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 ?

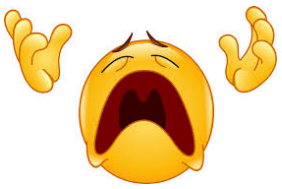
La cross-compilation est essentielle pour plusieurs raisons :

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

Mais pas de problème je t'ai concocté un petit PDF qui te permettra de tout installer assez rapidement (y compris le nécessaire pour la suite du dévellopement du kernel)
(voir https://github.com/Fiakx/Os/blob/main/Guide_Install.pdf)

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 librairies 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)

```
    bits 32                      ; Specifie l'assemblage en mode 32 bits

    section .multiboot           ; Section pour l'en-tete Multiboot
        align 4                  ; Alignement sur 4 octets (requis par Multiboot)
            dd 0x1BADB002        ; Magic number Multiboot (identifie le noyau)
            dd 0x00             ; Flags (aucun flag active)
            dd -(0x1BADB002 + 0x00) ; Checksum (magic + flags + checksum = 0)

    section .text                ; Section du code executable
    global start                ; Rend le symbole 'start' visible a l'editeur de liens
    extern kernel_main          ; Declare que kernel_main est defini ailleurs

    start:
            cli                  ; Desactive les interruptions (Clear Interrupt flag)
```

Des Flags???? Checksum???

Code du Noyau (kernel.c)

Fonction principale

Normalement pour dévelloper un kernel pouvant afficher du texte, on ecrit le noyau de manière a ce qu'il agisse directement dans la mémoire vidéo VGA (adresse `0xB8000`).
Comme ça :

```
    void kernel_main() {
        // Chaine a afficher
        const char *str = "Hello, kernel World!";

        // Pointeur vers le buffer VGA (memoire texte 80x25)
        unsigned short *vga_buffer = (unsigned short *)0xB8000;

        // Boucle d'affichage caractere par caractere
        for (int i = 0; str[i] != '\0'; i++) {
            // Combine le caractere avec les attributs (couleur)
            vga_buffer[i] = (unsigned short)str[i] | 0x0F00;
            /* Format:
             * Bits 0-7: caractere ASCII
             * Bits 8-11: couleur avant-plan (0xF = blanc)
             * Bits 12-15: couleur arriere-plan (0x0 = noir)
             */
        }
    }
```

Cependant de nos jour la mémoire VGA traditionnelle (0xB8000) n'existe plus sur les systèmes modernes en mode UEFI. Nous ne pouvons donc plus l'utiliser.
Le code que vous avez vu ci dessus fonctionne parfaitement sur des vielles machines, il fonctionne aussi si vous voulez l'essayer sur une machine virtuelle comme quemu ou virtualbox.
Pour remédier a ça nous utiliserons un Framebuffer (ou tampon d'image). C'est une zone de mémoire vive (RAM) qui stocke les données brutes des pixels affichés à l'écran.
C'est la méthode moderne pour gérer l'affichage graphique, remplaçant l'ancien mode texte VGA.
(ça arrive bientot)

Script de Linking (linker.ld)

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

```
    /* Point d'entree du noyau - correspond au symbole 'start' dans boot.asm */
ENTRY(start)

/* Definition des sections memoire */
SECTIONS {
    /* Le noyau sera charge a l'adresse 1 Mebioctet (0x100000) */
    . = 1M;

    /* Section du code executable */
    .text BLOCK(4K) : ALIGN(4K) {
        *(.multiboot)  /* D'abord le header Multiboot (doit etre au debut) */
        *(.text)       /* Puis tout le code des fichiers objets */
    }

    /* Section des donnees en lecture seule (constantes, etc.) */
    .rodata BLOCK(4K) : ALIGN(4K) {
        *(.rodata)     /* Toutes les sections .rodata */
    }

    /* Section des donnees initialisees (variables globales initialisees) */
    .data BLOCK(4K) : ALIGN(4K) {
        *(.data)       /* Toutes les sections .data */
    }

    /* Section des donnees non-initialisees (BSS = Block Started by Symbol) */
    .bss BLOCK(4K) : ALIGN(4K) {
        *(COMMON)      /* Variables globales non initialisees (communes) */
        *(.bss)       /* Toutes les sections .bss */
        *(.stack)     /* Notre pile definie dans boot.asm */
    }

    /* Sections a ignorer dans le fichier final */
    /DISCARD/ : {
        *(.comment)    /* Commentaires du compilateur */
        *(.note*)      /* Toutes les sections commencant par .note */
    }
}
```

