[Comment écrire son propre kernel x86 ? - Geek Madagascar](https://geek.mg/fr/tutoriel/comment-ecrire-son-propre-kernel-x86/)

**Comment fonctionne une machine x86**

Avant de penser à écrire un noyau, voyons comment la machine démarre et transfère le contrôle au noyau:

La plupart des registres du processeur x86 ont des valeurs bien définies après la mise sous tension. Le registre EIP (Instruction Pointer) contient l’adresse mémoire de l’instruction exécutée par le processeur. EIP est codé en dur à la valeur ***0xFFFFFFF0*** . Ainsi, le processeur (x86) est câblé pour commencer l’exécution à l’adresse physique ***0xFFFFFFF0***. Il s’agit des 16 derniers octets de l’espace d’adressage 32 bits. Cette adresse mémoire est appelée **vecteur de réinitialisation**.

Maintenant, le chipset de la RAM s’assure que ***0xFFFFFFF0*** est mappé à une certaine partie du BIOS, pas à la RAM. Pendant ce temps, le BIOS se copie dans la RAM pour un accès plus rapide. C’est ce qu’on appelle**shadowing**. L’adresse ***0xFFFFFFF0*** contiendra juste une instruction de saut à l’adresse dans la mémoire où le BIOS s’est copié.

Ainsi, le code du BIOS commence son exécution. Le BIOS recherche d’abord un périphérique amorçable dans l’ordre de périphérique configuré. Il vérifie un certain nombre magique pour déterminer si le périphérique est amorçable ou non. (*si les octets 511 et 512 du premier secteur sont 0xAA55*)

Une fois que le BIOS a trouvé un périphérique amorçable, il copie le contenu du premier secteur du périphérique dans la RAM à partir de l’adresse physique *0x7c00***,** puis saute dans l’adresse et exécute le code qui vient d’eêre chargé. C’est le **bootloader**.

Le bootloader charge ensuite le noyau à l’adresse physique ***0x100000***. L’adresse ***0x100000*** est utilisée comme adresse de début pour tous les *gros* noyaux sur les machines x86.

Tous les processeurs x86 commencent dans un mode 16 bits simpliste appelé ***mode réel*** . Le chargeur de démarrage [GRUB](https://www.gnu.org/software/grub/) passe en mode protégé 32 bits en définissant le bit du registre CR0 le plus bas sur 1. Ainsi, le noyau se charge en mode protégé 32 bits.

Notez que dans le cas du noyau linux, GRUB détecte le protocole de démarrage de Linux et charge le noyau Linux en mode réel. Le noyau Linux fait lui-même le passage en mode protégé.

Le registre CR0  a une longueur de 32 bits sur les processeurs 386 et supérieurs. Sur les processeurs x86-64 en mode long, il (et les autres registres de contrôle) a une longueur de 64 bits. CR0 a divers indicateurs de contrôle qui modifient le fonctionnement de base du processeur.

|  |  |  |  |
| --- | --- | --- | --- |
| **BIT** | **NAME** | **FULL NAME** | **DESCRIPTION** |
| 0 | PE | Protected Mode Enable | If 1, system is in [protected mode](https://en.wikipedia.org/wiki/Protected_mode" \o "Protected mode), else system is in [real mode](https://en.wikipedia.org/wiki/Real_mode) |
| 1 | MP | Monitor co-processor | Controls interaction of WAIT/FWAIT instructions with TS flag in CR0 |
| 2 | EM | Emulation | If set, no x87 [floating point unit](https://en.wikipedia.org/wiki/Floating_point_unit" \o "Floating point unit) present, if clear, x87 FPU present |
| 3 | TS | Task switched | Allows saving x87 task context upon a task switch only after x87 instruction used |
| 4 | ET | Extension type | On the 386, it allowed to specify whether the external math coprocessor was an [80287](https://en.wikipedia.org/wiki/80287) or [80387](https://en.wikipedia.org/wiki/80387) |
| 5 | NE | Numeric error | Enable internal [x87](https://en.wikipedia.org/wiki/X87) floating point error reporting when set, else enables PC style x87 error detection |
| 16 | WP | Write protect | When set, the CPU can’t write to read-only pages when privilege level is 0 |
| 18 | AM | Alignment mask | Alignment check enabled if AM set, AC flag (in [EFLAGS](https://en.wikipedia.org/wiki/FLAGS_register_(computing)) register) set, and privilege level is 3 |
| 29 | NW | Not-write through | Globally enables/disable write-through caching |
| 30 | CD | [Cache](https://en.wikipedia.org/wiki/CPU_cache) disable | Globally enables/disable the memory cache |
| 31 | PG | Paging | If 1, enable paging and use the CR3 register, else disable paging |

