

# OS

Par fiakx

## Il y a un début à tout, non ?

### Cross Compiler

#### Contexte

Quand tu développes un système d'exploitation, tu travailles généralement sur une machine (qu'on appellera **l'hôte**) qui a déjà un OS installé (comme Linux, Windows, ou macOS). Cependant, ton objectif est de créer un nouvel OS qui fonctionnera sur une autre machine (qu'on appellera ici **cible**), qui peut avoir une architecture matérielle différente (par exemple, x86, ARM, etc.).

Le problème est que le compilateur présent sur ta machine hôte est configuré pour produire des programmes qui fonctionnent sur cette machine hôte, pas sur la machine cible. C'est là qu'intervient la notion de **cross-compilation**.

#### Cross-Compilation

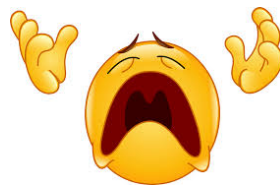
Un **cross-compilateur** est un compilateur qui fonctionne sur une plateforme (hôte) mais qui produit des exécutables pour une autre plateforme (cible). Par exemple, si tu développes sur un PC x86 sous Linux, mais que tu veux créer un OS pour une machine ARM, tu as besoin d'un cross-compilateur qui tourne sur x86 mais qui génère du code pour ARM.

#### Pourquoi est-ce nécessaire ?

La cross-compilation est essentielle pour plusieurs raisons :

- **Architecture différente** : Si la cible a une architecture matérielle différente de celle de l'hôte, le compilateur natif ne peut pas produire de code exécutable pour cette architecture.
- **Environnement de développement** : Tu veux pouvoir développer et tester ton OS sur une machine qui a déjà un OS fonctionnel, sans avoir à tout réécrire pour la cible.
- **Efficacité** : Utiliser un cross-compilateur te permet de générer rapidement des binaires pour la cible sans avoir à installer un environnement de développement complet sur la cible.

#### Oui, mais comment on installe tout ça ?



#### Télécharger touskifaut pour le compilateur :

Tu auras besoin des sources de GCC (GNU Compiler Collection) et de binutils (un ensemble d'outils pour manipuler les binaires).

#### Configurer le cross-compilateur :

Tu dois configurer GCC pour qu'il sache qu'il doit produire du code pour la cible et non pour l'hôte. Cela se fait généralement en spécifiant des options de configuration lors de la compilation de GCC.

Par exemple, si tu veux compiler pour une architecture ARM, tu utiliserais une commande comme :

```
1 ./configure --target=arm-none-eabi --prefix=/usr/local/cross
```

- **-target** spécifie l'architecture cible (ici, **arm-none-eabi**).
- **-prefix** indique où installer le cross-compilateur (ici, **/usr/local/cross**).

### Compiler et installer :

Une fois configuré, tu compiles et installes le cross-compileur. Cela peut prendre un certain temps car GCC est un gros projet.

### Utiliser le cross-compileur :

Une fois installé, tu peux utiliser ce compilateur pour générer des exécutables pour ta cible. Par exemple, si tu as un fichier C `hello.c`, tu peux le compiler avec :

```
1 arm-none-eabi-gcc -o hello hello.c
```

Cela produira un exécutable `hello` qui fonctionnera sur une machine ARM.



## Création de to premier kernel! (ou noyau en français)

Ici nous allons voir comment créer un noyau (*kernel*) minimal en C et assembleur pour une architecture x86. C'est quoi x86 ?.

### Prérequis

#### Outils nécessaires :

- **Compilateur** : GCC cross-compiler (ciblant `i686-elf` ou `x86_64-elf`)
- **Assembleur** : NASM (recommandé) ou GAS Voir plus.
- **Linker** : GNU LD
- **Émulateur** : QEMU, Bochs ou VirtualBox
- Mais comment on installe tout ça????

### Configuration du cross-compiler

Un cross-compiler est nécessaire pour éviter d'utiliser les bibliothèques de l'hôte. (Remonte en haut du Pdf pour configurer ton propre cross-compiler.)

