Introduction au noyau Linux

Objectifs

Comprendre ce qui se cache derrière nos applications.

La difficulté de réalisation

La complexité des optimisations

Introduction

Créé en 1991 par Linus Torvalds

Inspiré du système Unix

Est bien plus utilisé qu’on ne le croit

Présentation

Un kernel est un logiciel gérant toutes les interruptions de la machine et des applications.

Peut-on dire qu’un Linux = Ubuntu ?

Types de kernel

Monolitique

Micro-noyau

Temps-réel

Fonctionnalités

1. Gestion des systèmes de fichiers

1. Gestion de la mémoire

1. Ordonnancement

Linux

Linux est :

1. monolithique modulaire dynamique

1. réentrant

1. préemptif

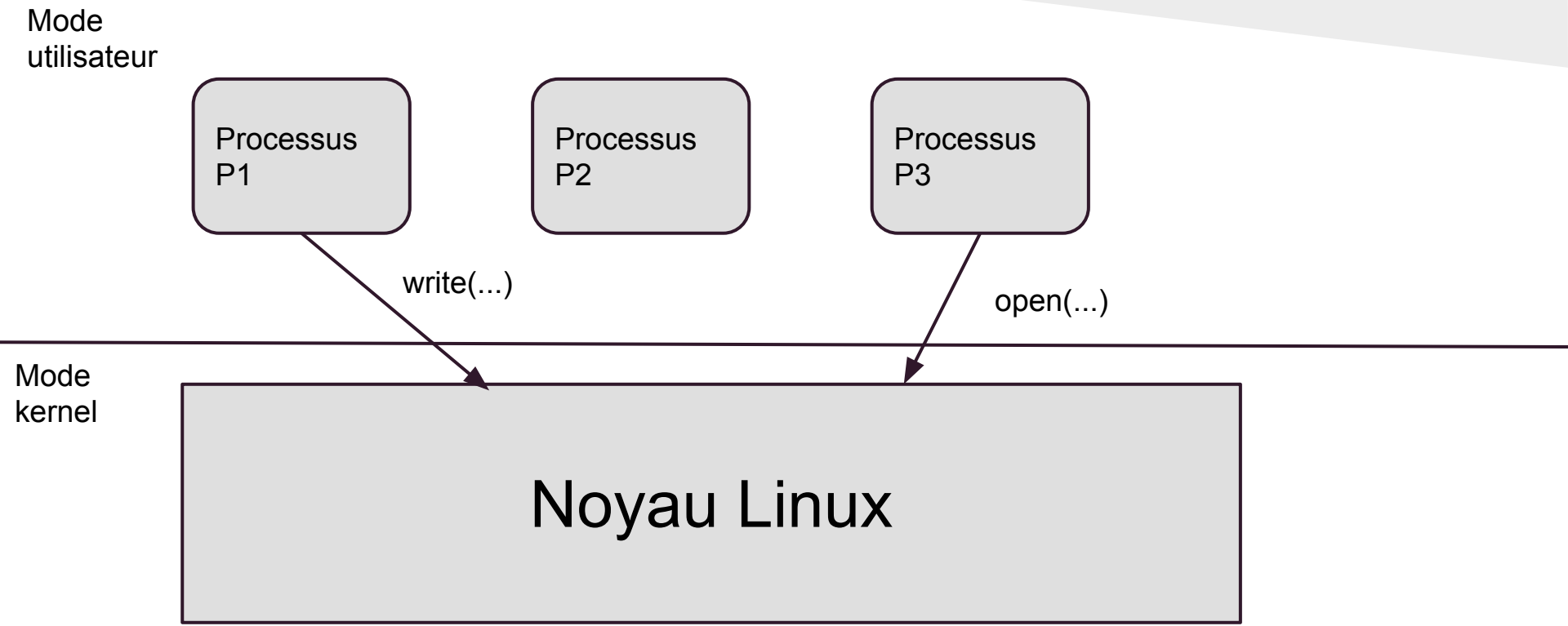
Modes d’exécution

Il existe deux modes dans les OS actuels

Ils permettent de séparer le noyau du code utilisateur

Le moyen de communiquer entre les deux modes se fait par … ?

Modes


Mode
utilisateur
Processus
P1
Processus
P2
Processus
P3
write(...)
open(...)
Mode
kernel
Noyau Linux

Processus

Qu’est-ce qu’un processus ?

Qu’est ce qu’un processus ?

```
struct task_struct {
  volatile long state;
  void *stack;
  int nr_cpus_allowed;
  cpumask_t cpus_allowed;
  struct mm_struct *mm;
  pid_t pid;
  int exit_code;
  // Et plein d’autres champs
}
```

Processus et threads

Quelle est la principale différence entre ces deux notions ?

Processus et threads

```
// current représente le processus appelant et qui s'exécute en mode utilisateur
oldmm = current->mm;
if (!oldmm)
  return 0;

if (clone_flags & CLONE_VM) {
  atomic_inc(&oldmm->mm_users);
  mm = oldmm;
  goto good_mm;
}

mm = dup_mm(tsk);
if (!mm)
  goto fail_nomem;

good_mm:
        tsk->mm = mm;
```

Processus

Tout processus a un parent (sauf init)

Tout processus est créé à partir d'un processus. Le fils devient une copie du père.

