République algérienne démocratique et populaire

Ministère de l'Enseignement Supérieure et de La Recherche Scientifique

Université Saad Dahleb Blida

Faculté Des Sciences

Département d'Informatique

Mémoire de Fin D'étude

Pour l'obtention du Diplôme de Master(LMD) en Informatique

Option : Ingénierie du Logiciel

THEME

Conception et Réalisation d'un Système d'exploitation didactique

Présenté par :

- ✓ **Bandoui Dounyazed**

Soutenue : Devant le jury composé de :

Président : Pr. Bounekhla M'hamed

Examinateur : Mlle. MANCER Yassmine

Promoteur : Mr. OULD-AISSA Ahmed

Promotion : 2016

Remerciements

Louange à Dieu de m'avoir donnée le courage, la santé, la patience et la force de terminer mes études.

-Au terme de ce modeste travail, on tient à exprimer toute notre gratitude et tous nos remerciements à ceux qui nous ont aidés à réaliser ce mémoire, ainsi qu'à tous les enseignants du département de Mathématique et Informatique qui ont contribué à notre formation notamment pour leurs conseils et leurs dévouements.

-On tient à remercier très vivement Mr. OULD-AISSA Ahmed. d'avoir bien voulu nous encadré et suivre notre travail, on lui exprime notre reconnaissance pour son attention et précieux conseils contribuant à la réalisation de ce travail.

-Enfin, on remercie tous nos collègues et amis de l'université de Saad Dahlab Blida pour leurs soutiens moral et pour les moments merveilleux toutes au long de notre cursus.

Bandoui DZ

Dédicace

Je dédie ce modeste travail à :

La mémoire de ma très chère maman ;

Mon père que je remercie tendrement pour son affection, son aide et son sacrifice pour ma réussite ;

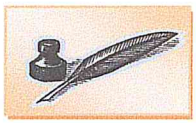
Mes deux frères : Khoya « Nasser Eddine » & Nour Eddine ;

Mes deux sœurs : Farah & Afifa ;

Mes grands-mères ; mes tantes et mes oncles ; mes cousins et mes cousines

Tous Mes ami(e)s

DZ...



Sommaire

Liste des figures et tableaux

Introduction générale :

Le système d'exploitation est imposé dans le domaine d'informatique dès les premiers jours comme étant le programme principale de fonctionnement et de gestion des différentes machines numériques.

L'élargissement des domaines d'utilisation de ces machines à imposer l'évolution rapide des systèmes d'exploitation pour couvrir tous les domaines et aussi les différentes catégories des usagers de ces machines et surtout les personnes à faible formation en informatique.

De l'autre côté les spécialistes du domaine (conception et réalisation) doivent avoir une formation très profonde pour bien l'utilisé et l'exploité, en plus de ça la sécurité des systèmes informatiques et information repose essentiellement sur le contrôle du système d'exploitation.

Dans cette optique, notre projet de fin d'étude concerne la mise en place d'un système d'exploitation didactique qui permet aux étudiants et aux développeurs de connaitre et testés les différents composent du système avec une accessibilité totale et contrôle.

Pour ce là, nous avons suivies les étapes suivantes :

- Présentation du système et les principes du contrôle
- Vu globale sur la machine ciblé (processeur et composants)
- Approche de conception
- Réalisation et teste

CHAPITRE I

Système d'exploitation et Machine

I- Système d'exploitation

I-1- Définition:

Un système d'exploitation est ensemble de programmes, qui permettent d'utiliser l'ordinateur (ensemble des composants matériels), de façon optimale et équitable. Plus spécifiquement, ces programmes regroupe un langage de commandes, un système de gestion des mémoires, un système de gestion de fichiers, un système de gestion de l'unité centrale de traitement, un système de gestion des entrées/sorties.

L'ensemble des services fournit par un système d'exploitation permet de définir, pour l'utilisateur, une nouvelle machine dite virtuelle, par opposition à la machine réelle ou physique.

La description et mode d'emploi de ces services constituent l'interface de système informatique. Cette interface définit, elle aussi, un langage (celui de la machine virtuelle), qui permet aux utilisateurs de communiquer avec le système, elle contient toute l'information nécessaire à une utilisation simple de celui-ci.

I-2- Système didactique :

Les systèmes didactiques ont pour objectives l'apprentissage de la conception et la manipulation du système lui-même. Donc un accès totale aux déférents composants du système pour les changé et observé leurs fonctionnements.

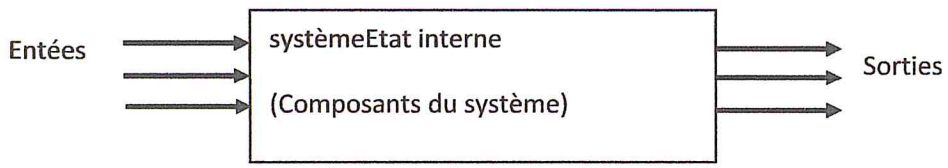


Fig. I.1 Schéma générale du système

Un système est caractérisé par trois parties : les entrées, les composants interne et les sorties.

- Les entrées assurent le passage de la commande externe à l'intérieur du système ;
- Les composantes interne c'est les parties constituant le système et assurent sont fonctionnement ;
- Les sorties véhiculant les actions du système à l'extérieur

La manipulation des systèmes (cas générale : informatique, électronique, mécanique …) se base essentiellement sur trois notions : Décomposé, Contrôlé et Observé.

Décomposable : Un système est constitué par des éléments internes, ces éléments peuvent être considéré comme un système donc en peu appliqué les même principes sur celui-ci jusqu'à l'arrivé aux composants élémentaires (atomique). Ce décomposition nous permettre d'étudie des sous système très simple et à la fin étudie les interactions entre aux pour constituer le système globale.

Contrôlable : Le système est contrôlable si il possède des éléments qui nous permettant d'introduire des actions externes pour changer l'état interne du système. Le système est dit totalement contrôlable si on peut agir directement sur tous les composants internes.

Observable : Le système est dit observable si on peut capter directement le comportement du système. Le système est totalement observable si on peut capter et mesurer tous les sorties du système.

La relation entre le matérielle et le système est fortement lier, le faite que toute opération de contrôle ou d'observé doit être réalisé physiquement. Tout ça nous ramène à vérifier si le processeur est doté des mécanismes qui permettent la réalisation de ces opérations.

Il est évident que le processeur opère dans un environnement matérielle varies, et que les éléments internes doivent trouver des équipements externes capable d'agir correctement vis à vis les opérations de contrôle ou d'observation.
L'étude des deux parties nous donnent une vue réelle pour la réalisation de notre système.

I-3- La machine PC d'IBM :

3-1- Architecture

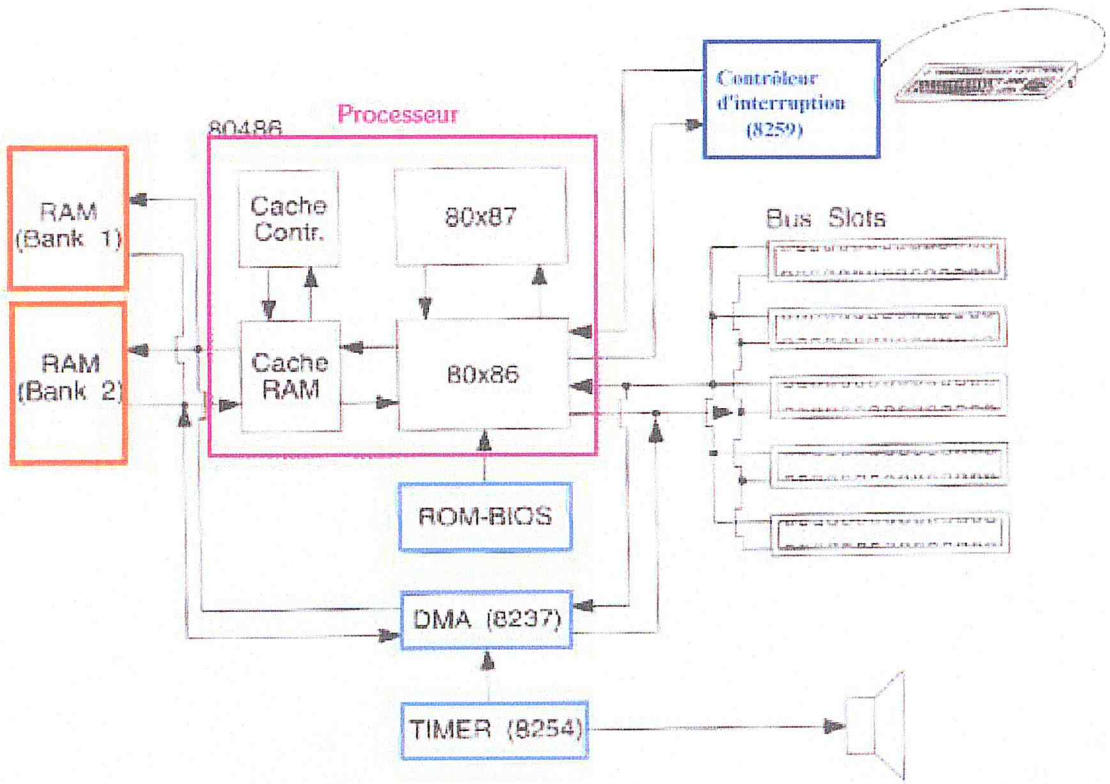


Fig. carte mère typique pour processeur 486

Sur cette carte figurent uniquement les éléments clés, à savoir le processeur 486, l'eprom contenant le bios, les RAM, un chip contrôleur DMA (Direct Memory Access) ainsi qu'un timer et les slots dans lesquels viendront s'enficher les cartes d'adaptateur graphique et celle spécialisée dans l'interfaçage des entrées sorties, mais aussi d'autres laissées au libre choix de l'utilisateur en fonction de l'emploi qu'il fera de son outil informatique.

