Rapport technique
Réalisation d'un Système d'Exploitation : Nonos

I.U.T. de Lens - Département Informatique

Julien Ellart
Jonathan Gricourt
Aurélien Provin
Guillaume Libersat

Année Scolaire 2004-2005

Table des matières

1 Mise en place **5**
1.1 Environnement de développement 5
1.1.1 GCC et le freestanding 5
1.1.2 GAS 5
1.1.3 GNU LD 5
1.1.4 L'automatisation avec GNU Make 6
1.2 Environnement d'exécution 6
1.2.1 La Machine Virtuelle Bochs 6
1.2.2 Création de l'image de boot 7

2 Boot **8**
2.1 GRUB 8

3 Coeur du Noyau **10**
3.1 Architecture générale 10
3.2 Mini Libc 10
3.2.1 Longueur de chaîne 10
3.2.2 Comparaison de chaînes 10
3.2.3 Copie de chaînes 10
3.2.4 Interprétation de chaînes 11
3.2.5 Conversions 12
3.3 Les Tables de descripteurs 12
3.4 La GDT 13
3.4.1 Concept 13
3.4.2 Le sélecteur NULL 13
3.4.3 Les sélecteurs CODE et DATA 14
3.5 Les interruptions 14
3.5.1 Concept 14
3.5.2 Interruptions matérielles 15
3.5.3 Interruptions logicielles 15
3.5.4 Interruptions exceptionnelles 15
3.5.5 Tableau des descripteurs d'interruption (IDT) 16
3.5.6 Les Routines de gestion (Handlers) 17
3.5.7 Interruptions désactivables 17
3.6 La mémoire 17
3.6.1 Principe 17
3.6.2 Implémentation 18
3.6.3 Initialisation 19

3.6.4 Allocation . . . 19
3.6.5 Désallocation . . . 20
3.6.6 Les autres fonctions . . . 20
3.6.7 Quelques calculs . . . 20

4 Pilotes du Noyau **21**
4.1 Le pilote IDE . . . 21
4.1.1 Principe . . . 21
4.1.2 Rôle . . . 21
4.1.3 Les ports . . . 21
4.1.4 Programmation . . . 22
4.1.4.1 Les structures de données . . . 22
4.1.4.2 Les fonctions . . . 23
4.2 Le pilote clavier . . . 24
4.3 Le terminal . . . 24
4.3.1 Le clavier . . . 25
4.3.2 L'affichage . . . 25
4.4 Le pilote vidéo . . . 25
4.4.1 Définition . . . 25
4.4.2 Principe . . . 25
4.4.3 Programmation . . . 25
4.5 Le pilote Bochs . . . 27

5 Le système de fichiers **28**
5.1 Généralités . . . 28
5.1.1 Définition et objectifs d'un système de fichiers . . . 28
5.1.2 Structure de NonFS . . . 28
5.1.3 Unités d'allocation . . . 29
5.1.4 La taille des blocs . . . 29
5.2 Espaces réservés . . . 29
5.2.1 Super bloc . . . 29
5.2.2 Les descripteurs de groupe . . . 29
5.2.3 Les Inodes . . . 29
5.2.4 La table d'inodes . . . 31
5.2.5 Les bitmaps . . . 31
5.2.6 Les « directories » . . . 31
5.2.7 Le formatage d'un disque . . . 32
5.2.8 Montage du système de fichier . . . 32
5.2.9 Les fonctions d'entrées sorties . . . 32
5.2.10 Les autres fonctions . . . 33
5.2.11 La fragmentation . . . 33

A Listings **34**
A.1 Fichier de configuration de Bochs . . . 34
A.2 Création d'une image disquette . . . 35

Table des figures

2.1 Structure des informations multiboot 9

3.1 Architecture générale de Nonos 11
3.2 Un segment x86 13
3.3 Illustration des DPL 13
3.4 Mécanisme d'interruption 14
3.5 Les PICs i8259A 15
3.6 Agencement de l'IDT 16
3.7 Liste circulaire doublement chaînée 18
3.8 Fonctionnement de la mémoire physique 19

4.1 Structure ide_device_t 23
4.2 Structure ide_controller_t 23
4.3 Fonctions d'initialisation du driver IDE 23
4.4 Fonctions d'opérations du driver IDE 24
4.5 Fonctionnement du driver clavier 24
4.6 Mémoire vidéo 26

5.1 Structure du système de fichiers et d'un groupe de blocs 28
5.2 Structure d'un super bloc 30
5.3 Structure d'un descripteur de groupe 30
5.4 Structure d'un inode 30
5.5 Blocs d'indirections 31
5.6 Structure d'un « directory » 32
5.7 Correspondance entre inodes 32

Liste des tableaux

2.1 Informations Multiboot 8

3.1 Liste des IRQs 16

4.1 Adresses des registres des contrôleurs 21
4.2 Registres de commandes en lecture et écriture 22

Chapitre 1

Mise en place

1.1 Environnement de développement

Pour la construction de nos composants, nous avons choisi la chaîne de compilation GNU[1].

1.1.1 GCC et le freestanding

Le compilateur GCC[2] offre un mode appelé le « freestanding » qui permet de compiler un programme en enlevant toutes les dépendances à une libc par exemple. Le programme, une fois compilé, est autonome. C'est donc ce qu'il nous fallait.

Bien entendu, ne pouvant se reposer sur aucune librairie, nous disposons d'un ensemble réduit de fonctions. Quatre fonctions sont accessibles lors de l'utilisation de ce mode :

- memcpy : *pour copier la mémoire* ;
- memcmp : *pour comparer la mémoire* ;
- memmove : *pour déplacer la mémoire* ;
- memset : *pour remplir la mémoire.*

Nous devrons donc réécrire un petite libc pour le noyau. Il faut noter aussi qu'il serait intéressant de réécrire ces quatre fonctions disponibles, car elles ne sont pas du tout optimisées.

1.1.2 GAS

Pour ce qui est de la compilation de l'assembleur x86, ce sera GAS qui sera utilisé. Il sera automatiquement appelé par GCC.

1.1.3 GNU LD

Enfin, pour ce qui est de l'assemblage des objets produits, l'éditeur de liens GNU sera lui aussi utilisé.

[1] GNU est un acronyme récursif de « Gnu's Not Unix ».
[2] GCC signifie « GNU C Compiler ».

Afin de permettre le boot du noyau, certains flags comme « -Ttext 0x100000 -e start » seront indiqués afin d'expliciter l'adresse de début du noyau et le symbole de la fonction d'amorcage.

1.1.4 L'automatisation avec GNU Make

Comme nous devrons compiler souvent nos programmes, nous avons décidé d'utiliser le programme GNU make. Nous essaierons de maintenir des Makefiles récursifs afin de scinder et clarifier les étapes de compilation.

1.2 Environnement d'exécution

Afin de pouvoir tester aisément le noyau compilé, nous avons fait le choix d'utiliser une machine virtuelle. En effet, grâce à ce genre d'outils, il est beaucoup plus simple de tester et de traquer les problèmes.

D'une part, pas besoin de redémarrer la machine à chaque test (gain de temps important), et il est possible de tester le noyau x86 sur une machine d'une autre architecture (une personne du groupe travaille sur PowerPC).

D'autre part, le débogage est bien plus simple : possibilité de suivre les registres processeurs, suivi des erreurs fatales avec GDB[3], etc.

Deux candidats ont été retenus :
- Bochs ;
- QEmu.

Après plusieurs tests et retours d'expériences, Bochs est apparu être le meilleur choix.

1.2.1 La Machine Virtuelle Bochs

La VM[4] Bochs est un émulateur d'ordinateur IA-32 capable de fonctionner sur de nombreuses architectures et divers systèmes d'exploitation.

Bochs intègre l'émulation d'un CPU Intel x86, d'un BIOS[5], et de périphériques classiques comme une carte vidéo compatible VESA[6], des ports séries, etc.

Il est capable de travailler sur des disques de données images : on peut donc créer des images de disques durs ou de disquettes.

Une autre particularité est sa capacité à se brancher sur l'un des ports parallèles et de dévier les informations vers le terminal depuis lequel la VM est exécutée. Pour l'inscription des messages de débug, c'est on ne peut plus pratique !

Nous avons donc configuré la VM afin qu'elle boot depuis une disquette virtuelle, qui est en réalité une image disque contenant le noyau.

Notre configuration de Bochs est fournie en annexe A.1 page 34.