Tout processus appartient à un *user* et un *group*

Processus

Tout processus a un père.

Comment le kernel gère la mort d’un père ?

Processus

L'appel à `pid_t fork(void)` créé un processus fils à l'appelant.

Cet appel renvoie 2 valeurs :

→ Le pid du fils, au père

→ 0 au fils.

Ordonnancement

Capacité à élire des processus de manière effective

Doit être très rapide !

Comment déterminer quel processus doit s’ exécuter ?

Ordonnancement

Temps partagé

Chaque processus s'exécute pendant un quantum de temps.

La complexité de l'ordonnanceur Linux est de … ?

Exemple

3 processus P1 / P2 / P3 avec P1 > P2 > P3

P1 s'exécute sur CPU1 et P3 sur CPU2

P2 avait été exécuté sur CPU1 et est prêt.

Sur quel CPU met-on P2 ?

Context switch

Élection d’un CPU d’un processus à un autre

Dans quel cas fait-on un context switch ?

Est-ce valable pour les threads aussi ?

Context switch

Tout le contexte est sauvegardé

Ce contexte inclut :

- L'état du processus (CPU, runtime …)
- Informations (enfants, IPC, privilèges, pid…)
- Registres CPU, PC
- IO informations

Quelques trucs

Peut-on ajouter du code malveillant dans la branche principale ?

Quelques trucs

Vaut-il mieux un micro-noyau ou un noyau monolithique ?

Conclusion

Il reste encore énormément de notions à expliquer (mémoire, fs, multi-coeur, …).

Linux est une référence en terme d’ optimisations, implémentées plus tard dans Windows :)

Développement dans le noyau Linux

Les modules

Les modules sont le moyen le plus simple d’ ajouter du code au noyau.

Peuvent être embarqués dans une distribution Linux

Chargés / Déchargés dynamiquement

Les modules

Ont un point d’entrée et de sortie

Ont accès à certaines fonctions du noyau (via la macro EXPORT_SYMBOL).

Vous n’avez pas accès à la glibc !

Les modules

Toujours tester la valeur de retour d'une méthode !

Une exception dans le kernel peut provoquer un plantage du kernel …

Code source

arch contient tout ce qui est propre à une architecture

include contient les headers

kernel contient les sources

Fonctions standards

printf → printk

malloc → kmalloc

free → kfree

atoi -> kstrtol

Mon premier module

```
#include <linux/module.h>   /* Needed by all modules */
#include <linux/kernel.h>   /* Needed for KERN_INFO */
#include <linux/init.h>     /* Needed for the macros */


int __init init_module(void)
{
      printk(KERN_INFO "Hello world\n");
      return 0;
}

void __exit cleanup_module(void)
{
      printk(KERN_INFO "Goodbye world\n");
}

module_init(init_module);
module_exit(cleanup_module);
```

Makefile

```
obj-m += monmodule.o

all:
    make -C /lib/modules/$(shell uname -r)/build
M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build
M=$(PWD) clean
```

Exécution

Ajouter le module : sudo insmod module.ko

Supprimer le module : sudo rmmod module

Lister les modules : sudo lsmod

Les logs sont écrits dans /var/log/kern.log

Ajouter des métadonnées

Certaines macros permettent de définir les informations de base d'un module

```
#define DRIVER_AUTHOR "Olivier Pitton"
#define DRIVER_DESC   "A sample driver"

MODULE_LICENSE("GPL");
MODULE_AUTHOR(DRIVER_AUTHOR); /* Who wrote this module? */
MODULE_DESCRIPTION(DRIVER_DESC);   /* What does this module do */
```

Ajouter des paramètres

Au lancement du module, nous pouvons spécifier des paramètres similaires à une ligne de commande

```
static int myint = 0;
module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(myint, "An integer");
```

Processus

Les API processus se situent dans linux/sched.h

Les API d’exit sont dans kernel/exit.c

/proc

Système de gestion de fichiers entièrement en mémoire situé dans /proc

Contient toutes les informations relatives au système pendant son exécution (processus, …)

/proc

Les fonctions se situent dans linux/proc_fs.h

Créer un fichier à la racine de /proc :

```
proc_create("salut", 0777, NULL, &proc_fops);
```

Supprimer un fichier à la racine /proc :

```
proc_entry("salut”);
```

Appel système et paramètres

Il faut copier les paramètres du mode utilisateur au mode système et vice versa

```
#include <asm/uaccess.h>

// Copie dans la mémoire kernel ce qu'à envoyer un utilisateur
copy_from_user(user_message,buf,count);

// Copie dans l'espace utilisateur le contenu de la mémoire kernel
copy_to_user(user_message,buf,count);
```

Récupérer un processus

Depuis un module :

```
int mypid = ...;
struct task_struct* process;
process = pid_task(find_vpid(pid), PIDTYPE_PID);
```

Travailler dans le kernel

Au vu de la quantité des sources, le plus simple est d'utiliser : http://lxr.free-electrons.com/

Ce site vous permet de parcourir facilement les sources du noyau

Conclusion

Faites attention à ce que vous faites, ou vous devrez redémarrer votre VM !

Le code présenté est la partie simple du kernel

Avoir accès aux API noyaux permettent de faire plein de trucs funs.