**Le point d’entrée en assembleur**

Nous écrirons un petit fichier en langage assembleur x86 qui servira de point de départ pour notre noyau. Tout ce que notre fichier fera est d’invoquer une fonction externe que nous écrirons en C, puis stoppera le flux du programme. Afin de  s’assurer que ce code servira de point de départ du noyau. Nous allons utiliser un script d’éditeur de liens (linker) qui lie les fichiers objets pour produire l’exécutable final du noyau. Dans ce script d’éditeur de liens (linker), nous spécifierons explicitement que nous voulons que notre binaire soit chargé à l’adresse *0x100000*. Cette adresse est l’endroit où l’on s’attend à ce que le noyau soit. Ainsi, le bootloader prendra soin de déclencher le point d’entrée du noyau.

Voici notre code:

;;kernel.asm

bits 32 ;directive 32 bit pour nasm

section .text

global start

extern kmain ;kmain est défini dans le fichier c

start:

cli ;bloque les interruptions

mov esp, stack\_space ;défini le pointeur de la pile

call kmain

hlt ;défini le pointeur de la pile

section .bss

resb 8192 ;8 Ko pour la pile

stack\_space:

La première instruction bits 32  n’est pas une instruction d’assembeur x86. C’est une directive à l’assembleur NASM qui spécifie qu’il devrait générer du code pour fonctionner sur un processeur fonctionnant en mode 32 bits. Ce n’est pas obligatoire dans notre exemple, mais il est inclus ici car c’est une bonne pratique d’être explicite.

La deuxième ligne commence la section texte ( section de code). C’est là que nous mettons tout notre code.

global  est une autre directive NASM pour déclarer les symboles du code comme globaux. Comme cela, l’éditeur de liens sait où se trouve le symbole start;  qui est notre point d’entrée.

kmain est la fonction qui sera définie dans notre fichier kernel.c. extern  déclare que la fonction est déclarée ailleurs.

Ensuite, nous avons la fonction start , qui appelle la fonction kmain  et arrête le CPU en utilisant l’ instruction hlt . Les interruptions peuvent relancer le processeur d’une instruction hlt . Nous désactivons donc les interruptions à l’avance en utilisant l’instruction cli (**clear-interrupts)**.

Nous devrions idéalement mettre de côté de la mémoire pour la pile et y pointer le pointeur de la pile (esp). Cependant, il semble que GRUB le fasse pour nous et le pointeur de la pile est déjà défini à ce stade. Mais, pour en être sûr, nous allons allouer de l’espace dans la section BSS et pointer le pointeur de la pile au début de la mémoire allouée. Nous utilisons l’instruction resb  qui réserve la mémoire donnée en octets. Après cela, il reste une étiquette qui pointera sur le bord de la partie réservée de la mémoire. Juste avant l’appel kmain, le pointeur de la pile ( esp) est amené à pointer vers cet espace en utilisant l’instruction mov.