[3]GDB est le débogueur de la suite GNU CC.
[4]Abréviation de « Virtual Machine », *soit « Machine Virtuelle »*
[5]BIOS signifie « Basic Input Ouput System ».
[6]VESA est une norme sur les modes graphiques.

1.2.2 Création de l'image de boot

C'est une image de type disquette qui a été retenue pour l'instant. Nous évoluerons vers une image de type disque dur dès que le driver IDE sera fonctionnel.

L'image est créée selon 7 étapes :

1. Création d'un fichier de longueur 1,44Mo rempli de « zéros » ;
2. Création d'un système de fichiers sur l'image ;
3. Écriture d'un boot-loader sur l'image ;
4. Montage de l'image ;
5. Rédaction du fichier de configuration du boot-loader ;
6. Copie du fichier noyau ;
7. Démontage.

Le listing d'un exemple de création d'une image sous GNU/Linux est situé en annexe A.2 page 35.

Chapitre 2

Boot

Afin de démarrer le système d'exploitation, il est indispensable d'avoir un boot-loader[1].

Nous aurions pu le programmer aussi, cependant, nous avons préféré nous focaliser sur le développement du noyau. C'est pourquoi, nous avons choisi d'utiliser GRUB[2], qui a la particularité de permettre de charger facilement des images ELF[3].

2.1 GRUB

Grub est un boot-loader générique. Avec lui, nous pouvons charger notre noyau avec une simple entrée dans le fichier de configuration.

Il utilise un standard appelé « multiboot » qui va charger des informations à un emplacement défini qui pourront être utilisées par le noyau.

Les informations insérées sont recensées dans la table 2.1.

Offset	Type	Nom du Champ	Note
0	u32	magic	Toujours présent
4	u32	flags	Toujours présent
8	u32	checksum	Toujours présent
12	u32	header_addr	si flags[16] est positionné
16	u32	load_addr	si flags[16] est positionné
20	u32	load_end_addr	si flags[16] est positionné
24	u32	bss_end_addr	si flags[16] est positionné
28	u32	entry_addr	si flags[16] est positionné
32	u32	mode_type	si flags[2] est positionné
36	u32	width	si flags[2] est positionné
40	u32	height	si flags[2] est positionné
44	u32	depth	si flags[2] est positionné

TAB. 2.1 – Informations Multiboot

[1]en français, « Chargeur de démarrage ».

[2]Abréviation de « Grand Unified Bootloader »

[3]ELF signifie « Executable and Linking Format ». Il s'agit d'un format binaire pratique et évolué.

La description précise de ces champs se trouve sur la page du standard multiboot[4].

Ainsi nous pourrons déclarer une structure dans le noyau afin de la mapper sur les entrées multiboot ce qui nous permettra de disposer de précieuses informations sur l'état de la machine avant le démarrage du noyau.

Le multiboot nous fournit aussi des informations post-boot, qui sont utiles à l'initialisation des composants logiciels. La structure de ces informations est référencée dans la table 2.1.

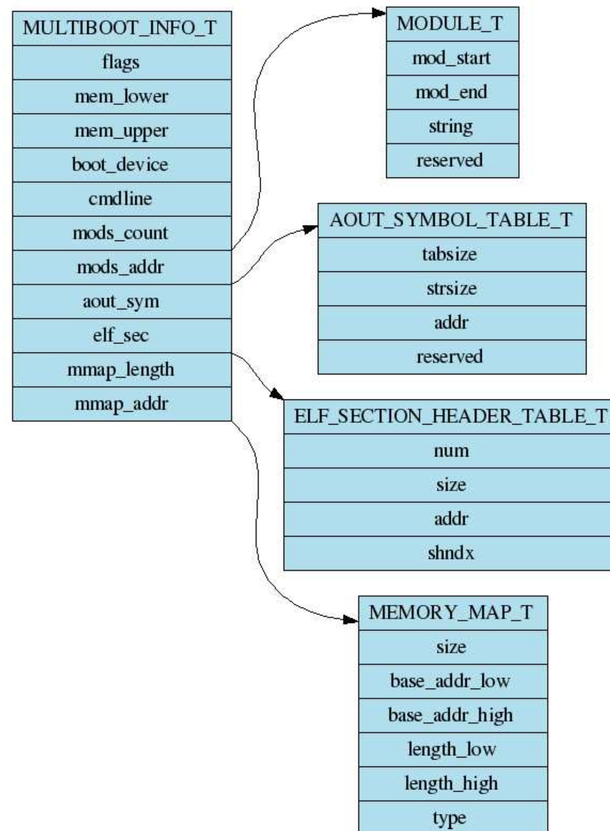

FIG. 2.1 – Structure des informations multiboot

[4]http ://www.gnu.org/software/grub/manual/multiboot/multiboot.html

Chapitre 3

Coeur du Noyau

3.1 Architecture générale

Nous avons décidé de ne nous baser sur aucun modèle d'architecture existant. Nous avons avancé dans le projet en essayant de développer l'architecture la plus logique et pratique pour nous. La figure 3.1, qui se situe à la page 11, présente notre modèle.

3.2 Mini Libc

Le fichier *string.c* contient une partie des fonctions essentielles au développement de l'OS.

3.2.1 Longueur de chaîne

La fonction ***k_strlen*** calcule la taille d'une chaîne de caractères et en retourne la taille.
*int k_strlen(const char *str)*

3.2.2 Comparaison de chaînes

La fonction ***k_strcmp*** compare deux chaînes de caractères et retourne la valeur 1 si $s1 > s2$, 0 si $s1 = s2$ ou -1 si $s1 < s2$.
*int k_strcmp(const char *s1, const char *s2)*

3.2.3 Copie de chaînes

La fonction ***k_strcpy*** copie une chaîne de caractères pointée par *src* dans *dst* ('\0' compris) et retourne un pointeur sur le début de la chaîne *dst*.
char k_strcpy(char *dst, const char *src)*

Cette fonction copie les *n* premiers caractères de la chaîne de caractères pointée par *src* dans *dst* (s'il n'y a pas de '\0' dans *src*, il n'apparaitra pas dans *dst*). Elle retourne un pointeur sur le début de la chaîne *dst*.
*char *k_strncpy(char *dst, const char *src, int n)*

