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

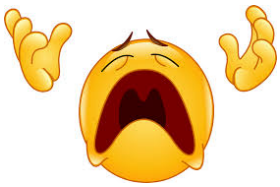
Un **cross-compilateur** est un compilateur qui fonctionne sur une plateforme (hôte) mais qui produit des exécutables pour une autre plateforme (cible). Par exemple, si tu développes sur un PC x86 sous Linux, mais que tu veux créer un OS pour une machine ARM, tu as besoin d'un cross-compilateur qui tourne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

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 :

```
./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-compilateur. Cela peut prendre un certain temps car GCC est un gros projet.

Utiliser le cross-compilateur 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 :

```
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

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           ; Mode 32 bits
section .text
global start      ; Point d'entree pour le linker
extern kernel_main ; Fonction principale en C

start:
    mov esp, stack_top ; Initialise la pile, ESP = pointeur de pile, place en haut de la zone reservee
    call kernel_main   ; Appel du noyau
    hlt                ; Arrete le CPU

section .bss
stack_bottom: resb 4096 ; Reserve 4 Ko pour la pile (stack_bottom : Adresse de base de la pile.)
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`).

```
void kernel_main() {
    const char *str = "Hello, kernel World!";
    unsigned short *vga_buffer = (unsigned short *)0xB8000;

    for (int i = 0; str[i] != '\0'; i++) {
        vga_buffer[i] = (unsigned short)str[i] | 0x0F00;
        /* Couleur blanc sur noir */
    }
}
```

Script de Linking (linker.ld)

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

```
ENTRY(start)            ; Point d'entree = 'start' (boot.asm)

SECTIONS {
    . = 0x100000;       ; Adresse de chargement

    .text : {
        *(.text)        ; Code
    }

    .data : {
        *(.data)        ; Donnees initialisees
    }

    .bss : {
        *(.bss)         ; Donnees non initialisees
    }
}
```

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.

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).

Exemple Simple

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

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