### Structure du projet (Arborescence)

Les fichiers essentiels sont :

```
ton_projet/  
boot.asm    # Code assembleur de démarrage  
kernel.c    # Noyau principal en C  
linker.ld   # Script de linking \hyperref[fig:linking]{voir plus sur linking}.
```

### Code Assembleur (boot.asm)

#### Rôle

- Passe en mode 32 bits
- Initialise la pile (*stack*) La pile?.
- Appelle la fonction `kernel_main` en C (une fonction externe a l'assembleur)

## Exemple (NASM)

```
1 bits 32 ; Mode 32 bits
2 section .text
3 global start ; Point d'entree pour le linker
4 extern kernel_main ; Fonction principale en C
5
6 start:
7     mov esp, stack_top ; Initialise la pile, ESP = pointeur de pile, place en haut de la
8     zone reservee
9     call kernel_main ; Appel du noyau
10    hlt ; Arrete le CPU
11
12 section .bss
13 stack_bottom: resb 4096 ; Reserve 4 Ko pour la pile (stack_bottom : Adresse de base de la
14 pile.)
15 stack_top: ; stack_top : Adresse du sommet (debut effectif, car la pile descend en memoire)
```

## Code du Noyau (kernel.c)

### Fonction principale

Le noyau écrit directement dans la mémoire vidéo VGA (adresse 0xB8000).

```
1 void kernel_main() {
2     const char *str = "Hello, kernel World!";
3     unsigned short *vga_buffer = (unsigned short *)0xB8000;
4
5     for (int i = 0; str[i] != '\0'; i++) {
6         vga_buffer[i] = (unsigned short)str[i] | 0x0F00;
7         /* Couleur blanc sur noir */
8     }
9 }
```

## Script de Linking (linker.ld)

**Rôle** Organise les sections en mémoire et définit l'adresse de chargement (0x100000).

```
1 ENTRY(start) ; Point d'entree = 'start' (boot.asm)
2
3 SECTIONS {
4     . = 0x100000; ; Adresse de chargement
5
6     .text : {
7         *(.text) ; Code
8     }
9
10    .data : {
11        *(.data) ; Donnees initialisees
12    }
13
14    .bss : {
15        *(.bss) ; Donnees non initialisees
16    }
17 }
```

## Compilation et Linking

### Commandes

1. Assembler boot.asm :

```
1 nasm -f elf32 boot.asm -o boot.o
```

2. Compiler kernel.c :

```
1 i686-elf-gcc -c kernel.c -o kernel.o -std=gnu99 -ffreestanding -O2 -Wall -Wextra
```

3. Linker les objets :

```
1 i686-elf-ld -T linker.ld -o kernel.bin boot.o kernel.o -nostdlib
```

## Tester avec QEMU

Lancer le noyau avec :

```
1 qemu-system-i386 -kernel kernel.bin
```

## Prochaines Étapes

- Gestion des interruptions (IDT, PIC)
- Allocation mémoire (paging, heap)
- Pilotes (clavier, écran, etc.)

## Remarques Importantes

- Pas de librairie standard (`printf` ne fonctionne pas).
- La pile doit être initialisée avant d'utiliser du C.
- Le cross-compiler est obligatoire pour éviter des problèmes.

# Compléments

## Mini tuto sur comment installer les Prérequis pour ton premier Kernel!

Ici et dans tout le tutoriel, on utilisera une machine linux et un shell sous linux, ce tutoriel n'est absolument pas dédiée aux machines sous windows ou sous mac-Os

### Téléchargement et compilation de GCC :

```
1 wget https://ftp.gnu.org/gnu/gcc/gcc-12.2.0/gcc-12.2.0.tar.gz
2 tar xf gcc-12.2.0.tar.gz
3 mkdir build-gcc
4 cd build-gcc
5 ../gcc-12.2.0/configure --target=i686-elf --prefix=/usr/local/cross --disable-nls
6 --enable-languages=c,c++ --without-headers
7 make all-gcc -j$(nproc)
8 sudo make install-gcc
```

### Assembleur NASM

— Installation directe :

```
1 sudo apt install nasm
```

— Vérification :

```
1 nasm -v
```

### Linker GNU LD

— Inclus dans binutils :

```
1 sudo apt install binutils
```

— Pour la version cross-compilée :

```
1 # Installée automatiquement avec le cross-compilateur GCC
2 /usr/local/cross/bin/i686-elf-ld --version
```