3-2- Le microprocesseur Intel 80x86 :

Il est développé en 1978 par Intel, le microprocesseur 8086 est un processeur CISC 16 bits. Il s'agit du premier microprocesseur de la famille des « x86 » comprenant les célèbres 80386 et 80486.

Ses caractéristiques principales sont les suivantes :

- ✓ Bus de données d'une largeur de 16 bits.

- ✓ Bus d'adresses de 20 bits, ce qui permet d'adresser un total de 1 mégaoctet de mémoire.
- ✓ 14 registres de 16 bits dont un registre d'état contenant des indicateurs binaires.

Le 8086 autorise un mode de fonctionnement en pas à pas, ainsi que l'utilisation d'opérations spécifiques appelées interruptions permettant au 8086 de communiquer avec les autres périphériques de l'ordinateur.

3-2-1- Les registres :

Un registre est une petite zone mémoire interne au processeur lui permettant de stocker une adresse ou une donnée.

Le microprocesseur Intel 8086 fournit des registres de 8 et 16 bits

Les registres du 8086 se décomposent en 5 grandes familles :

Les Registres généraux sont :

- ✓ 4 registres de données, se décomposant chacun en deux parties : une partie « haute » et une partie « basse » de 8 bits chacune, ce qui permet au microprocesseur de manipuler des données sur 8 ou 16 bits :
  - AX (décomposable en AH et AL) sert d'accumulateur et est principalement utilisé lors d'opérations arithmétiques et logiques ;
  - BX est la plupart du temps utilisé comme opérande dans les calculs ;
  - CX est utilisé comme compteur dans les structures itératives ;
  - DX, tout comme AX, est utilisé pour les calculs arithmétiques et notamment dans la division et la multiplication. Il intervient également dans les opérations d'entrées/sorties.
- ✓ 8 registres de segmentation :
  - CS segment de code permet de déterminer les adresses sur 20 bits ;
  - DS segment de données ;
  - SS segment de pile ;
  - ES segment supplémentaire. registres pointeurs ou d'index :
  - SP pointeur de pile pointe sur le sommet de la pile de données ;
  - BP pointeur de base pointe sur la base de la pile de données ;
  - SI index de source ;
  - DI index de destination.
- ✓ registre spécial contenant 9 indicateurs binaires nommés « flags » :
  - AF (indicateur de retenue auxiliaire) ;
  - CF indicateur de retenue;

- OF (indicateur de débordement)
- SF (indicateur de signe) ;
- PF (indicateur de parité);
- ZF (indicateur de zéro) est mis à 1 lorsque le résultat d'une opération vaut 0 ;
- DF (indicateur de direction) ;
- IF (indicateur d'autorisation d'interruption) ;
- TF (indicateur d'interruption pas à pas).

✓ Le registre de contrôle :

- Le registre CR0 (32bits) :c'est un registre de contrôle général qui inclut le registre MSW 80286 plus un bit de mise en fonctionnement de l'unité de pagination. Les 16 bits de poids faibles sont équivalents au MSW (Machine Status Word)

00 - PE : Protection Enable0
01 - MP : Monitor Coprocessor
02 - EM : Emulate Coprocessor
03 - TS : Task Switch
04 - ET : Processor Extension
05 - NE : Numeric Error Enable
16 - WP : Write Protect
18 - AM : Alignment Mask
29 - NW : Not Write-Through
30 - CD : Cache Disable
31 - PG : Paging Enable

- Le registre CR1 (32bits) : Réservé
- Le registre CR2 (32bits) : Contient l'adresse de la dernière page ayant provoqué une faute.
- Le registre CR3 (32bits) : Contient l'adresse de base du répertoire de pagination.

PCD (Page-level Cache Disable)
PWT (Page-level Writes Transparent)

3-2-2- Les instructions :

Le 8086 est programmable dans un langage d'assemblage comportant des instructions utilisant les registres, les flags, la mémoire et, en ce qui concerne les interruptions. Les instructions les plus couramment utilisées dans Le 8086 sont :

- ✓ Instructions arithmétiques : addition (ADD), soustraction (SUB), multiplication (MUL), division (DIV), incrémentation (INC), décrémentation (DEC) et échange (XCHG) ;
- ✓ Instructions logiques : et (AND), ou (OR) et non (NOT) ;
- ✓ Instruction de comparaison (CMP) : met à jour les flags pour permettre l'utilisation des instructions de saut ;
- ✓ Instructions de saut : saut si égal (JE), saut si différent (JNE), saut si inférieur (JL),
- ✓ Instructions de gestion de la pile : empilement (PUSH) et dépilement (POP)
- ✓ Instruction d'appel (CALL) et de retour (RET) ;

(Table des instructions dans Annex1)

3-2-3- Modes de fonctionnement du processeur

a) Mode réel

Au démarrage de la machine le processeur est dans un mode appelé réel. Ce mode correspond au fonctionnement des processeurs 8086 :

- Ses registres ont une taille de 16 bits
- Son bus d'adresse mémoire est accessible sur 20 bits

Les adresses accédant la mémoire ont donc une taille maximale de 20 bits, ce qui nous donne un espace mémoire accessible de 2^20 soit 1Mo.

Pourtant un problème se pose, comment adresser la mémoire sur 20 bits alors que nous disposons de registres d'une taille de 16 bits ? ... Pour cela nous allons utiliser 2 registres:

- ➢ Le premier appelé registre de segment est multiplié par 16 (ce qui décale l'adresse de 4 bits vers la gauche)
- ➢ Le deuxième appelé registre d'offset est simplement ajouté au premier afin de compléter les 4 premiers bits manquant

L'intérêt du mode réel est qu'il est le seul qui garantir une bonne utilisation des instructions du BIOS, qui sont des instructions de type 16 bits, du fait que dans ce mode les registres sont sur 16 bits.

b) Mode protégé :

Dans le 80286 a été introduit, en plus du mode réel ci-dessus, un mode d'adressage virtuel protégé (dit mode protégé) dans lequel 4 niveaux de privilèges d'accès sont définis ce qui permet de protéger les données et le code et d'exclure (en principe) le crash du PC.

En mode protégé on peut accéder aux 24 bits du bus, et non à 20 comme en mode réel : cela implique quelques registres supplémentaires :

- TR task register
- MSW machine status word
- IDT interrupt descriptor table
- GDT global descriptor table

Le **mode protégé** permet d'exploiter la totalité des possibilités du microprocesseur, avec notamment l'adressage de toute la mémoire et le support pour l'implémentation de systèmes multi-tâches et multi-utilisateurs.

Ce mode est celui du fonctionnement habituel du processeur :

- Les registres ont une taille de 32 bits
- Le bus d'adresse mémoire est accessible sur 32 bits

Les adresses accédant la mémoire ont donc une taille maximale de 32 bits, ce qui nous donne un espace mémoire accessible de 2^32 soit 4Go. De plus ce mode nous garantit une certaine sécurité sur les zones mémoires accessibles par le biais des niveaux de privilèges.

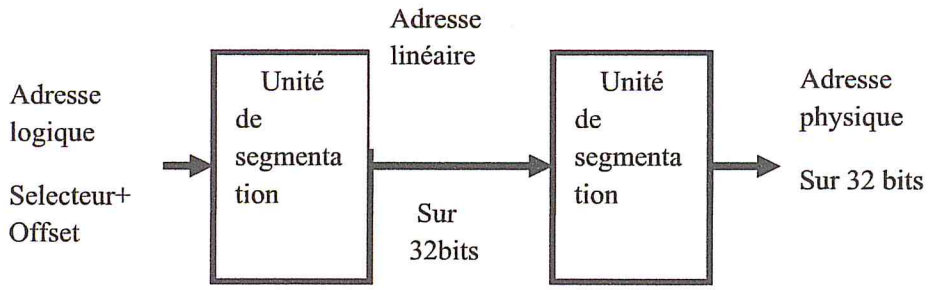
- **Adresser la mémoire en mode protégé**

  Différents types d'adresses

En mode protégé, il existe trois types d'adresses :

- L'**adresse logique** est directement manipulée par le programmeur. Elle est composée à partir d'un **sélecteur de segment** et d'un **offset**.
- Cette adresse logique est transformée par l'unité de segmentation en une **adresse linéaire**, sur 32 bits.
- Cette adresse linéaire est transformée par l'unité de pagination en une **adresse physique**. Si la pagination n'est pas activée, l'adresse linéaire correspond à l'adresse physique.

Ce schéma résume le principe de l'adressage en mode protégé :



Il peut aussi fonctionner en adressage virtuel avec protections.

Le 80286 passe en mode virtuel protégé par la mise à 1 d'un bit du registre mot d'état.
Chaque tâche dispose alors d'un giga octets (230) de mémoire virtuelle placée dans 16M octets (224) de mémoire réelle.
L'adresse est constituée d'un sélecteur et d'un déplacement.
Le sélecteur (16 bits) désigne une table de descripteurs et une entrée dans celle-ci.
Le descripteur (48 bits) contient, outre l'adresse de base en mémoire et la taille de la page de mémoire virtuelle, des indications concernant les droits d'accès à cette page.
Le déplacement est ajouté à cette adresse de base pour constituer l'adresse physique.
Le 80286 utilise comme sélecteurs les 4 registres de segments CS DS SS et ES. Il possède en outre 4 registres de 48 bits contenant le descripteur associé à chacun de ces sélecteurs.
La mise à jour de ces registres est automatiquement faite par le 80286 à chaque modification des sélecteurs par accès aux tables en mémoire.
Il existe 3 tables de descripteurs décrites chacune par un registre de 40 bits dont un champ de 24 bits pointe sur le début de la table en mémoire et un second champ de 16 bits donne la taille de la table. Les 3 tables sont les suivantes :

- Table globale GDT contenant les descripteurs des pages accessibles par toutes les tâches.
- Table locale LDT contenant les descripteurs des pages propres à une tâche.
- Table d'interruptions IDT contenant les descripteurs des procédures d'interruption (256 interruptions)

Le 80286 contient de plus les registres suivants :

MSW : Mot d'état (16 bits) 4 bits seulement sont utilisés pour le passage en mode virtuel,
GDTR : Descripteur de la table globale (24 bits d'adresse et 16 bits de longueur)
LDTR : Descripteur de la table locale (24 bits d'adresse et 16 bits de longueur)
IDTR : Descripteur de la table d'interruption (24 bits d'adresse et 16 bits de longueur)
CSDC : copie du descripteur dont le sélecteur est CS (48 bits)
DSDC : copie du descripteur dont le sélecteur est DS (48 bits)
SSDC : copie du descripteur dont le sélecteur est SS (48 bits)
ESDC : copie du descripteur dont le sélecteur est ES (48 bits)
TR : sélecteur associé à la tâche courante (16 bits) désigne dans la table locale ou globale un descripteur de tâche.
TRB : Adresse de base de la table d'état de la tâche courante (24 bits)
TRL : Taille de la table d'état de la tâche courante (16 bits)

Remarque : Le registre d'état SR utilise 3 bits supplémentaires :

- 2 pour décrire le niveau de privilège nécessaire à l'utilisation des instructions d'E/S.
- 1 (bit NT) pour l'enchaînement des tâches.

