Programmation Système Avancée

Rapport OS

ITI 3

2014 - 2015

Erhardt Markus - Bilgin Huseyin - Graber Vincent

Table des matières



[Etape 1: Bootloader](https://docs.google.com/document/d/1qnF48_PVz7qosy-lK7VnuwpNKetEBz-2MJpyltRVv34/edit" \l "heading=h.9szfwjufjlpz)

[Affectation des rôles](https://docs.google.com/document/d/1qnF48_PVz7qosy-lK7VnuwpNKetEBz-2MJpyltRVv34/edit" \l "heading=h.3f8jb55jdfkl)

[Description de l’étape](https://docs.google.com/document/d/1qnF48_PVz7qosy-lK7VnuwpNKetEBz-2MJpyltRVv34/edit" \l "heading=h.ma80xje5iyvk)

[Description des tâches effectuées](https://docs.google.com/document/d/1qnF48_PVz7qosy-lK7VnuwpNKetEBz-2MJpyltRVv34/edit" \l "heading=h.8zd3fw33vpb7)

[Fichier Makefile](https://docs.google.com/document/d/1qnF48_PVz7qosy-lK7VnuwpNKetEBz-2MJpyltRVv34/edit" \l "heading=h.caud26f46i1a)

[Fichier Bootloader](https://docs.google.com/document/d/1qnF48_PVz7qosy-lK7VnuwpNKetEBz-2MJpyltRVv34/edit" \l "heading=h.trmeyhnv0j80)

[Fichier Kernel](https://docs.google.com/document/d/1qnF48_PVz7qosy-lK7VnuwpNKetEBz-2MJpyltRVv34/edit" \l "heading=h.8r77dxxub85s)

[Etape 2: Appels systèmes + drivers de base](https://docs.google.com/document/d/1qnF48_PVz7qosy-lK7VnuwpNKetEBz-2MJpyltRVv34/edit" \l "heading=h.1m6qcvjkfgcv)

[Etape 3: Système de fichiers](https://docs.google.com/document/d/1qnF48_PVz7qosy-lK7VnuwpNKetEBz-2MJpyltRVv34/edit" \l "heading=h.dev02p1501z9)

[Etape 4: Exécution de processus + applications utilisateur](https://docs.google.com/document/d/1qnF48_PVz7qosy-lK7VnuwpNKetEBz-2MJpyltRVv34/edit" \l "heading=h.ehth6uwmms8c)

[Etape 5: Multi-tâche](https://docs.google.com/document/d/1qnF48_PVz7qosy-lK7VnuwpNKetEBz-2MJpyltRVv34/edit" \l "heading=h.iy86a8f1jre5)

[Etape 6: (Projet à choix)](https://docs.google.com/document/d/1qnF48_PVz7qosy-lK7VnuwpNKetEBz-2MJpyltRVv34/edit" \l "heading=h.axi774b1mg5)

# Etape 1: Bootloader

## **Affectation des rôles**

Markus Erhardt: Présentateur

Huseyin Bilgin: Chef de projet

Vincent Graber: Rédacteur

## **Description de l’étape**

Le but de cette étape est d'implémenter un bootloader qui chargera un mini noyau, le tout à l'aide de l'émulateur QEMU.

Pour ce faire nous devons créer une image d'un disque qui servira de disque de boot.

Cette image doit contenir un secteur de boot suivi du mini kernel.

[Cahier des charges](http://cyberlearn.hes-so.ch/pluginfile.php/584312/mod_resource/content/5/Cahier des charges.pdf)

## **Description des tâches effectuées**

### **Fichier Makefile**

Pour compiler notre bootloader et le kernel, nous utilisons la commande nasm :

    nasm ./boot/boot.asm

    nasm ./kernel/kernel.asm

Nous utilisons 3 commandes pour créer notre disque :

1. dd if=/dev/zero of=disk.img bs=512 count=100
2. dd conv=notrunc seek=0 if=./bootD/boot of=disk.img
3. dd conv=notrunc seek=1 if=./kernelD/kernel of=disk.img

La première commande nous créer un disque de 100 secteurs contenant chacun 512 bytes. Le disque est rempli de zéros.

La deuxième commande met dans le premier secteur du disque notre bootloader.  (seek=0, l’indexation commence à 0)

La dernière commande met dans le secteur numéro 2 notre kernel.

Pour finir, nous lancons l’émulation avec la commande qemu :

    qemu -hda disk.img

### **Fichier Bootloader**

L’instruction suivante nous permet de mettre la suite du code en position 0x7c00 de notre mémoire : [ORG 0x7C00]

MOV AH, 0x0E

MOV BH, 0x00

MOV BL, 0x07

Ces 3 instructions nous permettent de configurer l’interruption d’affichage (N°10).

En effet, le registre AH nous permet de configurer le mode d’affichage, ici en mode teletype.

Le BH = Pagination d’affichage , page 0 dans notre cas

Le BL pour configurer la couleur des caractères, ici 0x07 = gris clair

Nous créons une chaîne de caractères comme cela :

HelloString db 'le bloc est charge', 0;

Un 0 est placé à la fin de la chaîne pour nous permettre faire un test de fin de chaîne.

