation

Problème : le processeur est une machine

- Pour lui, système d'exploitation et processus n'existent pas
- Une instruction machine n'appartient à personne

Solution : deux modes d'exécution

- Un bit de mode dans le registre du mot d'état
- Mode noyau (0) : toutes les instructions sont utilisables
- Mode utilisateur (1) : certaines instructions sont interdites

→ le processeur refuse physiquement d'exécuter l'instruction si le mode n'est pas le bon

Mode noyau : politique

- Au démarrage le CPU est en mode noyau

$\rightarrow$ Le système d'exploitation se charge et configure la machine

- Quand le système démarre des processus, il passe le CPU en mode utilisateur

$\rightarrow$ Les applications sont restreintes sur ce qu'elles peuvent faire

- Quand le CPU revient au système, on repasse au mode noyau

$\rightarrow$ On va y revenir…

Mode noyau : beaucoup de détails

Le monde des CPU est complexe et plein de variété

- La liste des instructions et le mode auquel ils appartiennent est spécifique à chaque processeur
- Plutôt que désactiver l'instruction le mode peut limiter des comportements ou en changer le sens
- Pourquoi se limiter à 2 modes ? Intel en a 4. Certains ARM en ont 7
- On parle parfois d'anneaux de protection (*rings*). Le mode noyau est Ring0
- Les processeurs peuvent offrir d'autres types de modes d'exécution complémentaires au mode noyau

La documentation pour programmeurs des processeurs Intel fait plus de 5000 pages !

Appels système

Problème

Un processus veut faire une opération privilégiée

- Il ne peut pas le faire lui-même

→ Il est en mode utilisateur

- Il ne peut pas changer le mode lui-même

→ Sinon c'est pas un vrai privilège

- Il ne peut pas juste déléguer à une bibliothèque ou faire un `call` à un sous-programme

→ Le mode resterait non-priviliégié

Instruction machine spéciale

Appel système

- Sauvegarde registres (dont CO)
- Passe en mode noyau
- Branche sur du code spécifique du système d'exploitation

→ Le processus ne branche pas où il veut

→ Le processus perd donc le contrôle du CPU

Retour d'appel

- Passe en mode utilisateur
- Restaure les registres (dont CO)

→ Le processus s'est rendu compte de rien

Différence avec call ?

- `call` utilisé pour les sous programmes (processus et noyau)
- `call` prend en argument une adresse
  `syscall` prend un argument un numéro d'appel système

Liste définie d'appels système

- Chaque système d'exploitation est différent
- Plus de 400 sur Linux
- Mais beaucoup sont rarement utilisés

Performance

- `syscall` plus cher que `call` (temps de calcul)
- Coût important du **changement de contexte**

Détails spécifiques

- À chaque système d'exploitation
- Pour chaque architecture

Noms variés

- `syscall`, `int`, `trap`, `swi`, etc.
- Confusion avec d'autres mécanismes (interruption, fautes, etc.)

Nombreux détails

- Qui sauvegarde et restaure les registres ? Comment c'est fait ?
- Où on branche exactement ? Qui décide ?
- Comment on passe les arguments et retourne le résultat ?

Pas d'équivalent portable en C

- Quelqu'un doit les coder en assembleur
- Des enveloppes (*wrapper*) sont fournies

hello_syscall.c

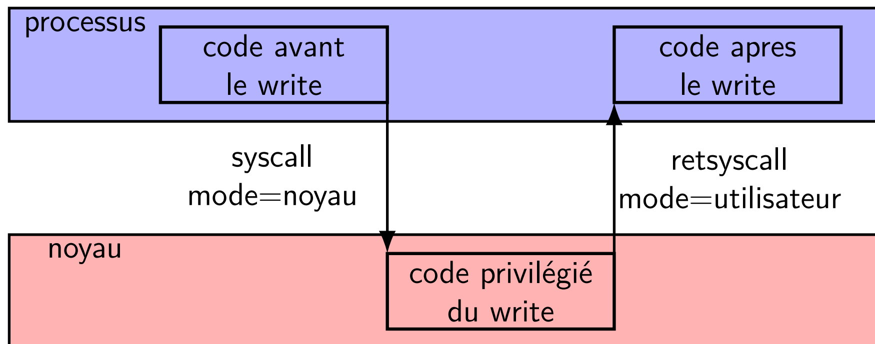
Appel système POSIX `write(2)` « à la main »

```
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>
int main(int argc, char *argv[]) {
  char msg[] = "Hello, World!\n";
  syscall(SYS_write, 1, msg, 14);
  return 0;
}
```