**Le noyau en C**

Dans kernel.asm , nous avons fait un appel à la fonction kmain() . Donc, notre code C commencera à s’exécuter à kmain() au lieu du traditionel main:

/\*

\* kernel.c

\*/

void kmain(void)

{

const char \*str = "my first kernel";

char \*vidptr = (char\*)0xb8000; //video mem begins here.

unsigned int i = 0;

unsigned int j = 0;

/\* this loops clears the screen

\* there are 25 lines each of 80 columns; each element takes 2 bytes \*/

while(j < 80 \* 25 \* 2) {

/\* blank character \*/

vidptr[j] = ' ';

/\* attribute-byte - light grey on black screen \*/

vidptr[j+1] = 0x07;

j = j + 2;

}

j = 0;

/\* this loop writes the string to video memory \*/

while(str[j] != '\0') {

/\* the character's ascii \*/

vidptr[i] = str[j];

/\* attribute-byte: give character black bg and light grey fg \*/

vidptr[i+1] = 0x07;

++j;

i = i + 2;

}

return;

}

Tout ce que notre noyau va faire, c’est effacer l’écran et afficher la chaîne “my first kernel”.

D’abord nous créer un pointeur vidptr qui pointe vers l’adresse ***0xb8000***. Cette adresse est le début de la mémoire vidéo en mode protégé. La mémoire de texte de l’écran est simplement un morceau de mémoire dans notre espace d’adressage. L’entrée/sortie mappée en mémoire pour l’écran commence à ***0xb8000*** et prend en charge 25 lignes, chaque ligne contient 80 caractères en ascii.

Chaque caractère dans cette mémoire de texte est représenté en 16 bits (2 octets), plutôt que 8 bits (1 octet). Le premier octet devrait avoir la représentation du caractère en ASCII. Le deuxième octet est le **attribute-byte**. Cela décrit la mise en forme du caractère, y compris les attributs tels que la couleur.

Pour imprimer le caractère ‘s’ en vert sur fond noir, nous allons stocker ce caractère dans le premier octet de l’adresse de la mémoire vidéo et la valeur ***0x02*** dans le deuxième octet.  
**0** représente le fond noir et **2** représente le premier plan vert.

Jetez un oeil à la table ci-dessous pour différentes couleurs:

0 - Black

1 - Blue

2 - Green

3 - Cyan

4 - Red

5 - Magenta

6 - Brown

7 - Light Grey

8 - Dark Grey

9 - Light Blue

10/a - Light Green

11/b - Light Cyan

12/c - Light Red

13/d - Light Magenta

14/e - Light Brown

15/f – White.

Dans notre noyau, nous allons utiliser le caractère gris clair sur un fond noir. Donc, notre attribut-octet doit avoir la valeur **0x07**.

Dans la première boucle while, le programme écrit le caractère vide avec l’attribut 0x07 sur les 80 colonnes des 25 lignes. Ceci efface ainsi l’écran.

Dans la deuxième boucle while, les caractères de la chaîne terminée par un caractère nul “my first kernel” sont écrits dans le morceau de la mémoire vidéo, chaque caractère contenant un octet d’attribut de ***0x07***.

**Le linker**

Nous allons assembler kernel.asm avec NASM en un fichier objet. Puis en utilisant [GCC,](https://gcc.gnu.org/) nous allons compiler kernel.ckernel.c  en un autre fichier objet. Maintenant, notre travail consiste à faire en sorte que ces objets soient liés en un noyau de démarrage exécutable.

Pour cela, nous utilisons un script d’éditeur de lien.

/\*

\* link.ld

\*/

OUTPUT\_FORMAT(elf32-i386)

ENTRY(start)

SECTIONS

{

. = 0x100000;

.text : { \*(.text) }

.data : { \*(.data) }

.bss : { \*(.bss) }

}

Tout d’abord, nous définissons le format de sortie de notre exécutable en 32 bits. ELF (Executable and Linkable Format) est le format de fichier binaire standard pour les systèmes de type Unix sur l’architecture x86.

**ENTRY** prend en argument le nom du symbole qui devrait être le point d’entrée de notre exécutable.

**SECTIONS** est la partie la plus importante pour nous. Ici, nous définissons la disposition de notre exécutable.

Dans les accolades qui suivent l’instruction SECTIONS, le caractère point (.) Représente le compteur d’emplacement . Le compteur d’emplacement est toujours initialisé à ***0x0*** au début du bloc SECTIONS. Il peut être modifié en lui affectant une nouvelle valeur.

