Projet de système d'exploitation
— Rock and Roll —

Benjamin Bonneau & Maxime Flin

13 mai 2020

Sommaire :

Introduction

ecos est un système d'exploitation conçu pour une architecture Intel x86 64 bits. Il propose une interface très proche des normes POSIX et fourni

- une gestion de processus concurrent
- une gestion de plusieurs systèmes de fichiers
- une implémentation de la librarie standard C
- un shell avec des programmes utilitaires courants (cat, ls, etc. . .)

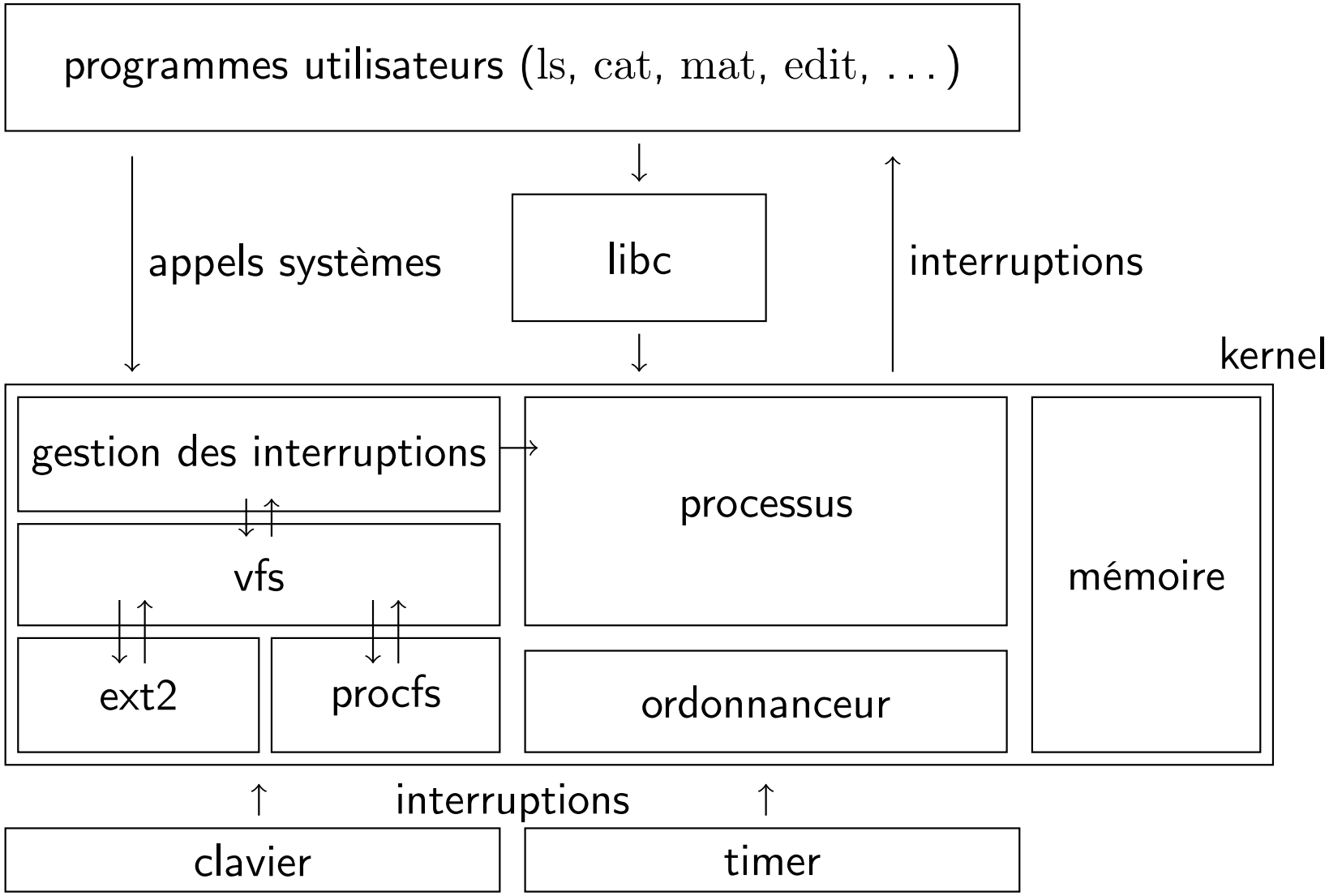


Figure – une vue globale du système

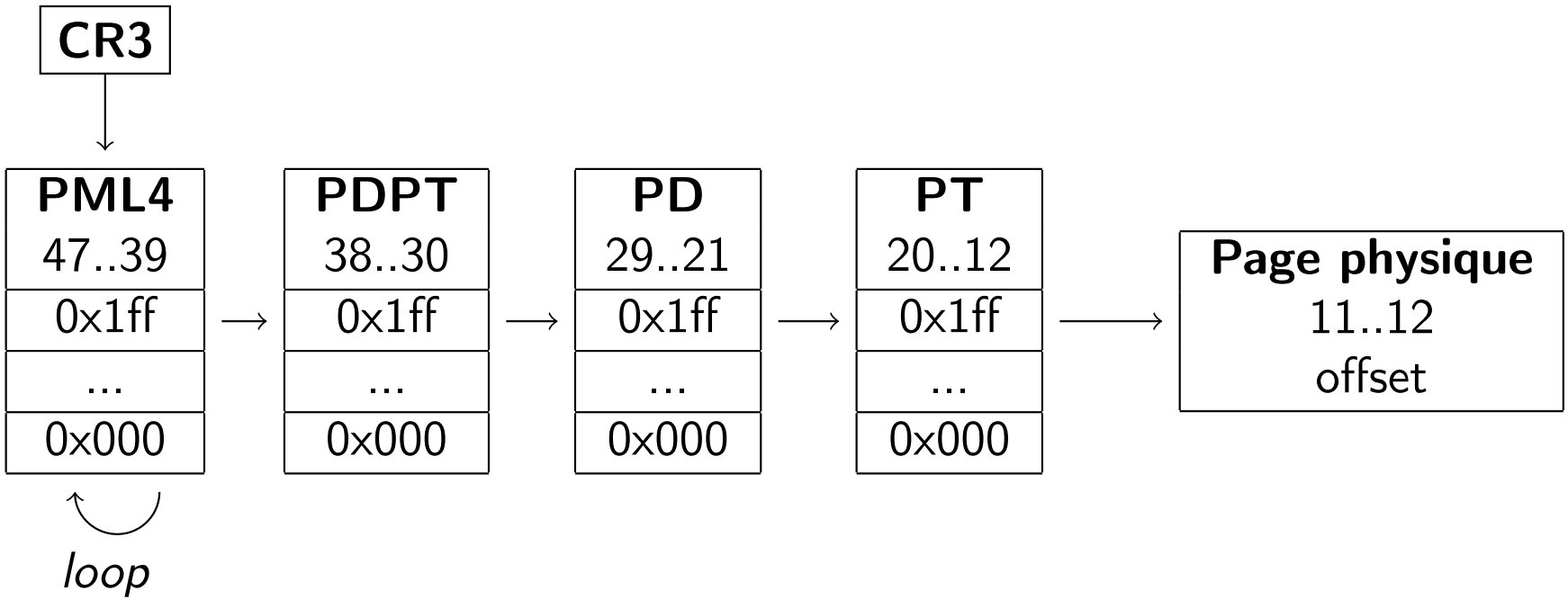
Boot

src/boot

- GRUB
  - kernel (ELF)
  - carte de la mémoire
  - affichage
- *protected mode 32 bits* vers *long mode 64 bits*
- Control Registers
- Paging
- Global Descriptor Table
- Chargement du kernel

Mémoire

Paging 4 niveaux : 48 bits d'adresse



Mémoire

Allocation des pages physiques disponibles

- découpage de la mémoire en blocs de 2Mo
- au niveau d'un bloc : arbre d'arité 8 (3 niveaux)
- pour gérer l'ensemble des blocs : 2 arbres d'arité 64. Blocs entièrement libres et blocs partiellement libres.

Mémoire

- chaque processus dispose de son PML4
- partie basse : userspace
- partie haute : kernel
- entrées spéciales dans la partie userspace :
  - Pages partagées (libc)
  - Pages allouées lors de l'accès (.bss, pile)
  - Pages copiées lors de l'accès (fork)

  actions effectuées lors de l'accès détecté par un page fault

Execve

- effectué par un processus auxiliaire avec privilèges de niveau 1
- gestion des fichiers spécifiant un interpréteur (#!)
- chargement de fichiers ELF
- création d'un nouveau paging
- allocation et chargement de sections depuis le fichier : .text, .data, .rodata
- marquage des pages à allouer lors de l'accès : .bss (initialisé à 0), pile
- ajout de la libc
- copie des arguments et paramêtres d'environnement

Processus

On maintient une tables de processus dans laquelle on sauvegarde pour chaque pid toutes les informations dont on pourrait avoir besoin :

- le status du processus associé (Libre, Zombie, Endormi, Run, ...)
- l'état des registres lors de la dernière interruption
- l'identifiant du paging
- la table des descripteurs de fichiers

Pour éviter d'avoir à parcourir toute la table on a ajouté aux processus une structure de liste chainée.