Les arguments de `syscall(2)` sont

- `SYS_write`: le numéro de l'appel système
- 1: le descripteur de la sortie standard
- `msg`: l'adresse du message à écrire
- 14: le nombre d'octets à écrire

Mise en œuvre matérielle



- Le processus s'exécute en mode utilisateur
- L'appel système `write` branche sur le noyau
- Le code du `write` s'exécute en mode noyau
- Le retour au processus perd le mode noyau
- Le processus continue en mode utilisateur

Voir les appels système

Sous Linux `strace`

- Outil de débogage
- Permet de surveiller les appels système Linux
- C'est magique ! (les détails dans INF600C)

```
$ strace ./hello_syscall
[...]
write(1, "Hello, World!\n", 14)         = 14
[...]
```

`strace` sait afficher de façon humaine:

- le nom
- les arguments (au bon format)
- le résultat

Enveloppes

Enveloppe

Problème

Les appels système sont peu portables

- Instructions machines spécifiques aux processeurs
- Choix particuliers des systèmes d'exploitation
- Pas de façon standard de les exprimer en langage C

Solution

Une bibliothèque standard fournit des fonctions spécifiques

- Enveloppe chacun des appels système (*wrapper*)
- Expose API/ABI simples et portables (en C ou C++)
- Connait l'architecture et les choix du système
- Implémenté avec des morceaux d'assembleur (mal nécessaire)

Portabilité interne

Sous Unix

- La `libc` contient les fonctions d'enveloppe
- La section 2 du `man(1)` les documente
- `unistd.h` déclare de nombreux appels système POSIX

RTFM: il peut y avoir des variations entre l'appel système et la fonction C

Sous Windows

- `kernel32.dll` contient les fonctions d'enveloppe
- Par exemple `WriteConsole`
- Les détails techniques ne sont pas documentés :(

Exemple `hello.c`

Pour le programmeur, voilà ce que ça donne

```
#include <unistd.h>
int main(int argc, char *argv[]) {
  char msg[] = "Hello, World!\n";
  write(1, msg, 14);
  return 0;
}
```

- `write(2)` est la fonction système POSIX qui écrit des données
- `ssize_t write(int fd, const void *buf, size_t count);`
- Une vraie fonction C avec une vraie signature

→ Les détails sont laissés à la libc

hello_asm.s

Version assembleur équivalente à `hello.c` (Linux/x86_64)

```
# Programme qui affiche "hello world"
# Compiler avec `gcc hello_asm.s -o hello_asm`

  .globl  main
main:
  # write(1, msg, 14)
  mov     $1, %rax          # appel système write (1)
  mov     $1, %rdi          # sortie standard (1)
  lea     msg(%rip), %rsi   # adresse du message (PIC)
  mov     $14, %rdx         # taille du message (14 octets)
  syscall                   # instruction TRAP

  # return 0
  mov     $0, %rax          # valeur de retour (0)
  ret                       # return
msg:
  .ascii  "Hello, World!\n"
```

Gestion des erreurs

En cas d'erreur

Les enveloppes des appels système:

- Retourne -1
- Positionne `errno(3)`
- La liste de errno est **fixe**: `errno -l`

Le programmeur doit gérer les cas d'erreurs

- Lire la doc (RTFM), section « `ERREURS` »
- Identifier les erreurs possibles
- Les traiter (ou pas)

→ Le traitement des erreurs est une chose **difficile**

  - Recommencer ? Ignorer ? Abandonner ?
  - Afficher un message ? Quel message ?
  - Ne pas réinventer la roue: `perror(3)`, `strerror(3)`