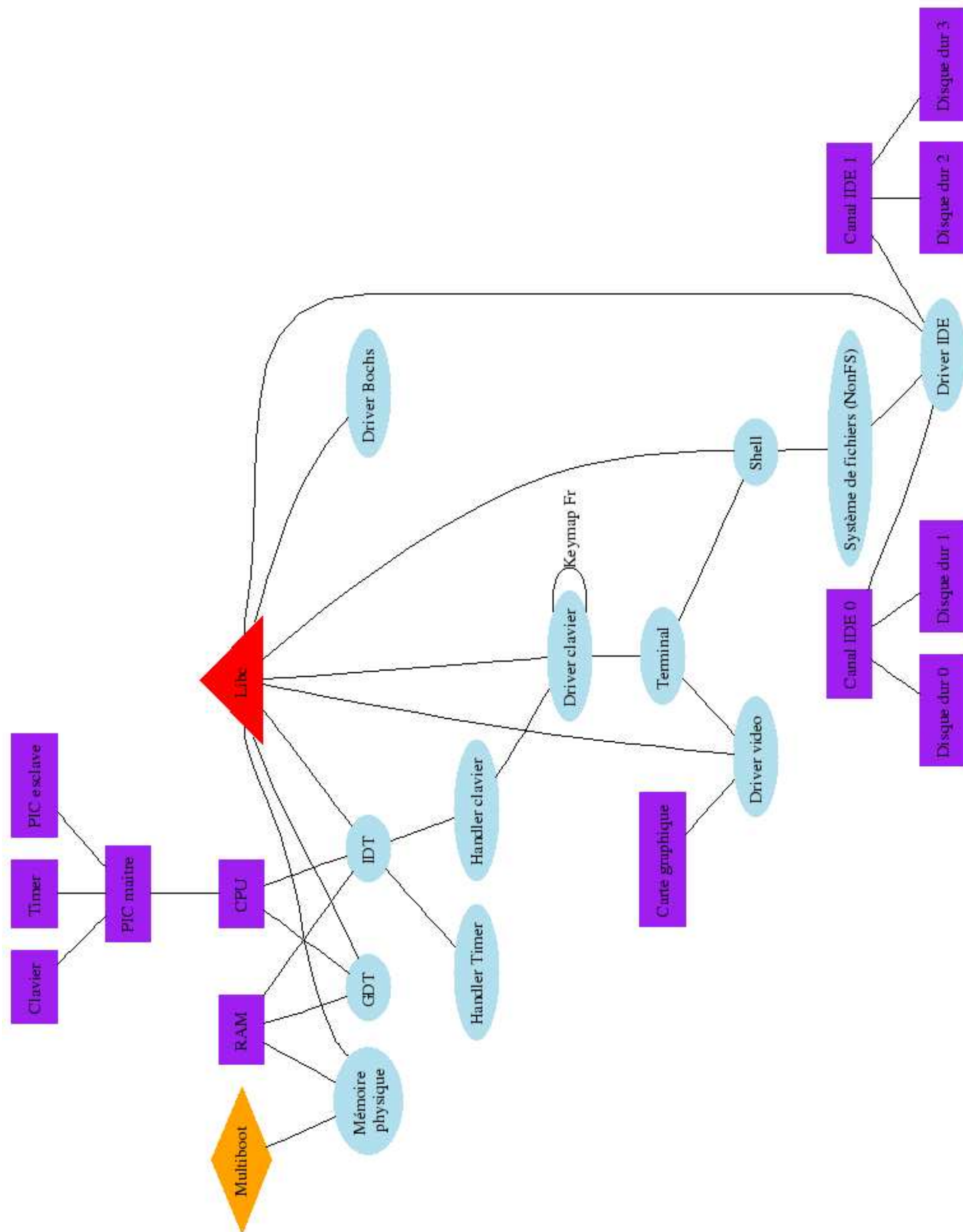

FIG. 3.1 – Architecture générale de Nonos

3.2.4 Interprétation de chaînes

La fonction ***k_ sprintf*** produit une sortie de *format* dans *buf*. *format* indique les conversions à effectuer sur les variables stockées dans une **va_ list**.

Les balises prises en compte sont « %d », « %s » et « %c ».

k_ sprintf retourne le nombre de caractères écrits dans *buf*. Voir l'exemple

de code numéro 1.

Algorithm 1 Exemple avec k_sprintf

```
char c = 'a';
int entier = 1;
char *s = "caractère";
k_sprintf("il y %c %d %s\n", c, entier, s);
```

résultat : "il y a 1 caractère\n"

*int k_sprintf(char *buf, const char *format, ...)*

3.2.5 Conversions

La fonction ***k_itoa*** transforme un entier *nb* en chaîne de caractères en écrivant dans *buf*. Les entiers signés et non signés sont pris en compte. Cette fonction retourne le nombre de caractères écrits dans *buf*.

*int k_itoa(char *buf, int nb)*

3.3 Les Tables de descripteurs

Sur l'architecture x86, il existe trois descripteurs principaux :

- la GDT : « Global Descriptor Table » ;
- la LDT : « Local Descriptor Table » ;
- l'IDT : « Interruption Descriptor Table ».

Pour définir la GDT et les LDT, on utilise des segments. Leur structure est extraite de la documentation officielle INTEL (Volume 3, « System Programming Guide »).

Un segment mesure 64 bits et est défini comme indiqué sur la figure 3.2 page suivante. Comme on le remarque, il doit être aligné sur 8 bits.

Les segments disposent d'un champ important, appelé DPL (Descriptor Privilege Level), qui indique à quel espace appartient celui-ci. Un segment ayant un DPL faible peut librement accéder aux supérieurs, mais l'inverse est interdit, et déclenchera une exception.

Il existe quatre niveaux de privilèges, mais nous n'utiliserons que deux niveaux : 0 et 3.

0 correspond à l'espace noyau, tandis que 3 correspondra à l'espace utilisateur.

On appelle ces espaces des « rings », comme présenté sur la figure 3.3 page suivante.

```
struct segment_descriptor {
uint16_t segment_limit_0; /* Segment limit 0->15 */
uint16_t base_addr_0; /* Base address 0->15 */
uint8_t base_addr_1; /* Base adresse 16->23 */
uint8_t segment_type :4;
uint8_t descr_type :1; /* Descriptor Type : 0 = system; 1 = code or data */
uint8_t descr_priv_level :2; /* Descriptor Privilege Level */
uint8_t segment_present :1;
uint8_t segment_limit_1 :4; /* Segment Limit 16->19 */
uint8_t system :1; /* Available custom bit for system */
uint8_t blank :1; /* Blank bit (uh?) */
uint8_t operation_size :1; /* 0 = 16bits segment, 1 = 32bits segment */
uint8_t granularity :1; /* Granularity of segment limit field. 0 = byte, 1 = 4Kbytes unit */
uint8_t base_addr_2 :5; /* Base address 24->31 */
} __attribute__ ((packed, aligned(8)));
```

FIG. 3.2 – Un segment x86

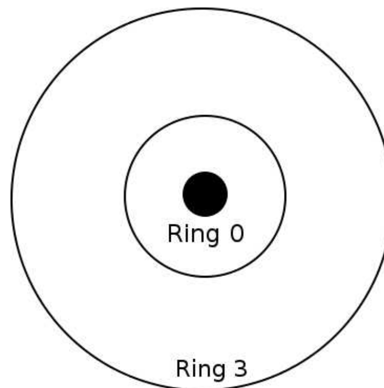

FIG. 3.3 – Illustration des DPL

3.4 La GDT

3.4.1 Concept

La GDT, ou table de descripteurs globale, est une table qui contient des segments et qui permet de définir le type de mémoire que l'on souhaite mettre en place (segmentation) et de définir les droits d'accès à ceux-ci.

Elle est composée de trois segments au minimum :

1. Un sélecteur NULL ;
2. Un sélecteur CODE ;
3. Un sélecteur DATA.

3.4.2 Le sélecteur NULL

Le sélecteur NULL n'a aucune utilité pour le noyau, mais, il est obligatoire car spécifié dans la documentation INTEL.

Selon cette dernière, il est nécessaire pour que le processeur reconnaisse un segment NULL comme invalide.

En pratique, il semble que l'on peut faire ce que l'on veut de ce segment, et qu'il n'est pas vraiment utile.

Comme son nom l'indique, pour le déclarer, il suffit de le remplir de zéros.

3.4.3 Les sélecteurs CODE et DATA

Ces deux sélecteurs permettent de définir à quels privilèges les programmes exécutés pourront utiliser la mémoire. On définit un DPL particulier pour ces segments, protégeant et/ou autorisant ceux-ci à écrire ou lire.

Ces deux sélecteurs sont quasiment identiques, à l'exception d'un champ qui les différencie entre lecture (DATA) et exécution (CODE).

3.5 Les interruptions

3.5.1 Concept

Même s'il en donne l'illusion, un processeur ne peut traiter qu'une tâche à la fois. C'est pourquoi, on va simuler le comportement d'une machine capable de traiter différentes choses au même moment avec les interruptions.

Lors du traitement d'une tâche, le processeur peut être interrompu par une interruption afin de temporairement suspendre la tâche en cours et répondre au demandeur de celle-ci. Une fois l'interruption traitée, le processeur peut reprendre la tâche suspendue (Figure 3.4).

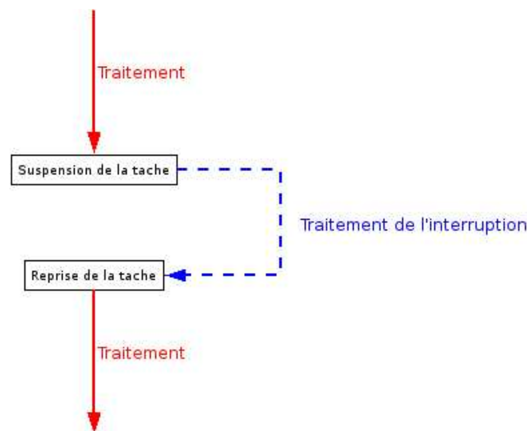

FIG. 3.4 – Mécanisme d'interruption

Une interruption peut avoir trois origines différentes (en réalité plus, mais seules ces trois sont importantes) :

- Matérielle
- Logicielle
- Exceptionnelle

On les compte au nombre maximum de 256.

3.5.2 Interruptions matérielles

Les interruptions matérielles sont générées par les composants de l'ordinateur. Par exemple, il peut s'agir du clavier.

Au niveau matériel, les interruptions sont liées au concept d'IRQ[1]. L'IRQ est un nombre compris entre 0 et 16 associé à un composant matériel.

Sur IA32, les IRQs sont contrôlées à travers deux PICs[2] montés en cascade. Ces PICs sont des i8259A et gèrent chacun 8 IRQs. Le PIC maître gère les interruptions entre 0 et 7, tandis que le second (esclave) gère celles entre 8 et 15.