### Emulateurs

— QEMU :

```
1 sudo apt install qemu-system-x86
```

— Bochs :

```
1 sudo apt install bochs bochs-x
```

— VirtualBox :

```
1 sudo apt install virtualbox
```

## Gnagnagna tu nous parles de x86 mais a quoi ça sert ???

- **Pour exécuter des programmes (logiciels, jeux, OS)**
  - Tous les programmes que tu lances (Chrome, Photoshop, Windows, Linux) sont écrits pour fonctionner sur une architecture spécifique.
  - x86 (et son extension x86-64) est la norme des PC depuis 40 ans.
- **Pour l'assembleur (le langage machine)**
  - L'assembleur x86 est le langage que le CPU comprend directement (ex : MOV EAX, 42 = "mets la valeur 42 dans le registre EAX").
  - Les compilateurs (C++, Rust, etc.) traduisent ton code en instructions x86 pour que le CPU l'exécute.
  - CPU = (Central Processing Unit), un microprocesseur installé sur la carte mère de l'ordinateur.
- **Pour l'encodage des instructions machine (bytes/code binaire)**
  - Chaque instruction (ADD, JMP, etc.) est encodée en binaire (ex : B8 2A 00 00 00 = MOV EAX, 42 en hexadécimal).
  - Le CPU lit ces bytes et les exécute.
- **Pour la compatibilité**
  - Un exécutable compilé pour x86 (32 bits) peut tourner sur un PC moderne en mode de compatibilité, même si le CPU est en 64 bits.

## Qu'est-ce que NASM ?

NASM (Netwide Assembler) est un assembleur libre et open-source pour les architectures x86 et x86-64 (processeurs Intel/AMD). Il permet d'écrire des programmes directement en langage assembleur, un langage de bas niveau proche du langage machine.

- **À quoi sert NASM ?**
  - Contrôle précis du matériel : Optimiser des morceaux critiques de code (ex : jeux vidéo, noyaux de systèmes d'exploitation) car en effet coder à bas niveaux proche de la machine permet une meilleure efficacité.
  - Reverse engineering : Analyser ou modifier des binaires compilés.
  - Apprentissage : Comprendre comment fonctionne un CPU en pratique.
- **Exemple de code NASM (x86)**

```
1      section .text
2  global _start
3
4  _start:
5      mov eax, 4      ; Appel système "write" (4)
6      mov ebx, 1      ; Sortie standard (1)
7      mov ecx, message ; Adresse du message
8      mov edx, len     ; Longueur du message
9      int 0x80         ; Interruption noyau
10
11     mov eax, 1      ; Appel système "exit" (1)
12     int 0x80
13
14  section .dataExemple de code NASM (x86)
15  message db 'Hello, World!', 0xA ; Message + saut de ligne
16  len     equ $ - message          ; Calcul de la longueur
```

## La Pile ?

- **La pile est une structure LIFO (Last In, First Out) utilisée pour :**
  - Stocker temporairement des variables locales.
  - Sauvegarder des adresses de retour lors d'appels de fonctions (call).
  - Passer des arguments aux fonctions.
- **Fonctionnement de la Pile en x86**
  - La pile grandit vers les adresses basses (décroissante).
  - push eax : Décrémenter esp de 4, puis stocker eax à [esp].
  - pop eax : Récupérer la valeur à [esp], puis incrémenter esp de 4.
- **Exemple d'utilisation :**

```
1      push 0x42      ; Empiler la valeur 0x42 (esp -= 4)
2      push eax       ; Empiler le registre eax (esp -= 4)
3      pop ebx        ; Dépiler dans ebx (esp += 4)
```

Voir plus sur les registres.

## Les Registres en Assembleur

Un **registre** est une petite zone de mémoire **ultra-rapide** située directement dans le CPU. C'est comme une "variable matérielle" utilisée pour stocker des données temporaires pendant l'exécution.