Compilation et Linking

Commandes

1. Assembler `boot.asm` :

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

2. Compiler `kernel.c` :

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

3. Linker les objets :

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

Tester avec QEMU

Lancer le noyau avec :

```
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 aussi, il faudra que tu suive ce tutoriel (ne reinstalle pas ce que tu as déja installé ça ne sert a rien).
voir https://github.com/Fiakx/Os/blob/main/Guide_Install.pdf

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'es 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 a bas niveaux proche de la machine permet une meilleur efficacité.
  — Reverse engineering : Analyser ou modifier des binaires compilés.
  — Apprentissage : Comprendre comment fonctionne un CPU en pratique.
- **Exemple de code NASM (x86)**

```
    section .text
global _start

_start:
    mov eax, 4        ; Appel systeme "write" (4)
    mov ebx, 1        ; Sortie standard (1)
    mov ecx, message  ; Adresse du message
    mov edx, len      ; Longueur du message
    int 0x80          ; Interruption noyau

    mov eax, 1        ; Appel systeme "exit" (1)
    int 0x80

section .dataExemple de code NASM (x86)
message db 'Hello, World!', 0xA  ; Message + saut de ligne
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 : Décrémente esp de 4, puis stocke eax à [esp].
— pop eax : Récupère la valeur à [esp], puis incrémente esp de 4.

- **Exemple d'utilisation :**

```
push 0x42       ; Empile la valeur 0x42 (esp -= 4)
push eax        ; Empile le registre eax (esp -= 4)
pop ebx         ; Depile 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"
```

Explication des " flags " et de " Checksum "

Dans l'en-tête **Multiboot**, les **flags** sont des bits indiquant au bootloader (ex : GRUB) les fonctionnalités requises par le noyau. Dans le code initial :

```
dd 0x00  ; Flags désactivés
```

Définition des Flags

Les flags courants sont définis par des bits individuels :

$$\text{Flags} = \sum(\text{Bit}_i \times 2^i)$$

Options principales

Hex	Bit	Description
0x01	0	Alignement des modules (chargeurs externes)
0x02	1	Informations mémoire (`mem_lower`, `mem_upper`)
0x04	2	Mode vidéo (framebuffer)
0x100	8	Carte mémoire détaillée

Exemple d'utilisation

Pour demander la mémoire et l'alignement des modules :

```
dd 0x03  ; 0x01 (Bit 0) + 0x02 (Bit 1)
```

Accès aux données

Si `flags & 0x02` est activé, GRUB stocke les infos mémoire dans une structure accessible via `EBX` :

```
struct multiboot_info {
    uint32_t flags;
    uint32_t mem_lower;  // Mémoire basse (KB)
    uint32_t mem_upper;  // Mémoire haute (KB)
    // ...
};
```

Checksum

Le **checksum** est un mécanisme de contrôle d'intégrité pour vérifier que l'en-tête Multiboot est valide.

Fonctionnement

— **Formule** :

$$\text{Checksum} = -(\text{Magic} + \text{Flags})$$

  — `Magic` = `0x1BADB002` (constante Multiboot)
  — `Flags` = Options activées (ex : `0x00`, `0x03`)

— **Vérification** : Le bootloader calcule :

$$\text{Magic} + \text{Flags} + \text{Checksum} = 0$$

Exemple

Avec `Flags = 0x03` :

$$\text{Checksum} = -(0x1BADB002 + 0x03) = \texttt{0xE4524FFB}$$

Utilité

— Empeche de charger un noyau corrompu.
— Garantit que le bootloader et le noyau se comprennent.

```
; Exemple en assembleur
dd 0x1BADB002   ; Magic
dd 0x03         ; Flags
dd 0xE4524FFB   ; Checksum
```

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