Nous mettons l’adresse notre chaîne dans le registre SI puis nous appelons notre fonction d’affichage.

MOV SI, HelloString

Call PrintString

On fait une boucle qui parcours chaque caractères de la chaîne et appelle l'interruption 10 pour l’afficher. L'interruption 10 affiche le caractère stocké dans le registre AL.

PrintString:

MOV AL, [SI]            ; Met le contenu du pointeur SI dans registre AL

OR AL, AL            ; Compare si le caractère = zéro

JZ exit\_function        ; Si oui on quitte

Call PrintCharacter            ; Si non, appel impression du charactère à l'écran

INC SI                ; On passe à l'adresse suivante

jmp PrintString        ; On revient au début de la boucle

exit\_function:

RET

PrintCharacter:         ; impression à l'écran

INT 0x10              ; appelle l’interuption 10

RET

Ensuite nous avons notre procédure pour charger notre kernel.

// partie à commenter encore

;--------------------- Load kernel procedure

LoadKern:

       mov ah, 0x02    ; Read Disk Sectors

       mov al, 0x01    ; Read one sector only (512 bytes per sector)

       mov ch, 0x00    ; Track 0

       mov cl, 0x02    ; Sector 2

       mov dh, 0x00    ; Head 0

       mov dl, 0x00    ; Drive 0 (Floppy 1) (This can be replaced with the value in BootDrv)

       mov bx, 0x2000  ; Segment 0x2000

       mov es, bx      ;  again remember segments bust be loaded from non immediate data

       mov bx, 0x0000  ; Start of segment - offset value

readsector:

       int 0x13        ; Call BIOS Read Disk Sectors function

       jc readsector  ; If there was an error, try again

       mov ax, 0x2000  ; Set the data segment register

       mov ds, ax      ;  to point to the kernel location in memory

       jmp 0x2000:0x0000       ; Jump to the kernel

//----------------------------------------------------

TIMES 510 - ($ - $$) DB 0x90 : On remplit la fin du secteur avec des zéros

DW 0xAA55 : les 2 derniers bytes permettent de dire au BIOS qu’il booter sur la partition.

### **Fichier Kernel**

Pour le fichier kernel, nous avons repris ce que nous avions fait dans le bootloader.

Nous avons repris les registre pour l'interruption 10 (AH,BH,BL). On a aussi une chaîne de caractère de la même forme que celle du bootloader et la même fonction qui affiche les caractères.

# Etape 2: Appels systèmes + drivers de base

## **Affectation des rôles**

Markus Erhardt: Chef de projet

Huseyin Bilgin: Rédacteur

Vincent Graber: Présentateur

## **Description de l’étape**

Faire un mini kernel en C afin d’implémenter les appels système.

# **22 octobre 2014 :**

# **Tâches effectuées :**

# Mettre le projet de la première étape sur github.

Déterminer l’adresse de la pile car pendant l’étape précédente

**Notes :**

Sp → Pointeur de pile

Pop, push comme méthodes,

Registre ax contient le retour de la fonction

Pointage est fait avec registres [si:sp]

Les registres sp et bp sont pour intitier la pile.

|  |
| --- |
| adresse Haut |
|  |
|  |
|  |
| basse |

# **27 octobre 2014 :**

# **Tâches effectuées :**

# Création du fichier main.c et kernel.c . Dans la main nous appelons le kernel afin de tester les appels systèmes implémentés.

Création du fichier util\_asm.s pour afin de pouvoir appeler les interruptions depuis un fichier c. Principalement celui-là contient les fonctions fournies par le Prof. comme la fonction \_interrup et \_syscall.

# notes prises le 29 octobre:

syscall\_handler(uint syscall\_nb, uint arg1, uint arg2, uint arg3) la fonction handler traite l’interruption par rapport au numéro de l’appel système et suite au traitement elle appel la fonction correspondante en passant les arguments reçus.

Adressage : AX = AH \* 256 + AL  OU (AH << 8) | AL est fait comme celui-là

donc on applique cela dans le kernel avant les appels systèmes

#define HLR2X(AH, AL)   (AH<<8|AL)

#define registre(AH, AL)   (AH<<8|AL)

Pour une fois on appelle la fonction syscall\_init() qui est définie dans le fichier util\_asm.s, le but de cette fonction est de nous aider faire nos propres appels système en écrasant l’adresse  0x80 pour qu’elle pointe vers nos routines.

Sans cette fonction nous 'navons pas la possibilité d'appeler une interruption logicielle depuis le C.

3 Novembre 2014:

**Tâches effectuées :**

Après mis en place de la fonction interrupt et syscall nous avons appeler une interruption depuis le fichier kernel.c et celle-là est redirigée vers notre handler sys\_call\_handler grâce à la fonction fournie par le Prof.

La fonction print\_char est implémentés est testé.

Mise au point un nouveau Makefile.

La fonction readChar et readString marchent.

Avec ReadSector et WriteSector , nous arrivons a lire et ecrire quelquechoses mais c'est erroné.

# Etape 3: Système de fichiers

# Etape 4: Exécution de processus + applications utilisateur

# Etape 5: Multi-tâche

# Etape 6: (Projet à choix)