Exercice: lire et comprendre les erreurs de `write(2)`

Appels de bibliothèque

Appels de bibliothèque

Appels système

- Services primitifs
- Spécifiques au système d'exploitation

Bonnes pratiques de génie logiciel

- Utiliser des services généraux
- Portable entre systèmes d'exploitation

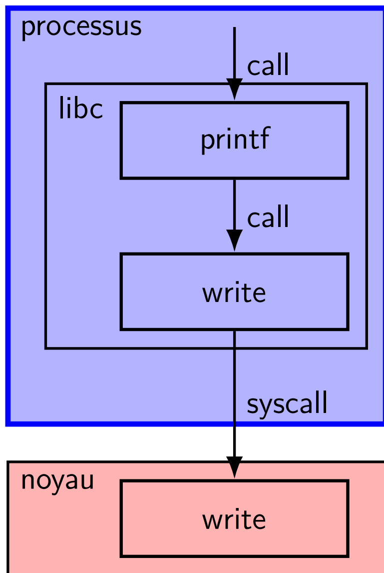
hello_printf.c

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf("Hello, World!\n");
  return 0;
}
```

Bibliothèques standard du C

- <stdio(h>): printf(3), fwrite(3), etc.
- Documentés dans la section 3 du man
- Revoir INF3135 pour les détails

Appels de bibliothèques



Les bibliothèques

- Font partie du processus

→ Aucun privilège particulier

→ Les calls sont normaux*

La libc

Fournit ≈ 1500 fonctions fonctions

- Les fonctions standard C
- Les enveloppes d'appels système

* Il y a une astuce quand les bibliothèques sont dynamiques.

Bibliothèque vs. système

Indépendance chez Linux

- Projet indépendant $\neq$ noyau Linux
- Généralement glibc (de GNU)

Efficacité

- Les appels système coûteux et bas niveau
- Les fonctions de bibliothèques optimisent
  - Caches additionnels
  - Factorisation des appels
  - Traitement direct si possible (sans appel système)
- Les fonctions de bibliothèques généralisent
  - Portables entre différentes versions et systèmes d'exploitation
  - Profitent de nouveaux appels système quand disponibles

Exclusivité (on insiste)

- Le système a l'**exclusivité** des mécanismes matériels
- Les processus sont **isolés** du reste
- Les **appels système** sont leur **seul** moyen d'interagir avec l'extérieur (utilisateurs, périphériques, autres processus)

→ Tout processus qui a besoin d'interagir passera par des appels système

Allégorie de la caverne

Les processus ne voient le monde qu'à travers ce que le noyau décide

Le noyau « ment » souvent :

- Les fichiers de `/proc` n'existent pas vraiment
- C'est pas un disque mais du réseau
- La mémoire n'est pas toujours disponible (sur-réservation)
- *There is no spoon!*

→ Mais ça permet beaucoup de choses!

Dans le cadre du cours

On utilisera le plus possible les appels système

- L'objectif c'est d'être le plus proche du noyau
- Et d'apprendre à le maitriser

On traitera (correctement) les cas d'erreur

- La robustesse sera prise en compte dans la notation des TP
- Les autres qualités aussi : exactitude, lisibilité, modularité, etc.

Compatibilité

Rétrocompatibilité

Le noyau Linux a une forte tradition de rétrocompatiblité

> *« WE DO NOT BREAK USERSPACE! »*
>
> – *Linus Torvalds (2012)*

Les appels système sont stables

- Leur interface de programmation (ABI)
- Leur comportement

Ce n'est pas le cas à l'intérieur du noyau

- Les sous-systèmes évoluent constamment
- Ajout de fonctionnalités non compatibles
- Ajout et mise à jour de pilotes de périphériques

Couche de compatibilité

Une **couche de compatibilité** permet

- À des applications d'un système d'exploitation (ex. Windows) de fonctionner sous un autre système d'exploitation (ex. Linux)

→ L'architecture processeur doit être la même

→ C'est différent d'un émulateur

Exemples

- Wine convertit les appels Windows en appels POSIX

→ Proton fork par Valve pour jeux vidéos sans support Linux

- Cygwin Convertit les appels POSIX en appels Windows
- WSL Windows Subsystem for Linux, de Microsoft

Couche de compatibilité

Mise en œuvre (en gros)

- Fournir une bibliothèque de base spéciale
- Se substitue à celle du système (libc.so, kernel32.dll, etc.)
- Traduit les appels système de l'un vers des appels équivalents
- Simule l'environnement attendu de l'application

Limites

En pratique, traduire les appels système est très compliqué

- Tous les appels système ne sont pas traduits à 100%
- Les performances peuvent varier
- L'environnement simulé doit être cohérent

→ Système de fichiers, accès au matériel, communication entre processus, etc.