- **À quoi ça sert ?**
  - Manipuler des données (calculs, comparaisons)
  - Stocker des adresses mémoire (pointeurs)
  - Contrôler le flux d'exécution (sauts, appels de fonctions)

## Exemples de Registres (x86/x64)

- **Registres généraux (32/64 bits) :**
  - EAX/RAX : Accumulateur (résultats de calculs)
  - EBX/RBX : Base (adresses mémoire)
  - ECX/RCX : Compteur (boucles)
  - EDX/RDX : Données (opérations I/O)

## Registres spéciaux :

- ESP/RSP : Pointeur de pile (*stack*)
- EIP/RIP : Pointeur d'instruction (adresse suivante à exécuter)

## Le Linking

Le **linking** (ou *édition de liens*) est l'étape finale de la compilation qui assemble plusieurs fichiers objets (`.o/.obj`) et bibliothèques (`.a/.lib`, `.so/.dll`) pour produire un **exécutable unique** (ou une bibliothèque).

### • À quoi ça sert ?

- Combiner plusieurs modules compilés séparément.
- Résoudre les références entre fichiers (ex : appels de fonctions).
- Inclure des bibliothèques externes (ex : `printf` de la `libc`).

## Explication de l'Adresse de Chargement (0x100000)

L'adresse de chargement (0x100000, soit 1 Mégaoctet en hexadécimal) est l'endroit en mémoire physique où votre noyau (kernel) sera chargé et exécuté. Cette valeur n'est pas choisie au hasard : elle est cruciale pour le fonctionnement d'un OS sur architecture x86.

### Pourquoi 0x100000 (1 Mo) ?

**a) Mémoire sous 1 Mo = Zone réservée** Sur les processeurs x86, la mémoire est divisée en plusieurs zones :

Adresse	Usage
0x00000–0x9FFFF	Mémoire conventionnelle (BIOS, interruptions, DOS, matériel)
0xA0000–0xFFFFF	Mémoire vidéo (VGA) + ROM BIOS (accès matériel)
0x100000+	Mémoire étendue (libre pour le noyau)

Si vous placez votre noyau en dessous de 1 Mo, vous risquez d'écraser :

- Le BIOS (qui utilise 0xF0000–0xFFFFF).
- La mémoire vidéo (écran texte VGA en 0xB8000).
- Les vecteurs d'interruptions (en 0x0–0x3FF).

**b) Au-delà de 1 Mo = Espace libre** À partir de 0x100000, la mémoire est non réservée et peut être utilisée librement par le noyau.

C'est la première adresse hors des limitations du mode réel x86.

### Que se passe-t-il si on met une autre adresse ?

Adresse	Conséquence
0x0–0x9FFFF	Crash garanti (écrasement du BIOS/DOS/interruptions).
0xA0000–0xFFFFF	Affichage corrompu (écriture dans la mémoire vidéo VGA).
0x200000 (2 Mo)	Fonctionne, mais inutile (perte d'espace entre 1 Mo et 2 Mo).
0x100000 (1 Mo)	Parfait : première adresse sûre et standard pour un noyau.

### Est-ce que cette adresse est fixe ?

Non ! Elle dépend de :

- **L'architecture** :
  - x86 32 bits : 0x100000 (standard pour les noyaux comme Linux).
  - x86 64 bits : Certains noyaux utilisent 0x200000 (2 Mo) pour l'alignement.
  - ARM/RISC-V : D'autres adresses sont utilisées (ex : 0x8000 pour le Raspberry Pi).
- **Le bootloader** :
  - GRUB, Limine ou UEFI peuvent charger le noyau à une adresse différente.
  - Votre linker script doit correspondre à ce que le bootloader attend.

### Comment vérifier où est chargé votre noyau ?

— **En debug avec QEMU** :

```
qemu-system-x86_64 -kernel kernel.bin -display curses -d in_asm
```

Cherchez Loading kernel at 0x100000.

— En inspectant le binaire :  
objdump -x kernel.bin | grep "start address"

### Exemple Simple

```
1 gcc main.o utils.o -o programme # Linking de 2 fichiers objets
```

sources : [wiki.osdev.org](http://wiki.osdev.org), [reddit.com](http://reddit.com), [stackoverflow.com](http://stackoverflow.com),  
operating system concepts par (Silberchatz, Gavin, Gagne)