Les instructions relatives aux tâches et à la mémoire virtuelle.

CTS : Remise à 0 du bit TS du mot d'état. Ce bit est mis à 1 par le 80286 lors d'un changement de tâche de façon à noter qu'il sera probablement nécessaire de changer le contexte des coprocesseurs s'ils doivent être utilisés par la nouvelle tâche. Si ce bit est à 1 et qu'un coprocesseur doit être utilisé le 80286 exécutera une interruption de type 7. CTS sera donc utilisée par le programmeur après avoir mis à jour le contexte des coprocesseurs.

LGDT : Chargement du descripteur de table globale dans GDTR

LIDT : Chargement du descripteur de table globale dans IDTR

LLDT : Chargement du descripteur de table globale dans LDTR

SGDT : Rangement du descripteur de table globale GDTR

SIDT : Rangement du descripteur de table globale IDTR

SLDT : Rangement du descripteur de table globale LDTR

LTR : Chargement du registre TR

STR : Rangement du registre TR

Toute modification de TR provoque un accès à la table locale ou globale permettant de lire le descripteur de tâche et de mettre à jour les registres TRB et TRL

LMSW : Chargement du mot d'état MSW

SMSW : Rangement du mot d'état MSW

LAR : Place dans l'opérande l'octet du descripteur associé au sélecteur spécifié décrivant les droits d'accès

LSL : Place dans l'opérande le double octet du descripteur associé au sélecteur spécifié contenant la taille du segment

ARPL : Ajuste le niveau de privilège du sélecteur désigné à celui décrit dans l'opérande (cette instruction ne peut que diminuer le niveau initial)

VERR : Vérifie si le segment décrit par le sélecteur spécifié peut être lu

VERW : Vérifie si le segment décrit par le sélecteur spécifié peut être écrit

Remarque : Pour les 5 dernières instructions se reporter à la description des descripteurs en 2.5 et des privilèges en 2.6

3-2-4- Les interruptions :

Les interruptions sont des signaux envoyés au processeur pour l'avertir d'événements particuliers. On distingue trois types d'interruptions :

a) les **interruptions matérielles** sont déclenchées par les périphériques (clavier, disque, souris, etc.). Par exemple, une interruption matérielle va être déclenchée par une unité de disque pour prévenir le processeur que des données sont prêtes en lecture ou par une carte réseau pour prévenir de l'arrivée d'une trame Ethernet.

- Si chaque périphérique pouvait envoyer directement un signal au processeur, il faudrait sur celui-ci autant de broches que de périphériques. Le contrôleur d'interruption programmable (PIC, Programmable interrupt controler) est un circuit qui permet d'éviter cela en multiplexant les requête d'interruption envoyées par les périphériques. Le 8259A est l'un des premiers PIC développé par Intel.

b) les **interruptions logicielles** sont déclenchées volontairement par le programme. Elles permettent de gérer les appels système.

c) les **exceptions** sont déclenchées par le processeur en cas de faute (division par zéro, défaut de page, etc.)

- **Parmi ces interruptions certaines sont associées à des événements particuliers :**

- 0 à 7 : même utilisation que dans le 8086
- 8 : en mode réel : n° d'IT dépassant les limites de la table en mode virtuel : 2 exceptions ont été détectées au cours de la même instruction
- 9 : Débordement du segment par le coprocesseur
- 10 : En mode virtuel seulement : table d'état de tâche incorrecte
- 11 : En mode virtuel seulement : segment de mémoire virtuelle non présent en mémoire physique
- 12 : En mode virtuel seulement : débordement de la pile ou segment de pile non présent en mémoire physique.
- 13 : En mode réel : débordement de segment

  En mode virtuel : violation de protection ou tentative de passage à un niveau de privilège supérieur

Lorsque le processeur reçoit une interruption, il interrompt la tâche en cours, sauvegarde le contexte de celle-ci, et exécute la routine de service associée à l'interruption (**ISR - Interrupt**

Service Routine). Une fois la routine exécutée, le système redonne en général la main à la tâche interrompue.

3-4- Gestion des interruptions matérielles externes : le 8259A

Les interruptions matérielles peuvent êtres vues comme des sonnettes que tirées par les périphériques pour prévenir le processeur que quelque chose se passe. Mais si chaque périphérique pouvait envoyer directement un signal au processeur, il faudrait sur celui-ci autant de broches que de périphériques. Le contrôleur d'interruption programmable (PIC, Programmable interrupt controler) est un chipset qui permet d'éviter cela en multiplexant les requête d'interruption envoyées par les périphériques. Le 8259A est l'un des premiers PIC développé par Intel

3-5- Plan de la Mémoire Centrale:

Programmer un secteur de boot et un noyau, cela signifie entre autres organiser l'occupation en mémoire des différents composants. Pour ne pas vous y perdre, je vous conseille d'utiliser des petits schémas. Les 5 minutes passées à crayonner sur un bout de papier vous ferons parfois économiser des heures de débogage ! Dans notre cas, la mémoire est occupée de la façon suivante après le chargement du noyau :

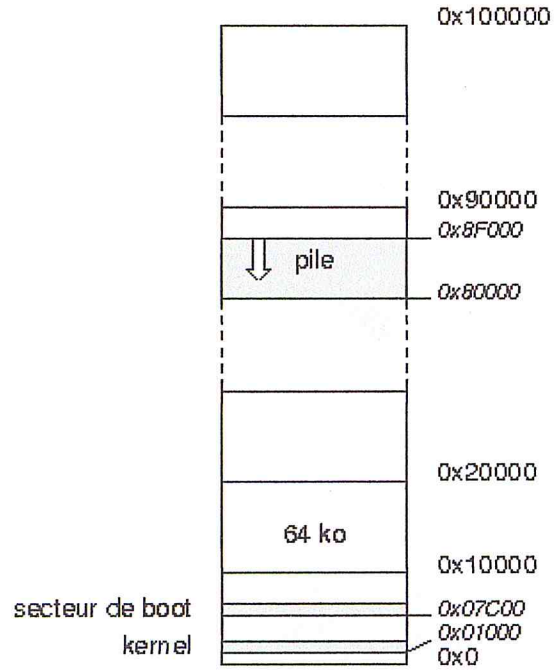


Fig I.4 Plan mémoire

En conséquence pour l'utilisateur, tant au niveau soft qu'au niveau hard, quel que soit le processeur utilisé, le plan mémoire peut se limiter à un espace de 1 M octet environ selon le schéma ci-dessous.

Ci-dessous un tableau de la zone voisine de 0K dans laquelle on trouve les interruptions et les ports d'entrée/sortie principaux

Tab I.1 Liste des interruptions

№	Adresse	Fonction
00	000-003	CPU : Division par zéro
01	004-007	CPU : Pas à pas
02	008 -00B	CPU : NMI (défaut dans RAM)
03	00C-00F	CPU : Point d'arrêt atteint
04	010-013	CPU : Débordement numérique
05	014-017	Copie d'écran
06	018-01B	Instruction inconnue (80286 seul)
07	01D-01F	Réservé
08	020-023	IRQ0 : Timer (appel 18,2 fois/sec)
09	024-027	IRQ1 : Clavier
0A	028-02B	IRQ2 : Deuxième 8259 (AT uniquement)
0B	02C-02F	IRQ3 : Interface série 2
0C	030-033	IRQ4 : Interface série 1
0D	034-037	IRQ5 : Disque dur
0E	038-03B	IRQ6 : Disquette
0F	03C-03F	IRQ7 : Imprimante
10	040-043	BIOS : Fonction vidéo
11	044-047	BIOS : Détermination configuration
12	048-04B	BIOS : Détermination taille RAM
13	04C-04F	BIOS : Fonction disquette/disque dur
14	050-053	BIOS : Accès à interface série
15	054-57	BIOS : Fonction cassette ou étendues
16	058-05B	BIOS : Interruption du clavier
70	1C0-1C3	IRQ08 : Horloge temps réel (AT seulement)
71	1C4-1C7	IRQ09 : (AT seulement)
72	1C8-1CB	IRQ10 : (AT seulement)
73	1CC-1CF	IRQ11 : (AT seulement)
74	1DO-1D3	IRQ12 : (AT seulement)
75	1D4-1D7	IRQ13 : 80287 NMI (AT seulement)
76	1D8-1DB	IRQ14 : Disque dur (AT seulement)
77	1DC-1DF	IRQ15 : (AT seulement)
78-7F	1E0-1FF	Inutilisé
80-F0	200-3C3	Utilisé à l'intérieure de l'interpréteur
F1-FF	3C4-3CF	Inutilisé

3-5- Le BIOS :

Le BIOS (Basic Input/Output System) est un petit programme qui situé sur la carte mère de l'ordinateur dans une puce de type ROM .Le nom BIOS provient du fait que la première partie doit implémenter la façon d'accéder au clavier, au moniteur et à la mémoire de masse (disquette, disque dur) pour pouvoir charger la seconde partie.

Le concept de BIOS est devenu si répandu qu'il a fini par permettre de charger plusieurs systèmes d'exploitation.