A l'origine, seul un PIC était utilisé (bus ISA sur 8bits). Un second a ensuite été ajouté, relié à travers l'IRQ 2 du maître et l'IRQ 7 de l'esclave, d'où le terme de « cascade » (Figure 3.5).

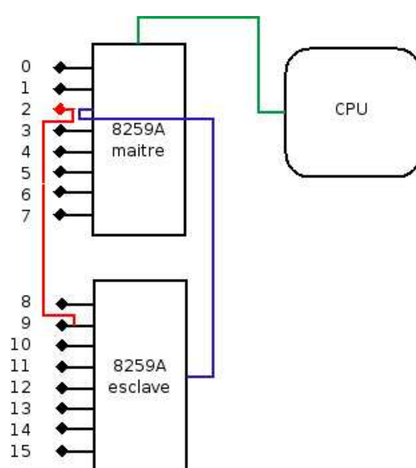

FIG. 3.5 – Les PICs i8259A

La liste des IRQ est recensée dans le tableau 3.1 page suivante.

On peut remarquer que deux périphériques peuvent avoir le même IRQ : c'est possible si les deux périphériques ne travaillent pas en même temps.

3.5.3 Interruptions logicielles

Ces interruptions sont des interruptions déclenchées par des programmes. Un programme peut par exemple demander une interruption afin d'exécuter une fonction système. Dans ce cas, le processeur va s'occuper de lui en priorité.

3.5.4 Interruptions exceptionnelles

Ce type d'interruption est envoyé par le processeur. Lorsqu'un problème se pose dans le traitement d'un programme, une exception sera lancée. Libre au

[1] Interruption ReQuest

[2] PIC : Programmable Interrupt Controler

IRQ	Périphérique
0	Horloge interne
1	Clavier
2	Vers PIC esclave (IRQ 8 à 15)
3	COM2/COM4
4	COM1/COM3
5	libre
6	Contrôleur de disquettes
7	LPT1
8	Horloge temps réel (CMOS)
9	libre
10	libre
11	libre
12	PS2/libre
13	Coprocesseur mathématique
14	Contrôleur IDE primaire
15	Contrôleur IDE secondaire

TAB. 3.1 – Liste des IRQs

programmeur de réagir comme il le souhaite (arrêt du système d'exploitation, affichage de l'erreur, etc).

3.5.5 Tableau des descripteurs d'interruption (IDT)

Afin de permettre l'utilisation des interruptions, toutes les correspondances sont stockées dans une grande table, appelée IDT. Cette table contient uniquement des vecteurs d'interruption.

Un vecteur d'interruption est une correspondance entre un code d'interruption et sa routine de gestion. Ces vecteurs sont codés sur 32bits, soit 4 octets.

L'IDT étant capable de stocker jusqu'à 256 vecteurs, elle fait donc 1ko.

Cette table est reprogrammable à volonté.

Quand la machine démarre, pour garder la compatibilité avec les anciens processeurs (< 386), l'adresse de base du PIC maître est 0x8 et celle du PIC esclave 0x70. Toutes les exceptions ne peuvent donc pas être gérées (on compte 32 exceptions actuellement).

On va donc remapper le PIC maître sur 0x20 et le PIC esclave sur 0x28 afin de laisser 0x20 (=32) espaces libres pour les exceptions.

On aura donc une table organisée comme décrit sur la figure 3.6.

Table IDT			
Exceptions	PIC1	PIC2	...
0x0 à 0x19	0x20 à 0x27	0x28 à 0x35	0x36 à 0x100

FIG. 3.6 – Agencement de l'IDT

Pour connaître le numéro d'interruption d'un matériel, il suffit donc d'utiliser à présent la formule :

VECTEUR_INTERRUPTION = BASE_VECTEUR_INTERRUPTION_PIC + IRQ

Par exemple, pour le clavier (IRQ 1), on fera 0x20 + 1. On sait donc que l'interruption du clavier est 0x21 (soit le vingt et unième vecteur).

3.5.6 Les Routines de gestion (Handlers)

Comme dit précédemment, chaque interruption doit (en théorie) être reliée à une routine de gestion au travers d'un vecteur. Cette routine va effectuer le travail correspondant. Par exemple, si le clavier interrompt le processeur, on va récupérer la touche pressée et traiter l'information.

Une fois le travail nécessaire effectué, il est indispensable de l'indiquer. Cela est réalisé avec un EOI[3]. S'il s'agit d'une interruption inférieure à 8 (donc venant du PIC maître), seul le PIC maître devra en être informé. Dans le cas contraire, le PIC secondaire puis le PIC maître devront être informés de la fin de l'handler.

Chacune des routines devra se terminer par un « iret ».

3.5.7 Interruptions désactivables

Il existe deux types d'interruptions matérielles : les MI[4] et les NMI[5]. Le premier se nomme ainsi car il est possible de les désactiver aisément, tandis que la seconde est théoriquement non désactivable.

Il peut être intéressant de désactiver les interruptions afin par exemple de préparer l'IDT, et de ne pas être gêné.

Les MI se désactivent via l'instruction « cli » et se réactivent via « sti ».

Les NMI peuvent en réalité être aussi désactivées, mais pour cela il faut passer par le bus ISA en envoyant la valeur 0x80.

3.6 La mémoire

3.6.1 Principe

La gestion de la mémoire est essentielle dans tout SE. En effet, sans gestion de la mémoire, il n'est pas possible d'allouer de la mémoire dynamiquement et seule l'allocation statique est possible. Avec cette allocation statique, les possibilités s'avèrent très limitées : impossibilité de créer des processus, de stocker les données nécessaires à la gestion du système de fichiers, ...

Le principe de la gestion dynamique de la mémoire est assez simple. Il suffit de connaître les zones mémoires occupées et les zones libres. Ensuite, des fonctions permettent d'allouer et de libérer ces zones.

[3]End Of Interrupt
[4]Maskable Interrupt
[5]Non Maskable Interrupt

3.6.2 Implémentation

Pour sauvegarder les zones mémoires occupées et les zones libres, nous avons choisi d'utiliser des listes chaînées.

Pour faciliter les échanges entre les listes, nous avons utilisé des listes doublement chaînées circulaires. Ces listes facilitent le parcours (grâce au double chaînage) et elles facilitent aussi l'ajout dans la liste (grâce à la connaissance de la tête et de la queue de la liste).

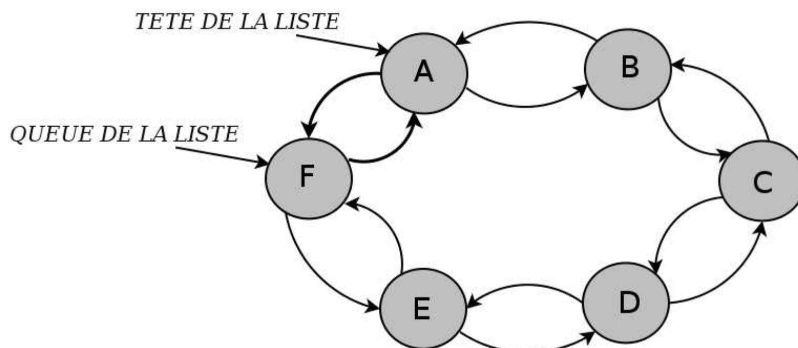

FIG. 3.7 – Liste circulaire doublement chaînée

Nous n'avons pas réécrit les différentes fonctions de manipulation des listes chaînées car cela ne nous apprendrait rien (l'implémentation des listes a déjà été vu en cours).

Dans notre SE, nous avons choisi de découper la mémoire en zones de 4096 octets (pour pouvoir facilement gérer la pagination). Ces zones de 4096 octets sont appelées des pages physiques. Chacune d'entre elles est associée à une structure appelée descripteur de pages physiques.

Cette structure contient bien évidemment l'adresse de la page physique qu'il décrit, mais elle contient aussi les adresses des pages immédiatement précédente et suivante. Elle contient également deux champs supplémentaires. Le premier permet de sauver l'état de la page (si elle est allouée ou non) et le second indique la position dans une allocation (utilisé pour l'allocation de plusieurs pages).

Deux listes chaînées sont donc présentes : une liste contenant la totalité des pages physiques libres et une liste pour les pages utilisées.

Ces deux listes représentent ainsi la totalité des pages physiques de la mémoire et sont stockées dans un tableau appelé « tableau des descripteurs de pages physiques ».