Rappelez-vous que le code du noyau devrait commencer à l’adresse ***0x100000***. Nous avons donc défini le compteur d’emplacement sur ***0x100000***.

La ligne suivante .text: {\* (. Text)}

L’astérisque (\*)  est un caractère générique qui correspond à n’importe quel nom de fichier. L’expression \*(.text)  signifie donc toutes les sections .text de toues les fichiers d’entrée.

Ainsi, l’éditeur de liens fusionne toutes les sections de texte des fichiers objet à la section de texte de l’exécutable, à l’adresse stockée dans le compteur d’emplacement. Ainsi, la section de code de notre exécutable commence à ***0x100000***.

De même, les sections data et bss sont fusionnées et placées aux valeurs then du compteur de position.

**Grub et Multiboot**

Maintenant, nous avons tous nos fichiers prêts. Mais, puisque nous aimons démarrer notre noyau avec le bootloader [GRUB](https://fr.wikipedia.org/wiki/GNU_GRUB), il reste une étape. Il existe une norme pour le chargement de noyaux x86 à l’aide d’un chargeur de démarrage.

[GRUB](https://fr.wikipedia.org/wiki/GNU_GRUB) chargera seulement notre noyau s’il est conforme à la spécification Multiboot .

Selon la spécification, le noyau doit contenir un en-tête (connu sous le nom d’en-tête Multiboot) dans ses 8 premiers KiloBytes. De plus, cet en-tête Multiboot doit contenir 3 champs alignés sur 4 octets, à savoir:

* Un champ ‘magic’ : contenant le nombre magique 0x1BADB002 , pour identifier l’en-tête.
* Un champ de flag : Nous ne nous soucierons pas de ce champ. Nous allons simplement le mettre à zéro.
* Un champ de somme de contrôle : le champ de somme de contrôle ajouté aux champs ‘magic’ et ‘flags’ doit donner zéro.

Ainsi notre kernel.asm  deviendra:

;;kernel.asm

;nasm directive - 32 bit

bits 32

section .text

;multiboot spec

align 4

dd 0x1BADB002 ;magic

dd 0x00 ;flags

dd - (0x1BADB002 + 0x00) ;checksum. m+f+c should be zero

global start

extern kmain ;kmain is defined in the c file

start:

cli ;block interrupts

mov esp, stack\_space ;set stack pointer

call kmain

hlt ;halt the CPU

section .bss

resb 8192 ;8KB for stack

stack\_space:

**Compiler le noyau**

Nous allons maintenant créer les fichiers d’objets à partir de kernel.asm  et kernel.c  puis le lier à l’aide du script de liaison.

La commande suivante exécute NASM pour créer le fichier objet kasm.o au format ELF-32.

nasm -f elf32 kernel.asm -o kasm.o

L’option ‘-c’ s’assure qu’après la compilation, la liaison ne se fait pas implicitement.

gcc -m32 -c kernel.c -o kc.o

Ceci exécutera ld avec notre script d’éditeur de liens et générera l’exécutable nommé kernel.

ld -m elf\_i386 -T link.ld -o kernel kasm.o kc.o

**Configurer grub et exécutez notre noyau**

GRUB nécessite que notre noyau soit préfixé de kernel-. Donc, il faut renommer le noyau. Maintenant, placez-le dans le répertoire /boot. Nous aurons besoin des privilèges de superutilisateur. Dans notre fichier de configuration GRUB, grub.cfg nous devons ajouter une entrée:

title myKernel

root (hd0,0)

kernel /boot/kernel-701 ro

Si la directive hiddenmenu existe, il faudra l’enlever.

Pour grub2 (le bootloader par défaut pour les nouvelles distributions), notre config devrait ressembler à ceci:

menuentry 'kernel 701' {

set root='hd0,msdos1'

multiboot /boot/kernel-701 ro

}

Maintenant, il sufit de redémarrer notre ordinateur pour voir une liste avec le nom de notre noyau.

Nous pouvons aussi utilisé qemu pour tester notre kernel.

qemu-system-i386 -kernel kernel