Le BIOS est le premier programme chargé en mémoire dès que vous allumez votre ordinateur. Il assure plusieurs fonctions:

- le POST (Pre-Operating System Tests ou Power-On Self-Tests selon les écoles) : c'est l'ensemble des tests qu'effectue le BIOS avant de démarrer le système d'exploitation:
    - vérifier que la carte mère fonctionne bien (barrettes de mémoire vive (RAM), contrôleurs de ports série, parallèle, IDE, etc.)
    - vérifier que les périphériques simples ("Basic") connectés à la carte mère fonctionnent bien (clavier, carte graphique, disques dur, lecteur de disquette, lecteur de CD-Rom...)
    - paramétrer la carte mère (à partir des informations stockées dans les CMOS (voir ci-dessous)).
- chercher un disque sur lequel il y a un système d'exploitation prêt à démarrer.

Le BIOS peut également rendre des services au système d'exploitation en assurant la communication entre les logiciels et les périphériques, mais seulement pour les périphériques simples (clavier, écran, etc.).

Le BIOS contient aussi généralement un programme qui permet de modifier les paramètres de la carte mère. Ce programme est appelé **setup**.

II-5-1- Liste des interruptions du BIOS :

Les concepteurs du 8086 ont réservé les interruptions 00h à 12h comme interruptions internes. Le BIOS proprement dit occupe les interruptions 13h `a 1Bh, et en fait également les interruptions 05h, 08h, 09h et 0Bh qui ne sont pas utilisées sur le 8086 bien que réservées.

On a :

– INT 00h : Division par zéro. Le gestionnaire est appel´e lorsqu'un essai de division par zéro est tenté. Le BIOS de l'IBM PC se contente de faire apparaitre un message et suspend le système.

– INT 01h : ´Etape par étape. Cette interruption est utilisée par debug et d'autres débogueurs pour exécuter un programme ligne par ligne.
– INT 03h : Point d'arrêt. Cette interruption est utilisée par debug et d'autres débogueurs pour arrêter l'exécution d'un programme.
– INT 04h : Dépassement de capacité.
– INT 05h : Impression de l''ecran.
– INT 08h : Temporisateur du système.
– INT 09h : Interruption du clavier avec de nombreuses fonctions.
– INT OBh : contrôle du premier port série COM1.
– INT 0Ch : contrôle du deuxième port série COM2.
– INT 0Dh : contrôle du deuxième port parallèle LPT2.
– INT 0Eh : contrôle des lecteurs de disquette. Signale une activité d'un des lecteurs, telle qu'une opération d'entrée-sortie.
– INT 0Fh : contrôle du premier port parallèle LPT1.
– INT 10h : interruption avec de nombreuses fonctions concernant le moniteur.
– INT 11h : détermination de l''equipement, utilisé en particulier lors du démarrage de l'ordinateur.
– INT 12h : détermination de la taille mémoire (jusqu''a 640 Ko, taille maximale gérée par (MS-DOS), en particulier lors du démarrage de l'ordinateur.

3-6- Le contrôleur DMA

Le contrôleur DMA (Intel 8237) permet d''ecrire des données directement dans la RAM ou bien de lire des données directement depuis la RAM, sans passer par le microprocesseur. Ceci est surtout utilisé pour l'entré-sortie sur disquette, qui est relativement lente. Il comprend quatre canaux. Le canal 0 est dédié au rafraichissement de la mémoire. Les trois autres canaux sont éventuellement utilisés pour le réseau, le contrôleur de disquette et le contrôleur de disque dur.

3-7- Le timer : joue un double rôle de cadencement des opérations du contrôleur DMA, mais aussi comme générateur de sons élémentaires à destination du haut-parleur du PC.

3-8- LES ADAPTATEURS

Dans un ordinateur de bureau, outre la carte-mère support du processeur, figurent le plus souvent un certain nombre de cartes adaptateurs (dites cartes filles). L'évolution technologique récente (hyper miniaturisation des composants) a permis d'intégrer certaines d'entre elles sur la carte mère, mais certains adaptateurs ne seront pas intégrés pour laisser à l'utilisateur le choix d'une configuration personnalisée. Par contre nous noterons que dans un microordinateur portable, en raison de la miniaturisation indispensable tous les adaptateurs seront figés sur l'unique circuit imprimé constituant la carte mère, la personnalisation sera exclue.

Dans tous les cas un adaptateur est caractérisé par trois éléments de configuration, généralement plus ou moins programmables :

- l'adresse du port PC qui lui est lié, c'est à dire l'adresse par laquelle cette carte (ou son circuit équivalent dans une machine récente) est reconnue par le processeur
- la zone adresse qui est réservée dans le plan mémoire du système à leur mémoire interne lorsqu'ils en possèdent
- l'interruption qui leur sera affectée
- enfin pour certains le canal DMA qu'ils utilisent

3-8-1- Adaptateur graphique

L'adaptateur graphique possède deux fonctions distinctes : textes et graphiques. Les caractères sont affichés en mode texte selon un réseau figé de points, alors qu'en mode graphique c'est libre. Pour afficher un caractère il suffit d'envoyer un code au chip contrôleur d'écran et c'est le générateur de caractères qui a la tâche de convertir ce code en un ensemble de pixels affichés par le contrôleur sur l'écran, tandis que la ram vidéo se contente de sauvegarder le code du caractère.
En mode graphique, la ram vidéo est lue directement et c'est le CPU qui vient écrire directement dans la ram vidéo. Inversement il arrive que le CPU lise la ram vidéo, pour identifier un caractère affiché en un certain point de l'écran par exemple. De plus le CPU peut relire la RAM vidéo et sauvegarder son contenu en DRAM avant de charger une nouvelle page, ce qui permettra de restaurer rapidement la page précédente en cas de besoin.

4- Les Disques

4-1- Définition : Les disques sont des mémoires de masse externes, utilisent pour un stockage permanent des fichiers système et utilisateurs.

Les disques dure sont relies au carte mère par des adaptateurs (contrôleurs) parallèle ou série.

La surface des disques est organisé en bloques reconnu par des numéros séquentiel.

4-2- Système de fichiers

L'organisation de ces bloques fat la déférence entres les systèmes des fichiers.

4-3- Le système de fichiers FAT

Un **système de fichiers FAT** est un type spécifique de système de fichiers architecture et une famille de systèmes de fichiers standard de l'industrie

Le système de fichiers FAT est un système de fichiers existant qui est simple et robuste. Il offre de bonnes performances, implémentations légers. Il est, cependant, pris en charge pour des raisons de compatibilité par les développement actuelle des systèmes d'exploitation pour ordinateurs personnels et de nombreux ordinateurs personnels , les appareils mobiles et les systèmes embarqués , et est donc un format bien adapté pour l' échange de données entre les ordinateurs et les périphériques

Présentation de FAT:

Le nom du système de fichiers provient de premier plan de l'utilisation du système de fichiers d'une table d'index, le tableau d'allocation de fichiers , statiquement alloué au moment de la mise en forme. Le tableau contient des entrées pour chaque pôle , une zone contiguë de stockage sur disque. Chaque entrée contient soit le numéro de la grappe suivante dans le fichier, ou bien un marqueur indiquant la fin du fichier, l'espace disque inutilisé, ou des zones réservées spéciales du disque. Le répertoire racine du disque contient le numéro du premier groupe de chaque fichier dans ce répertoire; le système d'exploitation peut alors parcourir la table FAT, en regardant le nombre de cluster de chaque partie successive du fichier de disque comme une chaîne de cluster jusqu'à la fin du fichier est atteinte. De la même manière, les sous-répertoires sont mis en œuvre sous forme de fichiers spéciaux contenant les entrées du répertoire de leurs fichiers respectifs.

Initialement conçu comme un système de fichiers de 8 bits, le nombre maximal de clusters a été augmenté de façon significative en tant que lecteurs de disques ont évolué, et ainsi le nombre de bits utilisés pour identifier chaque groupe a augmenté. Les principales versions

successives du format FAT sont nommés d' après le nombre de bits d'éléments de table: 12 (FAT12), 16 (FAT16), et 32 (FAT32). Sauf pour l'original FAT 8-bit précurseur, chacune de ces variantes est encore en usage. La norme FAT a également été élargie à d'autres égards, tout en préservant généralement la compatibilité ascendante avec les logiciels existant

✓ Un système de fichiers FAT est composé de quatre sections différentes:

- Les secteurs réservés, situé au tout début.

  Le premier secteur réservé (secteur logique 0) est le secteur de démarrage (aussi appelé Volume Boot Record ou simplement VBR). Il comprend une zone appelée le BIOS Parameter Block (BPB) , qui contient des informations sur le système de fichiers de base, en particulier de son type et des pointeurs vers l'emplacement des autres sections, et contient généralement le système d'exploitation boot loader code. Informations importantes du secteur de démarrage est accessible par une structure du système d'exploitation appelé le lecteur bloc de paramètres (DPB) sous DOS et OS

  Le nombre total de secteurs réservés est indiquée par un champ à l'intérieur du secteur de démarrage, et est généralement 32 sur les systèmes de fichiers FAT32

  Pour les systèmes de fichiers FAT32, les secteurs réservés comprennent un système de fichiers secteur de l' information au secteur logique 1 et une sauvegarde du secteur de démarrage au secteur logique 6.

  Alors que de nombreux autres fournisseurs ont continué d'utiliser une configuration mono-secteur (secteur logique 0 seulement) pour le chargeur de démarrage, démarrage du code du secteur de Microsoft a augmenté à étendre sur des secteurs logiques 0 et 2 depuis l'introduction de FAT32, avec le secteur logique 0 en fonction de sous-routines dans le secteur logique 2. La zone de sauvegarde du secteur de démarrage se compose de trois secteurs logiques 6, 7 et 8 ainsi. Dans certains cas, Microsoft utilise également le secteur 12 de la zone des secteurs réservés pour un chargeur de démarrage étendu.

- La région FAT.