Le compteur de référence, dont il est question auparavant, est un simple entier mis à jour par les fonctions d'allocation et de libération des pages de mémoire. Lorsqu'il est à 1, cela signifie que la page est allouée et donc, le descripteur de page se trouve dans la liste des pages utilisées.

Le champ restant dans la structure d'un descripteur permet d'allouer plusieurs pages et il représente la position de la page dans un bloc de pages.

Lorsqu'il est à 1, cela veut dire que la page est le premier élément d'un bloc.

La figure 3.8 page suivante résume le fonctionnement global de la gestion de la mémoire.

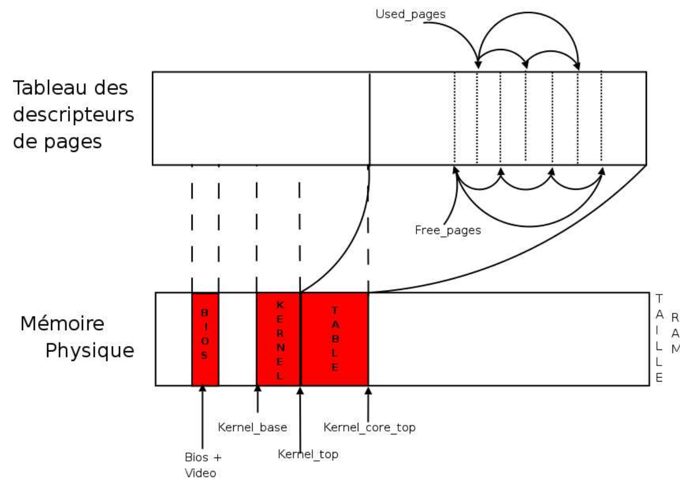

FIG. 3.8 – Fonctionnement de la mémoire physique

3.6.3 Initialisation

L'initialisation est effectuée assez simplement.

D'abord, on récupère la taille de la mémoire (passée en paramètre de la fonction). Puis on parcours la totalité des pages en initialisant chaque descripteur de page et en les mettant dans la liste des pages libres ou dans la liste des pages utilisées.

Le parcours des pages s'effectue à partir du *kernel_ core_ top* (première page libre au-dessus du noyau).

Le comportement associé à une page libre consiste à initialiser le compteur de références à 0 et à l'ajouter dans la liste des pages libres.

À la fin de cette fonction, l'initialisation est terminée. On trouve dans la liste des pages libres toutes les pages de la mémoire (à partir du *kernel_ core_ top*). En effet, la zone qui s'étend du début de la mémoire jusqu'au *kernel_ core top* a été ignorée pour simplifier.

3.6.4 Allocation

La fonction qui permet d'allouer de la mémoire est la fonction ***k_ malloc*** qui prend en paramètre le nombre d'octets souhaité.

Cette fonction fait appel à une fonction (***contigous_ list***) qui sert à obtenir un bloc de mémoire contigu et suffisamment grand pour pouvoir faire l'allocation.

Une fois cette zone récupérée, on fait appel à une fonction de référencement de page. Pour chacune des pages, le compteur de référencement est mis à 1, et le descripteur de page est mis dans la liste des pages utilisées.

Une fois que toutes les pages sont référencées, la fonction renvoie l'adresse de la première page allouée.

3.6.5 Désallocation

La fonction qui permet de libérer de la mémoire est la fonction ***k_free*** qui prend en paramètre l'adresse de la première page à libérer.

Cette fonction récupère toutes les pages qu'il faut libérer (on le sait grâce au *ref_id*).

Ensuite pour chaque page, on place le compteur de référence à 0, et on déplace le descripteur vers la liste des pages libres.

Lorsque toutes les pages ont été libérées, la fonction se termine.

3.6.6 Les autres fonctions

D'autres fonctions sont biensur présentes pour effectuer les différentes tâches.

Il y a tout d'abord la fonction ***list_get_nieme*** qui prend en paramètre une liste et une page physique et qui effectue une recherche de cette page dans la liste.

Une fois la page trouvée, elle est renvoyée.

La fonction ***get_free_page_after*** effectue une recherche de la position de la page à insérer. Ceci est nécessaire pour garder la liste triée.

La fonction ***find_page*** s'occupe de trouver, dans une liste, la page dont l'adresse est égale à l'adresse passée en paramètre.

Enfin, la fonction ***k_phys_mem_ref_page_new*** s'occupe de référencer une page. Son principe est simple : la page passée en paramètre est supprimée de la liste des pages libres, le compteur de référence est mis à 1, et la page est mise dans la liste des pages utilisées.

3.6.7 Quelques calculs

Voici quelques formules qu'il est nécessaire de détailler.

Tout d'abord, une page est égale à 4096 octets, ce qui signifie que les adresses de deux pages successives seront séparées par un pas de 0x1000. Par exemple, la page qui suivra 0x150000 sera 0x151000.

Deux macros ont été définis pour arrondir les adresses à l'adresse de la page directement supérieure ou inférieure. Par exemple, pour l'adresse 0x1234, ***NONOS_PAGE_ALIGN_SUP()*** renverra 0x2000 et ***NONOS_PAGE_ALIGN_INF()*** renverra 0x1000.

Pour connaître l'adresse de fin du tableau de descripteur, il suffit de faire :
NONOS_PAGE_ALIGN_SUP(kernel_top) + ***NONOS_PAGE_ALIGN_SUP***((size_mem > > 12) * ***sizeof***(struct phys_page)) ;

Chapitre 4

Pilotes du Noyau

4.1 Le pilote IDE

4.1.1 Principe

Pour programmer un pilote de périphérique IDE, il faut en fait programmer le contrôleur associé.

Ce contrôleur contient des registres où l'on trouve des informations sur les périphériques et d'autres où l'on doit indiquer les opérations à réaliser.

On trouve généralement deux contrôleurs IDE sur une carte mère (certaines en comptent quatre), chaque contrôleur pouvant gérer deux périphériques.

4.1.2 Rôle

Le rôle du pilote IDE est de permettre d'effectuer des tâches basiques : détecter et initialiser le périphérique, lire et écrire à un emplacement spécifié (secteurs du disque).

Il n'est pas responsable de la façon dont sont répartis les fichiers sur le disque ; il se contente de déplacer des données d'un secteur à la mémoire ou de la mémoire à un secteur.

4.1.3 Les ports

Adresse	Description
1F0 +X	Registres de commande du 1er contrôleur
3F0 + Y	Registres de contrôle du 1er contrôleur
170 + X	Registres de commande du 2ème contrôleur
370 + Y	Registres de contrôle du 2ème contrôleur
F0 + X	Registres de commande du 3e contrôleur
2F0 + Y	Registres de contrôle du 3e contrôleur
70 + X	Registres de commande du 4e contrôleur
270 + Y	Registres de contrôle du 4e contrôleur

Tab. 4.1 – Adresses des registres des contrôleurs

X	Signification
0	Données
1	Registre d'erreur
2	Nombre de secteurs
3	Secteur
4	Cylindre inférieur
5	Cylindre supérieur
6	Lecteur et tête
7	État

X	Signification
0	Données
1	Précompensation d'écriture
2	Nombre de secteurs
3	Secteur
4	Cylindre inférieur
5	Cylindre supérieur
6	Lecteur et tête
7	Commande

TAB. 4.2 – Registres de commandes en lecture et écriture

Ainsi, si on veut, par exemple, lire des données sur le premier disque du premier contrôleur, il faudra remplir le registre de commande correspondant avec les informations sur l'emplacement (nombre de secteurs, premier secteur, cylindre, lecteur) et la commande à effectuer . Le port « données » contiendra alors les données lues.

