OS

Par fiakx

1 Il y a un début à tout, non ?

1.1 Cross Compiler

1.1.1 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**.

1.1.2 Cross-Compilation

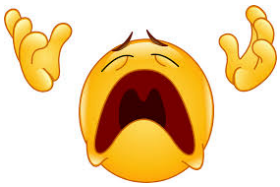
Un **cross-compilateur** est un compilateur qui fonctionne sur une plateforme (hôte) mais qui produit des exécutables pour une autre plateforme (cible). Par exemple, si tu développes sur un PC x86 sous Linux, mais que tu veux créer un OS pour une machine ARM, tu as besoin d'un cross-compilateur qui tourne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

1.1.3 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.

1.1.4 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.

1.2 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.

1.2.1 Prérequis

Outils nécessaires :

- **Compilateur** : GCC cross-compiler (ciblant `i686-elf` ou `x86_64-elf`)
- **Assembleur** : NASM (recommandé) ou GAS
- **Linker** : GNU LD
- **Émulateur** : QEMU, Bochs ou VirtualBox

1.2.2 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.)

1.2.3 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
```

1.2.4 Code Assembleur (boot.asm)

Rôle

- Passe en mode 32 bits
- Initialise la pile (*stack*)
- Appelle la fonction `kernel_main` en C

1.2.5 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
    call kernel_main   ; Appel du noyau
    hlt                ; Arrete le CPU

section .bss
stack_bottom: resb 4096 ; Reserve 4 Ko pour la pile
stack_top:
```

1.3 Code du Noyau (kernel.c)

1.3.1 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 */
    }
}
```

1.3.2 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
    }
}
```

1.3.3 Compilation et Linking

1.3.4 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
```

1.3.5 Tester avec QEMU

Lancer le noyau avec :

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

1.3.6 Prochaines Étapes

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

1.3.7 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.

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