  Cette région contient généralement deux copies (peut varier) de la table d'allocation de fichiers pour des raisons de contrôle de redondance, bien que rarement utilisé, même par les services de réparation de disque.

  Ce sont des cartes de la région de données, indiquant quels groupes sont utilisés par les fichiers et répertoires.

Chapitre 2

Assembleur : GNU/AS

L'Assembleur GNU Sous un système Linux sur Intel 80386

1. Introduction

L'assembleur est un langage dit **bas niveau**, c'est-à-dire qu'il est très proche du langage machine.

Autrement dit, pour programmer en assembleur vous devez :

- apprendre une architecture : Intel par exemple ;
- avoir quelques connaissances basiques sur les systèmes d'exploitation : Linux par exemple ;
- maîtriser un assembleur : l'assembleur GNU par exemple.

Apprendre une architecture, c'est comprendre le fonctionnement d'un processeur : les registres, l'adressage et l'organisation de la mémoire, les interruptions… et tout ce que vous avez appris dans le cours d'architecture des ordinateurs. Vous avez, sans doute, une idée claire et suffisante sur l'architecture Intel (IA ou x86) pour aborder ce tutoriel. D'autre part, apprendre un assembleur c'est apprendre une syntaxe pour programmer. C'est l'objectif de ce tutoriel !

Le langage assembleur, ou simplement l'assembleur, est une représentation symbolique du langage machine (données binaires et instructions du processeur). Il existe deux syntaxes d'assembleurs :

- l'assembleur **Intel** : l'assembleur principal utilisant cette syntaxe est NASM ;
- l'assembleur **AT&T** : l'assembleur principal est l'assembleur GNU ou simplement as.

2. L'assemblage et l'édition de liens sous Linux

Pour générer un fichier exécutable à partir du code source, nous utiliserons un programme dit **assembleur** et un programme dit **éditeur de liens** (linker). Sous un système GNU/Linux, l'assembleur est le programme **as** et l'éditeur de liens est le programme **ld** (loader).

Le schéma suivant montre les étapes à suivre pour générer l'exécutable :

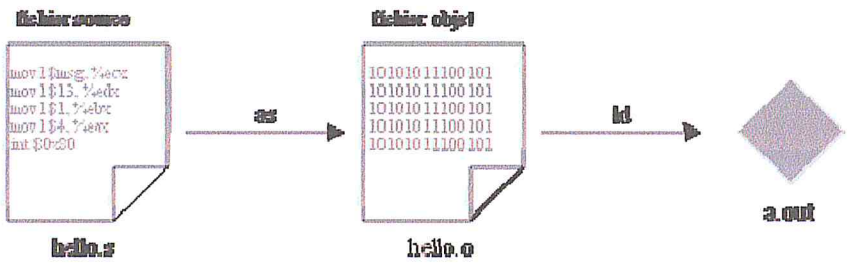


Fig II.1 Les étapes d'assemblage

L'assembleur **as** prend comme entrée le fichier nom.s, ayant obligatoirement l'extension « .s », pour générer le fichier objet nom.o. À son tour, l'éditeur de liens va lier les différents morceaux du code objet et affecter à chacun son adresse d'exécution (run-time address) et produire l'exécutable. Celui-ci est au format ELF (Executable Linkable Format). C'est le format des fichiers binaires utilisés par les systèmes UNIX. C'est l'alternative à l'ancien format a.out (assembler output). Notre makefile contiendra les commandes (rules) suivantes :
Important: Le symbole global _start est le point d'entrée par défaut de l'éditeur de liens. Il symbolise l'adresse à partir de laquelle le processeur commencera le fetch des instructions, lorsque le programme sera chargé en mémoire et exécuté.

3. Syntaxe de l'assembleur

A. Les sections : Un programme écrit en assembleur GNU peut être divisé en trois sections. Trois directives assembleur sont réservés pour déclarer ces sections :

1. .text (read only) : la section du code. Elle contient les instructions et les constantes du programme. Un programme assembleur doit contenir au moins la section .text ;
2. .data (read-write) : la section des données (data section). Elle décrit comment allouer l'espace mémoire pour les variables initialisables du programme (variables globales) ;
3. .bss (read-write) : contient les variables non initialisées. Dans notre code, cette section est vide. On peut donc l'éliminer.

B. Les commentaires

Il existe deux méthodes pour commenter un code source. Les commentaires écrits entre /* et */ peuvent occuper plusieurs lignes. Un commentaire préfixé par un # ne peut occuper qu'une seule ligne.

C. Les déclarations

Les déclarations peuvent prendre quatre formats :

1. .nom directive
2. label_1 : .nom directive attribut
3. label_2 :
4. expression
5. instruction op1 , op2 , ...

Une déclaration se termine par le caractère saut de ligne \n ou par le caractère « ; ».
Une déclaration peut commencer par une étiquette (label). Une étiquette peut être suivie d'un symbole clé qui détermine le type de la déclaration. Si le symbole clé est préfixé par un point,

alors la déclaration est une directive assembleur. Les attributs d'une directive peuvent être un symbole prédéfini, une constante ou une expression.

D. Les symboles

Un **symbole** est une séquence de caractères choisis parmi :

1. Les lettres de l'alphabet : a .. z, A .. Z ;
2. Les chiffres décimaux : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ;
3. Le nom du symbole d'une étiquette doit être unique dans un programme. Autrement dit, vous ne devez jamais utiliser le même symbole pour représenter deux locations (étiquettes) différentes.

Important:

- Un symbole ne doit jamais commencer par un chiffre. De plus, la casse est significative. Ainsi, on peut utiliser les symboles msg, Msg et MSg dans le même programme : ils représentent des locations différentes dans la mémoire.
- L'assembleur GNU réserve un ensemble de symboles clés pour les directives.

Une **étiquette** est un symbole suivi, immédiatement, du caractère « : ». Elle a deux usages :

1. Si l'étiquette est écrite dans la section du code, alors le symbole représente une valeur du compteur programme.
2. Si l'étiquette est écrite dans la section de données (.data), alors le symbole représente une adresse dans la zone mémoire des données. Par exemple, le symbole msg représente l'adresse du premier octet (code ASCII) de la chaîne de caractères d'un message.

D-1. Le symbole « . »

Le symbole spécial « . » peut être utilisé comme une référence à une adresse au moment de l'assemblage. L'expression suivante permet de calculer précisément len :

1. len = . - msg

Ici, le symbole « . » fait référence à l'adresse de l'octet situé juste après le message dans la zone mémoire des données.

E. Les constantes

Les nombres entiers :

Un nombre entier peut être représenté dans plusieurs systèmes de numération. L'assembleur GNU en identifie quatre : le binaire, le décimal, l'octal et l'hexadécimal.

1. Un nombre **binaire** est un 0b ou 0B suivi de zéro ou plusieurs chiffres binaires {0, 1}.
   Exemple : 0b10011.
2. Un nombre **octal** est un 0 suivi de zéro ou plusieurs chiffres octaux {0, 1, 2, 3, 4, 5, 6, 7}.
   Exemple : 09122.

3. Un nombre **décimal** ne doit pas commencer par 0. Il contient zéro ou plusieurs chiffres décimaux {0..9}.
   Exemple : 97340.
4. Un nombre **hexadécimal** est un 0x ou 0X suivi de zéro ou plusieurs chiffres hexadécimaux {0..9, A, B, C, D, E, F}.
   Exemple : 0x6FC9D.

Les caractères

Comme le montre le , chaque caractère est identifié par son code ASCII. Dans un code assembleur, on définit un caractère en écrivant son code ASCII approprié. En utilisant, la syntaxe de l'assembleur GNU, un caractère peut être écrit comme une apostrophe suivie immédiatement de son symbole. Exemple :

- **'A** est le code ASCII **65** de **A** ;
- **'9** désigne le code ASCII **100** de **9**.

Les chaînes de caractères

Une chaîne de caractères (string) est une séquence de caractères écrite entre guillemets. Elle représente un tableau contigu d'octets en mémoire.

```
msg :           .asciz "Bonjour !\n"
```

Le caractère n, lorsqu'il est préfixé par un antislash, est équivalent au code ASCII (10) du caractère saut de ligne.

F. Les expressions

Une expression spécifie une adresse ou une valeur numérique. Une expression entière est un ou plusieurs arguments délimités par des opérateurs. Les arguments sont des symboles, des chiffres ou des sous-expressions. Une sous-expression est une expression écrite entre deux parenthèses. Les opérateurs peuvent être des préfixes ou des infixes :

Préfixes

Les opérateurs préfixes prennent un argument absolu. as propose deux opérateurs préfixes :

- L'opérateur ~ complément bit-à-bit (bitwise not).
  Exemple : ~ 0b10001110 = 10001110 = 01110001 ;
- L'opérateur négation - (complément à deux) : −N = ~ N + 1.
  Exemple : N = 142 = 0b10001110, −142 = 01110001 + 1 = 01110010.

Infixes

Un opérateur de type infixe prend deux arguments : *, /, %, <, <<, >, >>, +, −, == …

Exemples d'expressions :

1. len = . - msg
2. SYSSIZE = 0x80000
3. SYSSEG = 0x1000
4. ENDSEG = SYSSEG + SYSSIZE # ENDSEG = 0x81000

G. Syntaxe des instructions 80386 (IA-32)

Les opérandes

- **Les registres** : les opérandes de type registre doivent être préfixés par un **%**.
  Exemples : %eax, %ax, %al, %ah, %ebx, %ecx, %edx, %esp, %ebp, %esi, %edi…
- **Les opérandes immédiats** : les opérandes de ce type doivent être préfixés par un **$**.
  Exemples : $4, $0x79ff03C3, $0b10110, $07621, $msg…
- **Les opérandes mémoire** : lorsqu'un opérande réside dans le segment de données du programme, le processeur doit calculer son adresse effective (EA : Effective Address) en se basant sur l'expression suivante : **EA = base + scale * index + disp**.
  L'assembleur GNU utilise la syntaxe AT &T qui traduit l'expression précédente en celle-ci :
  EA = disp (base, index, scale) avec :
  - **disp** : un déplacement facultatif. disp peut être un symbole ou un entier signé ;
  - **scale** : un scalaire qui multiplie index. Il peut avoir la valeur 1, 2, 4 ou 6. Si le scale n'est pas spécifié, il est substitué par 1 ;
  - **base** : un registre 32 bits optionnel. Souvent, on utilise EBX et EBP (si l'opérande est dans la pile) comme base ;
  - **index** : un registre 32 bits optionnel. Souvent, on utilise ESI et EDI comme index.