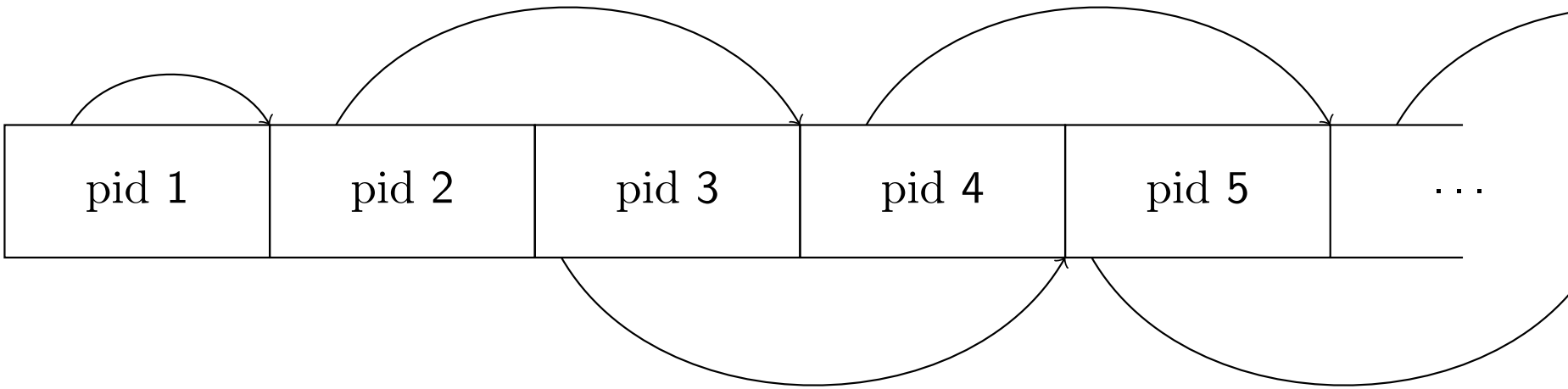


Figure – Table de processus avec la liste des enfants de 1 (en haut) et la liste des pid libres (en bas)

Ordonnacement des processus

On utilise une file de priorité pour classer les processus en fonction de leurs états. Dans la file, on trouve tous les processus qui attente de pouvoir retravailler.
Le noyau reçoit régulièrement des interruptions du PIT [1] qui interrompent le processus en cours. Si celui-ci travaille déjà depuis plus d'une certaine unité de temps [2], alors on laisse travailler un autre processus. De cette manière, on évite les famines.
On distingue deux processus particulers :

- le processus init, de pid 1, est le premier processus lancé par le noyau au moment du boot. Il lance un shell et attend que celui-ci termine pour le relancer.
- le processus stop, de pid 2, est utilisé en cas de panique du noyau.

1. Programmable Interval Timer
2. définie dans kernel/params.h

Système de fichier

On n'interragit avec aucune device réelle, par manque de temps, nous avons décidé de simplement simuler les comportement d'un tel composant. Lors de la compilation, nous générons un fichier binaire qui représente le système de fichier et le chargeons directement dans le binaire du noyau.
Néamoins nous avons la possibilité d'interagir avec plusieurs devices en même temps. Pour chacune de ces devices, on précise

- sa position dans le système de fichier parent (si ce dernier existe)
- le système de fichier à utiliser pour lire dans cette device
- des informations spécifique au point de montage.

Extended Filesystem 2

Un système de fichier développé dans les années 90 pour remplacer Minix, le système de fichier supporté par les premières versions de Linux. Les données sont réparties groupes de blocks pour essayer de maintenir les blocs d'un même fichiers adjacent.