Algorithm 2 Exemple simplifié de lecture

```
outb( 0 , 0x1F1 ) ; // précompensation
outb( 1 , 0x1F2 ) ; // 1 secteur à lire
outb( num_secteur , 0x1F3 ) ; //A partir du secteur num_secteur
outb(cyl_lo , 0x1F4) ; // Numéro du cylindre inférieur
outb(cyl_hi , 0x1F5) ; // Numéro du cylindre supérieur
outb(head , 0x1F6) ; // Numéro de la tête
outb(IDE_CMD_READ , 0x1F7) // Envoie de la commande de lecture
for(i = 0 ; i < 256 ; i++)
{
buffer[i] = inw(0x1F0) ;
}
```

4.1.4 Programmation

4.1.4.1 Les structures de données

La structure *ide_ device* représente un disque dur avec toutes les informations nécessaires sur la nature et la structure du disque. On trouve dans cette structure un pointeur sur le contrôleur auquel le disque dur est relié.

```
typedef struct ide_device {
uint id;
uint status;
uint16_t cylinder;
uint16_t head;
uint16_t sector_per_track;
uint16_t block_count;
size_t block_size;
int lba;
struct ide_controller *ctrl;
} ide_device_t;
```

FIG. 4.1 – Structure ide_device_t

Un contrôleur est représenté par une structure contenant les informations de celui-ci (id, adresse, état) ainsi qu'un tableau de périphériques.

```
typedef struct ide_controller {
uint id;
uint16_t addr;
uint status;
struct ide_device devices[2];
} ide_controller_t;
```

FIG. 4.2 – Structure ide_controller_t

4.1.4.2 Les fonctions

Le pilote IDE peut être divisé en deux parties :

- La partie initialisation : elle est exécutée au démarrage du SE. Son rôle est de détecter les contrôleurs et les disques durs, de récupérer les informations nécessaires sur ces disques.
- La partie opération : celle-ci est exécutée à chaque opération de lecture / écriture commandée par le système de fichiers. À partir d'un pointeur sur le disque dur concerné, le pilote effectue une lecture / écriture à l'emplacement désiré.

```
result_t init_ide(ide_controller_t *ctrl);
result_t ide_init_controller(ide_controller_t *ctrl);
result_t ide_device_info(ide_device_t *dev);
```

FIG. 4.3 – Fonctions d'initialisation du driver IDE

La fonction ***init_ide*** est appelée par le système lors du démarrage. Elle se charge d'initialiser la structure de type *ide_controller*. Elle appelle la fonction ***ide_init_controller*** qui vérifie l'existence du contrôleur et détecte les disques durs présents sur le contrôleur. Pour chaque disque dur détecté, on appelle la fonction ***ide_device_info*** qui récupère des informations sur le disque.

```
result_t ide_operation(ide_device_t *dev, int operation,uint32_t block_start,uint16_t *buffer);
result_t ide_read(ide_device_t *dev,uint32_t block_start,uint16_t block_count,uint16_t *buffer);
result_t ide_write(ide_device_t *dev,uint32_t block_start,uint16_t block_count,uint16_t *buffer);
```

FIG. 4.4 – Fonctions d'opérations du driver IDE

La fonction ***ide_ operation*** permet d'effectuer au choix une lecture ou une écriture sur un unique secteur. Le code pour effectuer une opération de lecture ou une opération d'écriture est semblable, seule la commande à envoyer change.

ide_ read et ***ide_ write*** permettent de lire ou écrire un nombre *block_ count* de secteurs à partir du secteur *block_ start*.

4.2 Le pilote clavier

Le pilote clavier permet de recevoir une demande de lecture du clavier, de lire les données, puis de les envoyer à la couche supérieure.

Ce pilote est lié à une interruption, qui déclenche la lecture d'un caractère.

Il est en lui-même très simple car il n'effectue aucun traitement sur les données.

La figure 4.5 décrit le fonctionnement du driver.

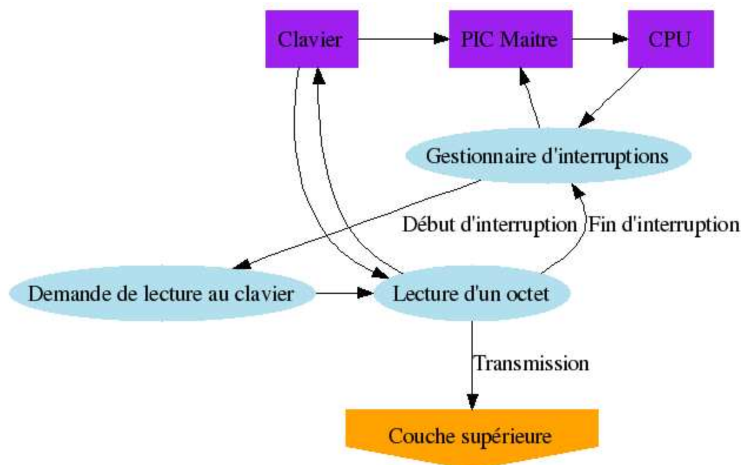

FIG. 4.5 – Fonctionnement du driver clavier

4.3 Le terminal

Nous avons décidé d'inclure un terminal dans notre SE, qui gère les entrées et sorties avec les périphériques de communication, et qui servira de relais central pour l'utilisation homme-machine.

4.3.1 Le clavier

Le terminal reçoit les informations brutes du clavier.

Le but de ce composant est d'interpréter la donnée, éventuellement de la transformer, et de la donner au shell qui à son tour devra gérer correctement cette donnée.

Il est prévu de pouvoir gérer plusieurs « keymaps » de clavier différents. Celui par défaut est bien entendu le « keymap » français. Lors de la réception d'une information, il va donc consulter la table française pour connaître la correspondance et ainsi trouver le caractère qui a été pressé.

Une fois les travaux de lecture et d'interprétation terminés, il va fournir la nouvelle donnée à la couche supérieure.

4.3.2 L'affichage

Le terminal est bien entendu capable d'afficher sur l'écran.

Pour cela, il utilise le pilote vidéo. Le terminal sert de relais pour recevoir les informations à afficher, puis pour lui transmettre.

Les informations reçues peuvent être de différentes origines : shell, clavier, etc...

4.4 Le pilote vidéo

4.4.1 Définition

Le pilote vidéo est une partie essentielle du SE. En effet, ce pilote fait le lien entre le noyau et l'écran. Il est utilisé pour afficher toutes les informations fournies par le noyau.

4.4.2 Principe

Sur un ordinateur, une zone d'adresse physique est présente et correspond à la mémoire vidéo. Ce mode vidéo est biensur très basique, et ce mode d'adressage n'est disponible que pour ce mode texte.

La première adresse de cette zone mémoire est 0x8000. Cette zone s'étend sur 4000 octets. En effet, la mémoire vidéo est représentée par un tableau de vingt-cinq lignes et de quatre-vingt colonnes comportant des cases de deux octets.

Les cases sont de la forme suivante :

- les huit premiers octets contiennent le code ASCII du caractère ;
- les huit suivants les attributs de la case (voir schéma 4.6 page suivante).

4.4.3 Programmation

Pour la programmation du pilote vidéo, nous avons utilisé une structure *mem_video* composée de deux champs : un pour le caractère et un pour les attributs.

On déclare donc un tableau de cette structure. Il nous suffit donc pour chaque case de remplir les deux champs.