La taille des opérandes peut être codée dans le dernier caractère de l'instruction : **b** indique une taille de 8 bits, **w** indique une taille de 16 bits et **l** indique une taille de 32 bits.

Exemples :

1. movb $0x4f,%al # 8-bit
2. movw $0x4ffc,%ax # 16-bit
3. movl $0x9cff83ec,%eax # 32-bit

L'ordre des opérandes

L'assembleur GNU utilise la syntaxe AT&T. Ainsi, une instruction mov doit être écrite comme suit :

mov source,destination

Les instructions LJMP/LCALL

Les instructions LJMP (Long Jump) et LCALL (Long Call) permettent de transférer le contrôle à un autre programme situé dans un autre segment (autre que celui où elles sont écrites !). La syntaxe de ces deux instructions est :

```
Ljmp    $segment,$offset

lcall   $segment,$offset
```

Le premier opérande de chaque instruction sera chargé dans le registre CS et le deuxième sera chargé dans le registre EIP (ou IP en mode réel !).

H. Les directives

Les directives sont des pseudo-opérations. Elles sont utilisées pour simplifier, pour le programmeur, quelques opérations complexes telles que l'allocation de la mémoire. Comme on l'a dit dans la section précédente, le nom d'une directive est un symbole clé préfixé par un point.

Il existe plus de 70 directives définies par l'assembleur GNU. Les plus utilisées sont :

.ascii "string_1", "string_2" ...

Définit zéro ou plusieurs chaînes de caractères séparées par des virgules.

```
1. msg1 :          .ascii "Hello, World !\ n"
2. bootMsg :       .ascii "Loading system ..."
```

.asciz "string_1", "string_2" ...

Définit zéro ou plusieurs chaînes de caractères séparées par des virgules et dont chacune se termine par le caractère '**\0**.

Ainsi, ces deux déclarations sont équivalentes :

```
1. msg1 :          .ascii "Hello, World !\0"
2. msg2 :          .asciz "Hello, World !"
```

.byte expression_1, expression_2 ...

Définit et initialise un tableau d'octets.

```
tab_byte :         .byte 23,'%,0xff,9,'\b
```

Le premier élément est le nombre 23 et il est situé à l'adresse tab_byte, suivi par le code ASCII du caractère **%**...

.word expression_1, expression_2 ...

Définit et initialise un tableau de mots binaires (16 bits ou word).

```
1. descrp :        .word   0x07ff
2.                 .word   0x0000
3.                 .word   0x9200
4.                 .word   0x00C0
```

.quad expression_1, expression_2 ...

Définit et initialise un tableau de quad-mots.

```
_gdt :          .quad   0x0000000000000000

1.              .quad   0x00c09A00000007ff
2.              .quad   0x00C09200000007ff
3.              .quad   0x0000000000000000
```

Un .quad est équivalent à quatre .word successifs.

.int expression_1, expression_2 ...

Définit et initialise un tableau d'entiers (quatre octets).

```
tab_int :        .int 3*6,16,190,-122
```

.long expression_1, expression_2 ...

Définit et initialise un tableau d'entiers (quatre octets).

```
matrix3_3 :      .long 22,36,9
```

```
1.               .long 10,91,0
2.               .long 20,15,1
```

.fill repeat, size, value

Réserve repeat adresses contiguës dans la mémoire et les charge avec les valeurs value de taille size.

```
1.  idt :        .fill 256,8,0
```

Définit un tableau contenant 256 quad-mots à l'adresse idt. Chaque élément du tableau contient zéro comme valeur.

.org new-lc, fill

Avance le compteur location de la section courante à l'adresse new-lc.

```
.text

1.          .global    _start
2.  _start :
3.          .org 510,0
4.          .word 0xAA55
```

Avance le compteur location (registre EIP) de la section texte (code segment) à l'adresse 510 puis écrit le mot 0xAA55. Les octets 0..509 seront chargés par des zéros.

.lcomm symbol, length

Déclare un symbole local et lui attribue length octets sans les initialiser.

```
.bss

1.          .lcomm buffer,1024
```

.global symbol

Déclare un symbole global et visible pour l'éditeur de liens ld.

```
.global    _start,main
```

Par défaut, l'éditeur de liens ld reconnaît le symbole global _start comme point d'entrée à l'édition de liens.

Important [9] : Un symbole global déclaré dans un fichier source peut être utilisé dans un autre fichier sans le redéclarer.

.set symbol, expression et .equ symbol, expression

Ces deux directives sont similaires à l'expression : symbol = expression.

```
.set SYSSIZE,0x80000

1.          .equ SYSSEG,0x1000
```

1- Introduction :

La conception d'un système d'exploitation consiste à mettre en œuvre plusieurs parties, une vision théorique sur les systèmes une réalité technique sur la machine et les outils disponible pour la réalisation.

2- Diagramme de classe :

Le diagramme de classes représente l'architecture conceptuelle du système.

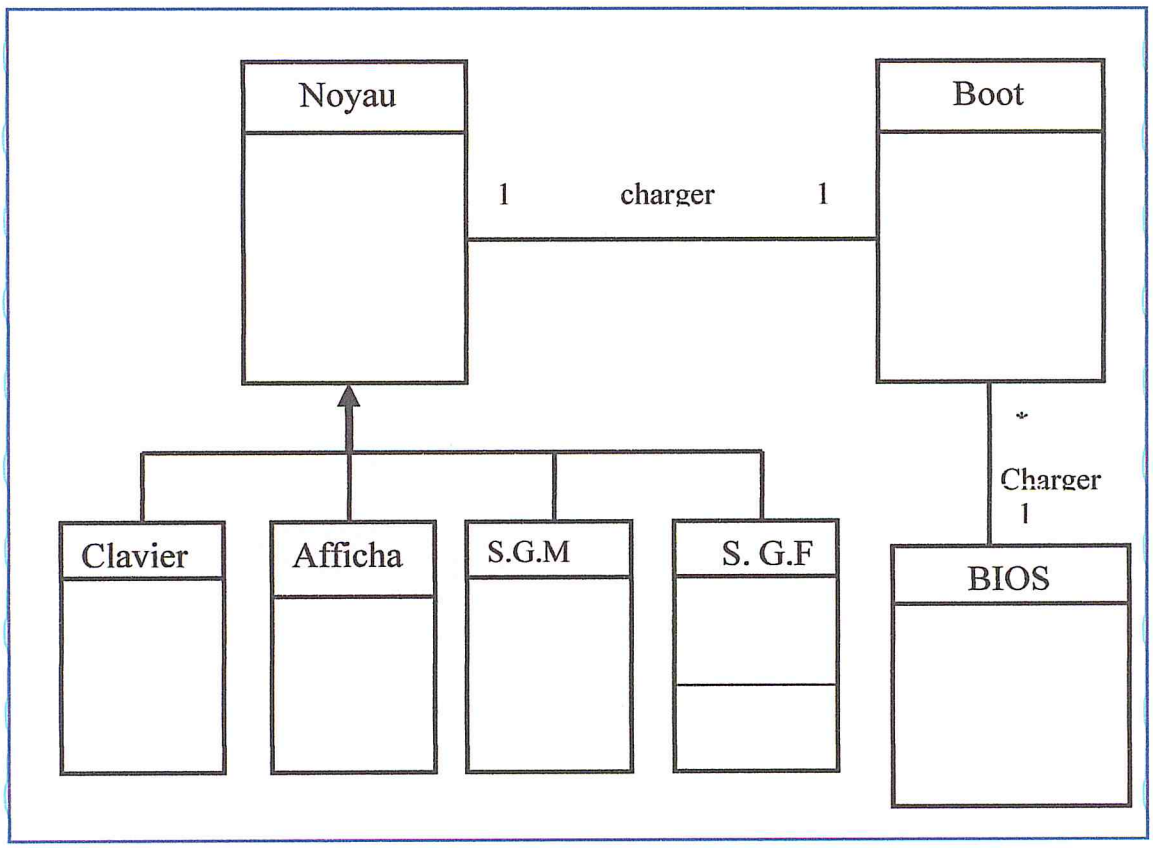


Fig III.1 Diagramme des classes

Scenario:

Le Bios : est lancé par l'interruption reset
Vérifier la Machine (entrées/sorties de base)

Charger le Boot

Lancer le Boot

Le Boot : active le mode protégé

Charger le Noyau du système

Lancer le Noyau

Initialisation de toutes les parties du système

Les contraintes techniques imposées sur le programme boot sont :

- La taille du programme est égale exactement à 512 octets
- Le programme est en mode réelle (16 bits)
- Le programme se termine par les deux octets 0x55 ; 0xaa
- La première partie donne les informations sur le système de fichier utilisé pour le disque de démarrage.

