Kernel Development

Partie I – Fondations

Introduction au développement kernel

Bonjour et bienvenue développeurs !
Dans ce cours, nous nous efforcerons de vous apprendre les bases du développement Kernel, les différentes étapes à suivre.
Pour suivre ce cours convenablement, vous aurez besoin d'être familier avec le développement en général, de solides bases en langage C, ainsi que quelques connaissances en Assembleur (nous utiliserons NASM, mais libre a vous de prendre une autre version si vous vous sentez à l'aise).

Concernant nos profils, nous sommes 3 à avoir construit ce cours:

- Damien Nithard: Développeur/Architecte logiciel
- Jory Grezczszak: Développeur embarqué
- Enzo Hugonnier: Développeur/Architecte logiciel

Nous interviendrons chacun de nôtre côté au fur et à mesure de ce cours.
Le résultat final sera accessible via github à but de correction.

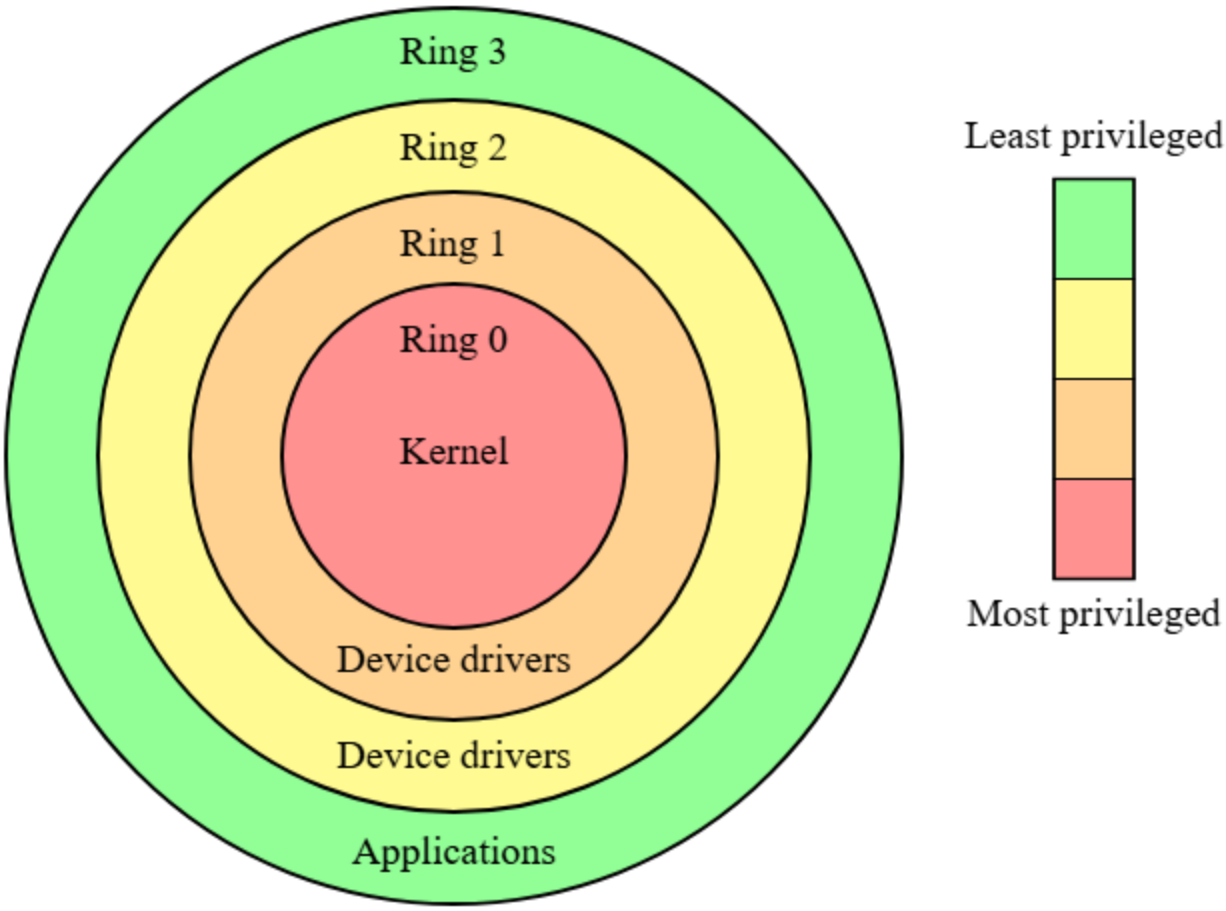
Différence kernel / userland

Avant de commencer à coder, nous passerons par une légère partie théorique pour comprendre les bases, nous vous conseillons de ne pas passer cette partie car elle vous sera utile tout au long de votre apprentissage.

Dans un processeur (CPU), il existe différent anneaux (de privilèges) , un peu comme des droits administrateurs.
Le ring 0 (aussi appelé kernel land), et le mode administrateur absolu au sein d'un CPU, tout est permis, ce qui peut facilement provoquer des plantages si on n’est pas prudent. On y retrouve tout le code qui doit accéder au hardware directement.

Les ring 1 et 2, souvent réservés aux drivers, mais rarement utilisés dans des architectures monolithiques (plus souvent dans des microkernels, ou systèmes expérimentaux), nous ignorerons donc ces deux anneaux pour se concentrer sur le 0 ainsi que le 3.

Le ring 3 (aussi appelé user land), anneau applicatif, l'utilisateur ne peux pas exécuter des instructions sensibles (écriture mémoire ou manipulation hardware).



NB: Ici nous parlons de mémoire physique, contrairement à la mémoire virtuelle qui est accessible par l'utilisateur, la différence sera détaillée ultérieurement.

Pourquoi monolithique ?

L'architecture monolithique est historique, inventée en 1960, une des raisons de pourquoi nous l'avons choisie, mais aussi grâce à son accessibilité, ici, nous coderons tout au même endroit.

Il existe de nouvelles variantes, plus sûre, comme l'architecture micronoyau, la ou l'idée est de minimiser le plus possible l'applicatif situé dans le kernel. Cette approche est plus sure, mais aussi plus complexe, ce pourquoi nous nous orientons vers un système plus classique.

Architecture x86/x86_64

Outils nécessaires (cross-compiler, QEMU, GDB)

Bootloader et démarrage de la machine

Ecriture d'un premier kernel

Partie II - Gestion mémoire

Segmentation et GDT

Interruptions et IDT

Paging et mémoire virtuelle

Allocateurs mémoire

Partie III - Intéractions avec le matériel

Programmation et interruptions matérielles (PIC/APIC)

Timer et multitâches basique

Drivers de périphériques simples

Partie IV - Processus et Syscalls

Structure de processus et scheduling

Appels système (syscalls)

Chargement et exécution des programmes

Partie V - Userland et système minimal

Le processus init

Ecriture d'un shell minimal

Gestion de fichiers simplifiée

Extension des syscalls

Partie VI - Conclusion et perspectives

Etat final du système

Optimisations possibles

Ouvertures