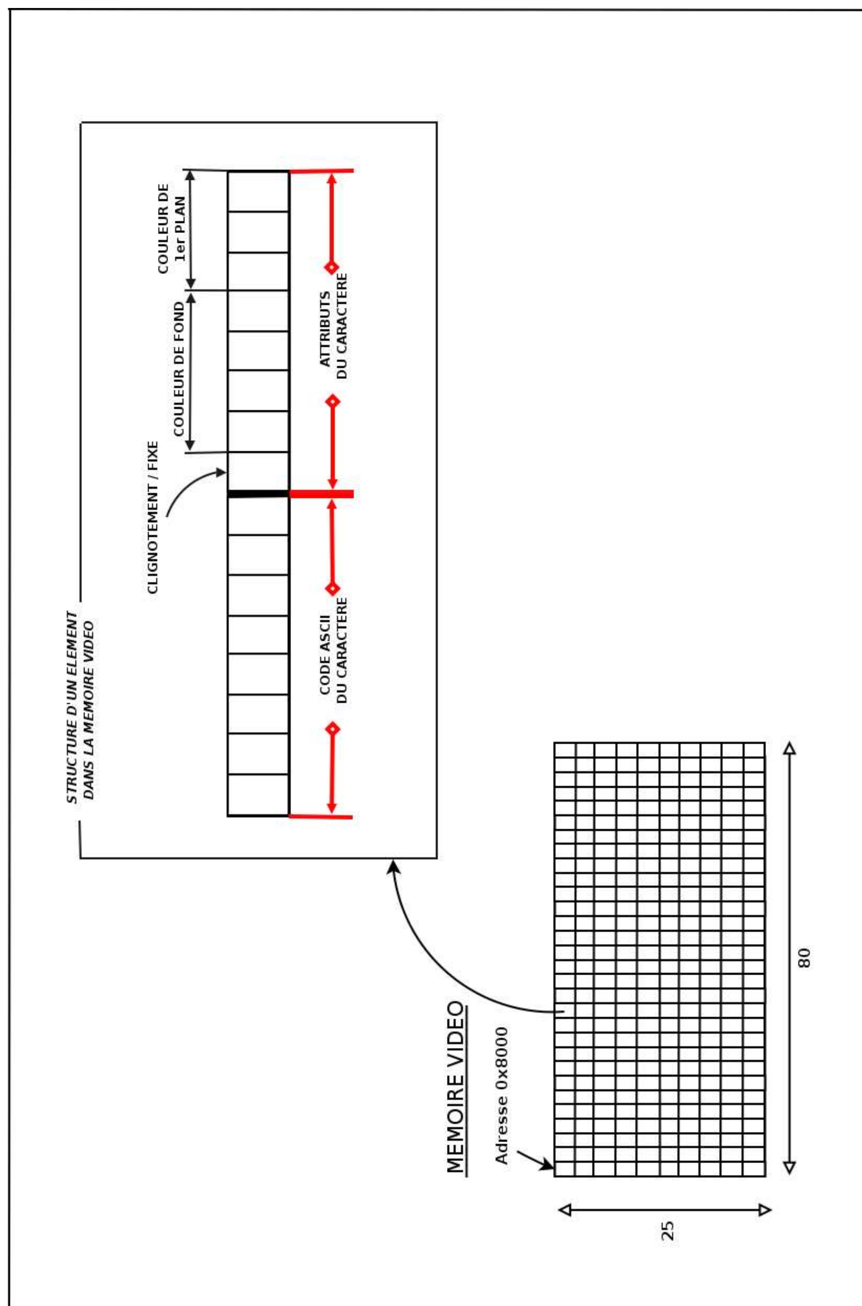

FIG. 4.6 – Mémoire vidéo

Ensuite, on incrémente le pointeur et on recommence l'opération avec tous les caractères à afficher. Cette fonction gère également les passages à la ligne et les tabulations.

Si le caractère « \n » est rencontré, la fonction met un nombre de caractères

nuls suffisants pour arriver en début de ligne suivante. De même, si elle rencontre le caractère « \t » elle met quelques espaces.

Le pilote vidéo est programmé de manière à faire défiler l'écran si le curseur est positionné à la dernière position. Ainsi, les applications n'ont qu'à utiliser les fonctions fournies pour écrire sur l'écran : l'affichage sera géré correctement dans tous les cas.

L'affichage supporte aussi plusieurs couleurs (de fond et de premier plan) ainsi que certains attributs (soulignement, clignotement, etc).

4.5 Le pilote Bochs

Le pilote Bochs est utilisé pour afficher divers messages lors de déroulement de notre noyau.

Avec ce pilote, nous pouvons afficher ce que nous souhaitons dans le terminal de la machine sur laquelle fonctionne l'émulateur Bochs. En effet, Bochs écoute sur le port 0xe9 et répète les messages sur la sortie standard.

Ce pilote est très simple puisqu'il se contente de récupérer le message désiré et de l'envoyer au port 0xe9.

Chapitre 5

Le système de fichiers

5.1 Généralités

5.1.1 Définition et objectifs d'un système de fichiers

Pour stocker des données sur un périphérique de stockage (disque dur, disquette, cdrom, clef usb ...), un système de fichiers est nécessaire.

En effet, on peut comparer un système de fichiers à un système de classement de données. Il serait impossible de retrouver des données si elles ne sont pas ordonnées.

Notre système de fichiers est basé sur l'ext2. Nous y avons apporté des modifications afin qu'il soit plus rapide et moins complexe à développer. NonFS ne possède donc pas toutes les fonctionnalités de l'ext2.

5.1.2 Structure de NonFS

Le NonFS est constitué d'un ensemble de groupes qui servent à limiter la fragmentation des fichiers et à accélérer l'accès aux données. Chaque groupe est subdivisé en plusieurs parties :

- le super bloc ;
- les descripteurs de groupes ;
- le bitmap du groupe ;
- la table d'inodes ;
- une partie « données ».

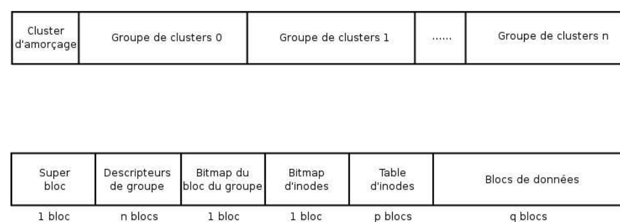

FIG. 5.1 – Structure du système de fichiers et d'un groupe de blocs

5.1.3 Unités d'allocation

Pour notre projet, le système de fichiers nous servira essentiellement pour stocker des données dans un disque dur.

Les disques durs sont des périphériques de stockages spécifiques, ils obéissent donc à des règles pré-établies. Les disques durs sont subdivisés en petites unités de 512 octets appelées « secteurs ».

Cependant le système de fichiers n'accèdent pas directement aux données en lisant secteur par secteur. Le système de fichiers a sa propre unité minimale de stockage appelée « bloc ».

Celle-ci n'a pas de taille fixe. C'est lors du formatage du disque que l'on définit la taille d'un bloc. Un bloc contient n secteurs. Il est important de comprendre l'utilité d'un bloc pour définir sa taille.

5.1.4 La taille des blocs

Fixons une taille arbitraire aux blocs, dix secteurs.

Un bloc mesure alors 5120 octets soit 5 kilo-octets. Le bloc étant la plus petite partie utilisable, si on veut stocker un fichier de 6 octets, les 5114 octets restant sont perdus.

Les conséquences peuvent être importantes s'il y a beaucoup de petits fichiers. Il est donc primordial de choisir un nombre adéquat de secteurs par bloc.

De plus il est plus rapide de lire un fichier de 5 kilo-octets dans deux blocs plutôt que dans dix.

En résumé, plus il y a de blocs, plus l'accès aux données sera lent mais moins il y aura de gaspillage. Et inversement.

5.2 Espaces réservés

5.2.1 Super bloc

Le super bloc est la première structure écrite sur le disque après la zone d'amorçage. Cette structure contient diverses informations sur les caractéristiques de la partition ou du disque :

5.2.2 Les descripteurs de groupe

Un descripteur de groupe est une structure qui contient les informations relatives au groupe de blocs :

5.2.3 Les Inodes

Un inode est une structure qui contient les caractéristiques d'un fichier :

Dans le cas d'un fichier, le champ *i_block* ne contient que quinze éléments maximum. Ce fichier a donc une taille de quinze fois la taille d'un bloc.

Il paraît impossible d'écrire des fichiers très gros. Cependant, la notion de blocs d'indirections permet de palier le problème.

FIG. 5.2 – Structure d'un super bloc

```
typedef struct{
uint32_t s_inodes_count ; /* Nombre total d'inodes */
uint32_t s_blocks_count ; /* Taille du système de fichiers en bloc */
uint16_t s_group_count ; /* Nombre de groupe de blocs */
uint32_t s_free_blocks_count ; /* Nombre de blocs libres */
uint32_t s_free_inodes_count ; /* Nombre d'inodes libres */
uint32_t s_first_data_block ; /* Numéro du premier bloc utilisable */
uint32_t s_block_size ; /* Taille d'un bloc : 0 pour 1024 ; 1 pour 2048 */
uint32_t s_root ; /* Numéro de l'inode racine */
uint32_t s_blocks_per_group ; /* Nombre de blocs par groupe */
uint32_t s_inodes_per_group ; /* Nombre d'inodes par groupe */
uint32_t s_first_inodes ; /* Numéro du premier inode non réservé */
uint16_t s_inode_size ; /* Taille d'un inode dur le disque */
uint16_t s_block_group_nr ; /*Numéro du groupe de blocs de ce super bloc
*/
uint8_t s_uuid[16] ; /* identificateur du système de fichiers */
uint8_t s_padding[816] ; /* Nuls pour compléter jusqu'à 1024 octets
} superBlock ;
```

FIG. 5.3 – Structure d'un descripteur de groupe

```
typedef struct {
uint32_t bg_block_bitmap ; /* Numéro du bloc de bitmap du bloc */
uint32_t bg_inode_bitmap ; /* Numéro du bloc de bitmap pour un inode */
uint32_t bg_inode_table ; /* Numéro de bloc du premier bloc de la table d'in-
odes */
uint16_t bg_free_blocks_count ; /* Nombre de blocs libres dans le groupe */
uint16_t bg_free_inodes_count ; /* Nombre d'inodes libres dans le groupe */
uint16_t bg_used_dirs_count ; /* Nombre de répertoires dans le groupe */
uint16_t bg_pad ; /* Alignement à un mot */
uint32_t bg_reserved[3] ; /* Nuls pour compléter jusqu'à 32 octets */
} group_desc ;
```

FIG. 5.4 – Structure d'un inode

```
typedef struct {
uint16_t i_mode ; /* Type de fichier */
uint32_t i_size ; /* Longueur du fichiers en octets */
uint32_t i_blocks ; /* Nombre de blocs de données du fichier */
uint32_t i_block[15] ; /* Pointeurs sur les blocs de données */
uint8_t i_padding[56] ; /* Nuls pour compléter jusqu'à 128 octets */
} inode ;
```

Les champs de 0 à 11 sont des pointeurs directs, cela signifie que le pointeur contenu dans ces champs pointe directement sur les blocs de données : c'est un adressage direct.