3- Le programme "boot.s"

```
.code16
 .text
 .global _start

 _start :
      jmp start1
      nop
L3:     .byte 0x4d, 0x53, 0x44, 0x4f, 0x53, 0x35, 0x2e, 0x30  #MSDOS5.0
#BIOS Parametres (25)
dap:
L0b:    .byte 0x00, 0x02        #byte per sector
L0d:    .byte 0x08              #Sector per Cluster
L0e:    .byte 0x02, 0x00        # Reseved Sector
L10:    .byte 0x02              # Number of file allocation table
L11:    .byte 0x00, 0x02        # root Entrie
L13:    .byte 0x00, 0x00        # Small Sector
L15:    .byte 0xf8              # Media type  'f8=Hard disk'
L16:    .byte 0x00, 0x00        # Sector par file allocation table
L18:    .byte 0x3f, 0x00        # Sector per track
L1a:    .byte 0x01, 0x00        #number of heads
L1c:    .byte 0x08, 0x00, 0x00, 0x00          # Hidden Sector
L20:    .byte 0x00, 0x00, 0x00, 0x00          # Large Sector
```

```
#Extende Bios parametres (26)
L24:    .byte 0x80              #Physical disk number
L25:    .byte 0x00              #Current head. not used by FAT
L26:    .byte 0x28              #Signature. Must be 0x28 or 0x29 reconized by NT
L27:    .byte 0x00, 0xff, 0x7f, 0xee    #Volume serial number
L2b:    .ascii "Linux Boot "            #Volume Label 11 bytes
L36:    .ascii "FAT32"                  #system ID 7 bytes 'FAT32   '
L3b:    .byte 0x00, 0x00, 0x00
#L3e:   .org 0x3e
start1: cli
        mov     %cs,%ax
        mov     %ax,%ds     # DS = 0x7c00
        movw    0x8000,%ax
        mov     %ax,%es     # ES = 0x8000
        mov     %ax,%ss     # SS = 0x8000

        movw    $MSG_SIZE, %cx          # taille du message
        movb    $0, %bh                 # page 0
        movb    $0x97, %bl              # coleur: background = noir  0x4
                                        #         foreground = blanc 0x9
        movw    $msg2, %bp              # [ES:BP]: offset du message
        movb    $0x13, %ah              # afficher une chaîne de caractères
        movb    $0x01, %al              # déplacer le curseur
        int     $0x10
end0:   jmp     end0

        mov     $0x100,%sp  # TOS = 256 (512 octets)
        movb    $0x42,%ah   # fonction 0x42 : lecture étendue des secteurs
        movw    $dap,%si    # SI = dap : pointeur vers la structure
        int     $0x13
      #ljmp    $0x900,$0x0  # JMP vers [segment:offset] =
                            # 0x900 x 0x16 + 0x0 = 0x9000
```

```
        .org 0x170
msg2:   .byte 0xff,0x0d,0x0a
        .ascii " Le système démmare…V0.2   "
MSG_SIZE = .-msg2
#       .ascii "NTLDR      "
        .org 0x1ae
        .ascii "Retirez les disques"
        .byte 0xff,0x0d,0x0a
        .org 0x1c4
        .ascii "Err. disque"
        .byte 0xff,0x0d,0x0a
        .org 0x1d2
        .ascii "pressez une touche pour demarrer"
        .byte 0x0d,0x0a
      .org 0x1fe
      .byte   0x55, 0xaa
```

4- Sous-système gestion de la mémoire

4-1- Mémoire virtuelle et mémoire physique

Présentation

Nous avons déjà vu que dans le système de segmentation, une adresse physique est calculée à partir d'un sélecteur de segment et d'un offset. Avec la pagination activée, l'adresse obtenue par la segmentation est une **adresse linéaire** qui sera ensuite traduite en une **adresse physique** :

Fig III.2 Conversion d'adresse

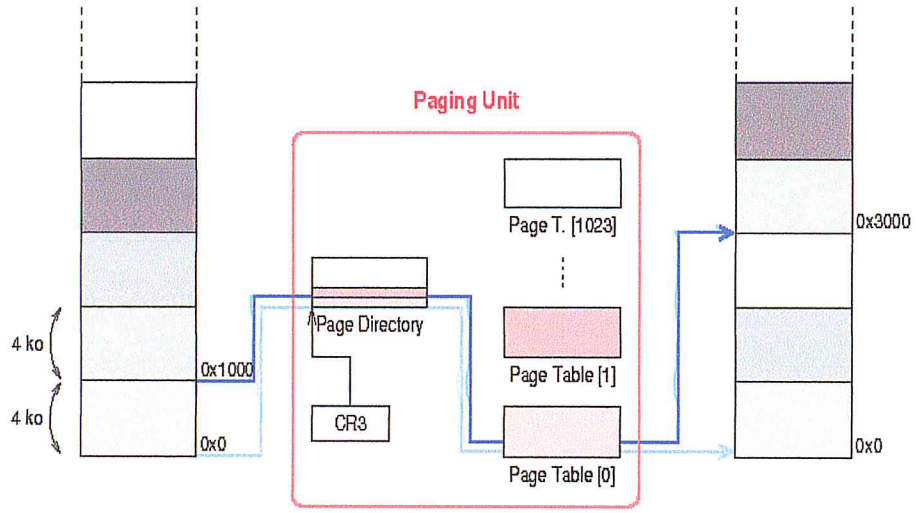
La pagination est un mécanisme d'adressage de la mémoire qui permet :

- de s'affranchir des limites en mémoire de l'ordinateur en utilisant le disque dur comme extension
- d'affecter à chaque tâche son propre espace d'adressage
- d'allouer et de désallouer dynamiquement de la mémoire de façon simple

Quand la pagination est activée, le processeur découpe l'espace d'adressage linéaire (ou virtuel) et la mémoire physique en pages de taille fixe (généralement 4ko ou 4Mo) qui peuvent être associées librement. La traduction des adresses linéaires en adresses physiques est réalisée par le processeur grâce à plusieurs structures :

- le **répertoire de pages (Page Directory)** est un tableau dont les entrées pointent vers des tables de pages. Son adresse est stockée dans le registre **CR3**
- les **tables de pages (Page Table)** sont des tableaux dont les entrées sont des pointeurs vers des pages

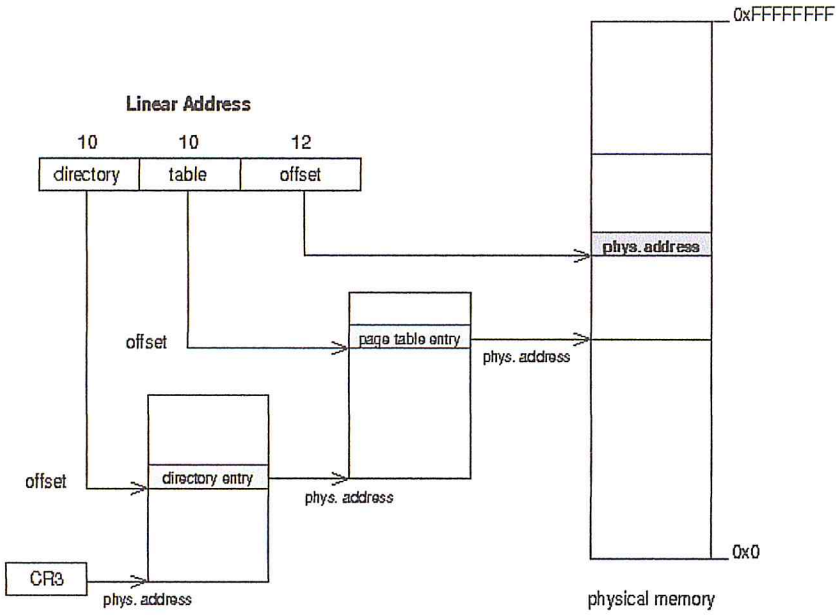
Le schéma ci-dessous illustre l'association entre des pages en mémoire linéaire (ou virtuelle) et en mémoire physique par l'unité de pagination de la **MMU (Memory Management Unit)** :



Le fonctionnement

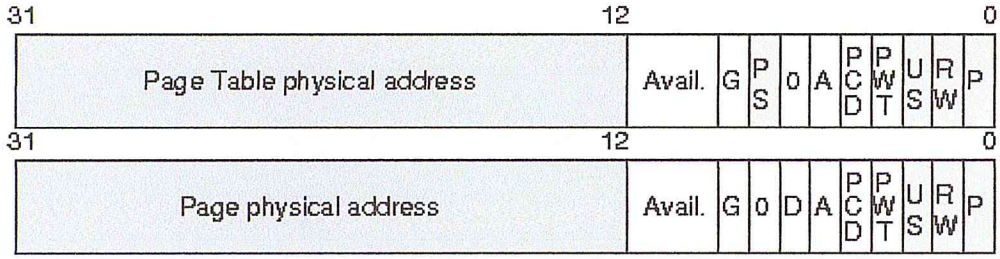
La traduction d'une adresse linéaire en une adresse physique se fait en plusieurs étapes :

1. le processeur utilise le registre CR3 pour connaître l'adresse physique du répertoire de pages
2. les 10 premiers bits de l'adresse linéaire forment un offset, entre `0` et `1023`, qui pointe sur une entrée de ce répertoire de pages
3. cette entrée du répertoire contient l'adresse physique d'une table de pages
4. les 10 bits suivants de l'adresse linéaire forment un offset qui pointe sur une entrée de cette table de pages
5. l'entrée sélectionnée dans la table de page pointe sur une page de 4ko
6. les 12 derniers bits de l'adresse linéaire forment un offset, d'une valeur comprise entre `0` et `4095`. Ce déplacement permet de pointer sur une adresse précise en mémoire.



Les entrées des répertoires et des tables de pages

Les entrées de ces deux tableaux se ressemblent beaucoup. Seuls les champs en grisé seront utilisés dans nos premières implémentations :



- le bit **P** indique si la page ou la table pointée est en mémoire physique
- le bit **R/W** indique si la page ou la table est accessible en écriture (bit à 1)
- le bit **U/S** est à 1 pour permettre l'accès aux tâches non-privilégiées
- le bit **PWT** et le bit **PCD** sont liés à la gestion du cache des pages, que nous ne verrons pas pour le moment
- le bit **A** indique si la page ou la table a été accédée
- le bit **D** (table de pages seulement) indique si la page a été écrite
- le bit **PS** (répertoire de pages seulement) indique la taille des pages (bit à 0 pour 4ko et à 1 pour 4Mo)
- le bit **G** est lié à la gestion du cache des pages
- le champ **Avail.** est librement utilisable

On peut noter que les adresses physiques du répertoire de pages ou de la table des pages sont... sur 20 bits ! En fait, ces adresses doivent être alignées sur 4ko, ce qui signifie que les 12 bits de poids faible doivent être à 0. Le processeur forme cette adresse à partir des 20 bits de poids fort et la complète avec 12 bits à 0. Nous avions déja vu ce type de mécanisme avec les sélecteurs de segment de la GDT.