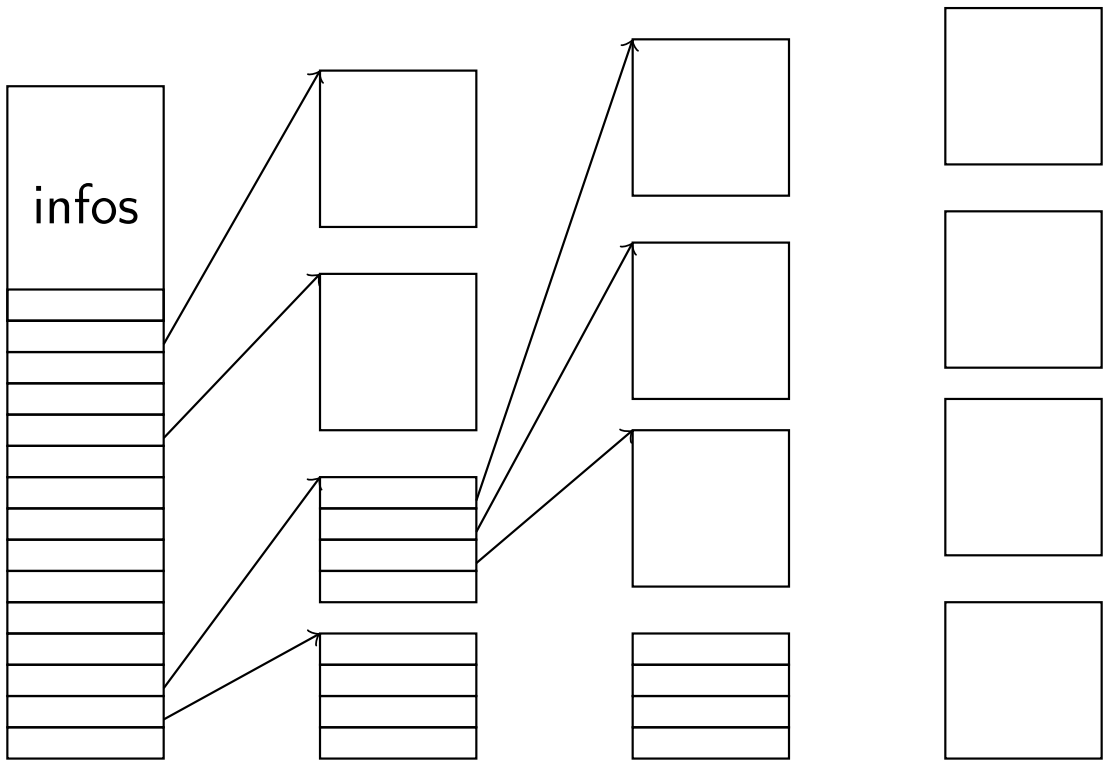


Figure – représentation d'un inoeud

procfs

procfs est un système de fichier virtuel qui permet d'obtenir des informations sur le système à travers une interface classique de fichiers. Notre implémentation de ce système de fichier est spécifique à notre système. Il se structure de la manière suivante :

- tty
- pipes
- null
- display
- un dossier pid pour chaque processus vivant
  - stat
  - cmd
  - fd

Cette interface est très pratique pour développer des outils comme ps ou lsof qui peuvent ainsi facilement accéder aux données du kernel sans avoir à ajouter des appels systèmes complexes.

Système de fichier virtuel

Pour que l'utilisateur puisse manipuler tous les fichiers du systèmes à travers une unique interface, on ajoute une couche d'indirection entre lui et les systèmes de fichiers.
C'est le système de fichier virtuel qui gère la hyérarchie entre les systèmes de fichier. Par exemple, dans notre démonstration $/$ est une partition $\mathrm{ext2}$ qui contient un dossier $/\mathrm{bin}$ et deux autres systèmes de fichiers : $/\mathrm{proc}$ et $/\mathrm{home}$.
On maintient une table de *fichiers virtuels* qui représentent les fichiers ouverts par le système. On y conserve le numéro d'inoeud du fichier et la device à laquell il appartient.
Le même fichier peut être ouvert plusieurs fois a des endroits différents. Pour représenter ceci, les processus n'agissent pas directement sur des fichiers virtuels mais sur des channels qui référencent un fichier virtuel, un mode (écriture, lecture ou les deux) et une position dans le fichier.

Figure – Les différentes couches d'indirections entre le fichier et le processus

Libc et Libk

Libc

Dans la libc, on trouve

- des fonctions utilitaires (strlen, malloc...)
- des abstractions au dessus des appels systèmes (opendir, fopen, ...)
- les appels systèmes, qui se sont plus que des int x80 précédé d'un mov de l'identifiant de de l'appel dans %rax.

Il y a également une séquence de controle qui permet d'initialiser la libc, de commencer son programme dans la fonction main et d'exit le programme au retour de main.
Pour éviter de dupliquer le code de la libc dans les binaires chargées dans le système de fichier, on la charge dynamiquement quand on lance le processus.
De plus, grâce aux interfaces les plus communes de la libc, du code générique respectant les normes POSIX pourrait être executé sur notre système.