Les trois derniers champs utiliseront les blocs d'indirections. Le champs 12 utilise un adressage simple, le 13 double et le 14 triple.

Ce système permet de contenir des fichiers très volumineux.

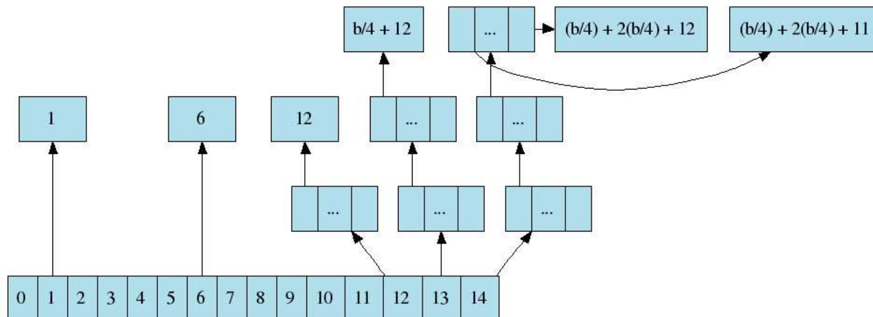

FIG. 5.5 – Blocs d'indirections

5.2.4 La table d'inodes

Cette table est une série de blocs consécutifs, dont chacun possède un nombre prédéfini d'inodes. Elle représente un index pour les fichiers. Quand un fichier, un répertoire ou un lien symbolique est créé, un inode est placé dans la table.

Cet inode pointe :
- sur des blocs de données si celui-ci est un fichier ;
- sur un bloc de données spécial (directory) si c'est un répertoire ;
- sur un inode s'il s'agit d'un lien symbolique.

5.2.5 Les bitmaps

Les bitmaps sont des blocs qui servent uniquement pour savoir si un bloc est utilisé ou s'il est libre.

Il en existe deux par groupe de blocs :
- un bloc pour la table d'inodes ;
- un bloc pour la partie « données ».

Un bitmap n'est rien d'autre qu'un buffer initialisé à 0. Dès qu'un bloc est écrit à l'adresse n, le n-ième octet de ce buffer passe à 1.

5.2.6 Les « directories »

Les « directories » sont des structures liées aux inodes. Elles sont écrites dans la partie « données ».

Exemple (voir la figure 5.7 page suivante) :

Dans le répertoire « / », il y a 2 dossiers (*« lost+found »* et *« usr »*) ainsi qu'un fichier « *fic* ».

L'inode du « / » pointe vers un tableau de directories qui contient « *lost+found* », « *usr* » et « *fic* ». Ces directories vont pointer respectivement vers l'inode associé.

Bref, l'ensemble des inodes et des directories forment un arbre complexe qui représente l'arborescence des fichiers dans le système de fichiers.

FIG. 5.6 – Structure d'un « directory »

```
typedef struct {
uint32_t num_inode; /* Numéro de l'inode associé */
uint16_t next; /* Possède encore un directory ? */
uint8_t name_len; /* Longueur du nom */
uint8_t file_type; /* Type de fichier */
uint8_t name[255]; /* Nom du fichier */
} directory;
```

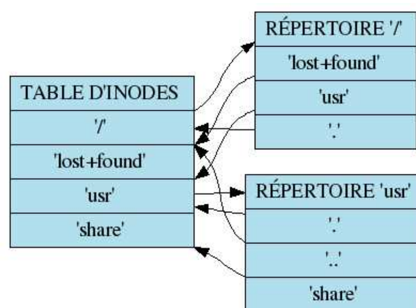

FIG. 5.7 – Correspondance entre inodes

5.2.7 Le formatage d'un disque

La première chose à définir lors d'un formatage est la taille des blocs. Il initialise le super bloc, les descripteurs de groupes, les bitmaps et les tables d'inodes.

Il crée le répertoire racine « / », ainsi que le répertoire « *lost+found* ».

La création de ces deux répertoires inclus la création de « directories » :

- « . » pour le répertoire « / » ;
- « . » pour le répertoire « *lost+found* » ;
- « .. » pour le répertoire « *lost+found* ».

5.2.8 Montage du système de fichier

Le montage d'un disque formaté en NonFS est la première étape pour pouvoir utiliser le disque. Cela consiste à initialiser une variable de type *file_system* afin que les programmes puissent faire des modifications sur le disque. Cette initialisation charge en mémoire le super bloc, les descripteurs de groupe, ainsi que l'adresse de l'inode racine.

5.2.9 Les fonctions d'entrées sorties

L'écriture et la lecture de blocs s'effectuent grâce au driver IDE. Cependant l'unité qu'utilise le driver IDE est différente de celle du système de fichiers, ce qui nous a imposé de créer des "fonctions hybrides".

De plus d'autres fonctions permettent de lire une structure d'inodes, en autres, dans un buffer passé par le driver IDE.

Donc, le système de fichiers possède un large panel de fonctions permettant de faciliter le développement du système de fichiers lui-même.

5.2.10 Les autres fonctions

Le système de fichiers met à disposition de nombreuses fonctions afin de permettre à une console notamment d'effectuer des tâches telles qu'un mkdir, ls, cd, etc, ...

5.2.11 La fragmentation

Pour écrire un fichier, le système de fichiers vérifiera que la place nécessaire est disponible. Il se chargera ensuite de trouver les n blocs successifs. S'il ne les trouve pas, il devra se charger de diviser le fichier dans plusieurs groupes en limitant le nombre de fragments.

Annexe A

Listings

A.1 Fichier de configuration de Bochs

```
# configuration file generated by Bochs
config_interface : textconfig
display_library : x
megs : 16
romimage : file=/usr/share/bochs/BIOS-bochs-latest, address=0xf0000
vgaromimage : file=/usr/share/vgabios/vgabios.bin
boot : floppy
floppya : 1_44="nonos.img", status=inserted
# no floppyb
#ata0 : enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
#ata0-master : type=disk, path=dd.img, cylinders=304, heads=16, spt=63
ata0 : enabled=0
ata1 : enabled=0
ata2 : enabled=0
ata3 : enabled=0
parport1 : enabled=1, file=""
parport2 : enabled=0
com1 : enabled=1, mode="null", dev=""
com2 : enabled=0
com3 : enabled=0
com4 : enabled=0
i440fxsupport : enabled=0
usb1 : enabled=1, ioaddr=0xff80, irq=10
# no sb16
floppy_bootsig_check : disabled=0
vga_update_interval : 30000
keyboard_serial_delay : 20000
keyboard_paste_delay : 100000
floppy_command_delay : 50000
ips : 500000
text_snapshot_check : 0
mouse : enabled=0
```

```
private_colormap : enabled=0
clock : sync=none, time0=local
# no ne2k
pnic : enabled=0
# no loader
log : -
logprefix : %t%e%d
debugger_log : -
panic : action=ask
error : action=report
info : action=report
debug : action=ignore
pass : action=fatal
keyboard_mapping : enabled=0, map=
keyboard_type : mf
user_shortcut : keys=none
# no cmosimage
```

A.2 Création d'une image disquette

```
# On obtient le nombre de 2880 avec :
# Sectors = Cylinders * Heads * SectorsPerTrack
# Sectors = 80 * 2 * 18
dd if=/dev/zero of=nonos.img bs=512 count=2880

# Création du système de fichiers
mke2fs -m 0 nonos.img

#Installation de grub
grub-install nonos.img

#montage de l'image
mount -o loop nonos.img /mnt/img

#Creation d'un fichier de boot
mkdir /mnt/img/boot/grub -p
cp ~/menu.lst /mnt/img/boot/grub/
cp ~/nonos/nonos /mnt/img/boot/

#Démontage de l'image
umount /mnt/img
```