Calcule arithmétique

- Un répertoire ou une table de pages occupent en mémoire 1024*4 = 4096 octets = 4k
- Une table de pages peut adresser en tout 1024 * 4k = 4 Mo
- Un répertoire de pages peut adresser en tout 1024 * (1024 * 4k) = 4 Go

Notes sur le cache

Le **TLB (Translation Lookaside Buffer)** est un cache des répertoires et des tables des pages utilisés afin d'accélérer la traduction d'adresse. Quand un répertoire ou une table des pages est modifiée, les pages concernées doivent être immédiatement invalidées dans le TLB. Dans un premier temps, nous n'aurons pas besoin de nous préoccuper de ces détails.

Activer la pagination

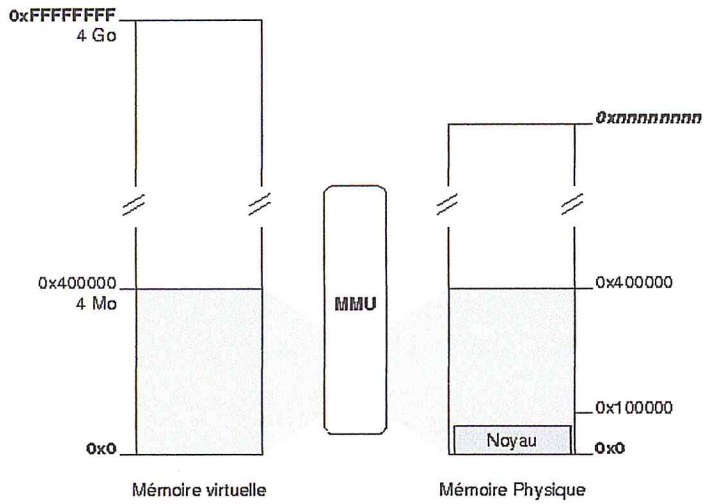
Pour activer la pagination, il suffit de passer le bit 31 du registre CR0 à 1 :

```
asm("  mov %%cr0, %%eax; \
       or %1, %%eax;     \
       mov %%eax, %%cr0" \
       :: "i"(0x80000000));
```

Il faut au préalable avoir initialisé un répertoire et au moins une table des pages !

Implémentation de la pagination

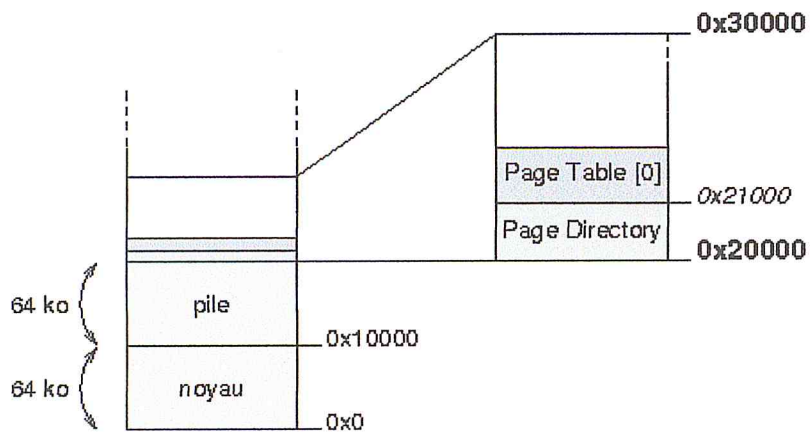
Pour une première implémentation, nous n'allons pas créer de tâche utilisateur. La pagination s'appliquera donc uniquement au noyau.
Dans cette première implémentation, nous allons activer la pagination pour le noyau tel que les 4 premiers Mo de mémoire virtuelle coïncident avec les 4 premiers Mo de mémoire physique :



Ce modèle est simple : la première page en mémoire virtuelle correspond à la première page en mémoire physique, la deuxième page en mémoire virtuelle correspond à la deuxième en mémoire physique et ainsi de suite...

Nous avons vu dans les chapitres précédent que notre noyau est tout petit et qu'il loge dans moins de 64 ko, nous allons adopter l'organisation suivante :

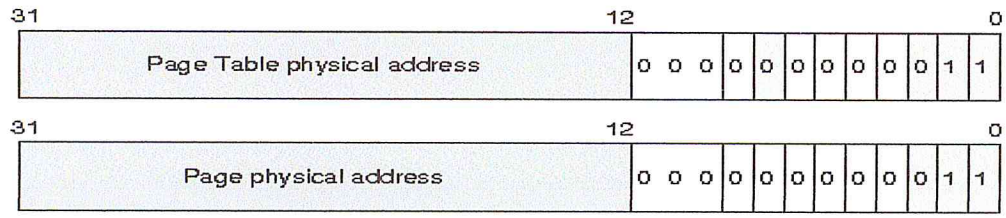
- le code et les données du noyau occupent l'espace entre l'adresse `0x0` et `0x10000`
- la pile du noyau utilise l'espace entre l'adresse `0x10000` et `0x20000`
- le répertoire de pages sera chargé à l'adresse `0x20000`
- pour adresser 4 Mo, une seule table de pages suffit. Elle sera en `0x21000`



Initialiser le répertoire et les tables de pages

Les entrées du répertoire et de la table seront initialisés avec les valeurs suivantes :

- Le bit 0 et le bit 1 mis à 1 indiquent que les pages / tables pointées sont présentes en mémoire et qu'elles sont en lecture et en écriture.
- Le bit 2 est à 0 pour indiquer que leur accès est réservé au noyau.
- Le bit 7 du répertoire est à 0 pour indiquer que les pages font 4ko.



La fonction `init_mm()` initialise le répertoire et les tables de pages du noyau et active la pagination :

```
#include "types.h"

#define PAGING_FLAG     0x80000000  /* CR0 - bit 31 */
#define PD0_ADDR 0x20000 /* addr. page directory
kernel */
#define PT0_ADDR 0x21000 /* addr. page table[0] kernel
*/

/* cree un mapping tel que vaddr = paddr sur 4Mo */
void init_mm(void)
{
    u32 *pd0;    /* kernel page directory */
    u32 *pt0;    /* kernel page table */
    u32 page_addr;
    int i;

    /* Creation du Page Directory */
    pd0 = (u32 *) PD0_ADDR;
    pd0[0] = PT0_ADDR;
    pd0[0] |= 3;
    for (i = 1; i < 1024; i++)
        pd0[i] = 0;


    /* Creation de la Page Table[0] */
    pt0 = (u32 *) PT0_ADDR;
    page_addr = 0;
```

```
    for (i = 0; i < 1024; i++) {
        pt0[i] = page_addr;
        pt0[i] |= 3;
        page_addr += 4096;
    }

    asm("   mov %0, %%eax     \n \
            mov %%eax, %%cr3 \n \
            mov %%cr0, %%eax \n \
            or %1, %%eax     \n \
            mov %%eax, %%cr0" :: "i"(PD0_ADDR),
"i"(PAGING_FLAG));
}
```

Description du programme :

```
pd0 = (u32 *) PD0_ADDR;
```

Cette instruction fait pointer la variable pd0 sur le répertoire de pages à l'adresse physique 0x20000. Notez que pd0 peut aussi être vu comme un tableau de 1024 entrées.

```
pd0[0] = PT0_ADDR;
pd0[0] |= 3;
```

Ensuite, on initialise la première entrée du répertoire en pd0[0] pour la faire pointer sur la première table de pages. Cette table est située juste après le répertoire, à l'adresse physique 0x21000.

Pourquoi avoir choisi la première page de table et pas une des 1023 autre ? Tout simplement parce que l'un des points clef de notre schéma d'adressage est que l'adresse 0 virtuelle doit correspondre à l'adresse 0 en mémoire physique et ainsi que pour toutes les adresses dans la plage adressée. Le choix d'une autre table aurait été possible, mais ce là aurait généré un décalage entre les adresses en mémoire virtuelle et celles mémoire physique.

Les deux premiers bits sont mis à 1 pour indiquer que les pages / tables pointées sont présentes en mémoire et qu'elles sont accessibles en lecture et en écriture.

```
for (i = 1; i < 1024; i++)
    pd0[i] = 0;
```

Les autres entrées du répertoire de pages sont mises à zéro. On n'utilise donc qu'une seule table de page, ce qui suffit pour adresser 4 Mo.

```
/* Creation de la Page Table[0] */
pt0 = (u32 *) PT0_ADDR;
page_addr = 0;
for (i = 0; i < 1024; i++) {
    pt0[i] = page_addr;
    pt0[i] |= 3;
    page_addr += 4096;
}
```

La table de page est initialisée de façon à ce que chacune de ses entrées pointe sur une page mémoire successive. La première page adresse la mémoire à partir de l'adresse 0 et les autre pages se succèdent.

Une nouvelle exception pour gérer les Page Fault

```
init_idt_desc(0x08, (u32) _asm_exc_PF, INTGATE, &kidt[14]); /* #PF */
```

Lorsque le processeur tente d'accéder à une page qui n'est pas présente en mémoire physique, il déclenche une exception de type **Page Fault**. L'adresse qui a provoqué le défaut de page est stockée dans le registre **CR2**. A partir de cette adresse et selon le contexte d'exécution qui a provoqué l'exception, la routine de traitement devrait au choix :

- terminer le programme en renvoyant une erreur de type segmentation fault
- charger en mémoire la page qui était temporairement stockée sur le disque dur dans la zone de **swap**
- allouer une page

5-Exemple d'un programme « boot » et noyau pour le système « linux »

```
!
! bootsect.s est chargé dans l'adresse 0x7c00 par le bios, il se
déplace en
! 0x9000
! Il charge 'setup' immédiatement après (0x90200), et le système
! En 0x10000, utilisent BIOS interruptions.
!
.globl begtext, begdata, begbss, endtext, enddata, endbss
.text
begtext:
.data
begdata:
.bss
begbss:
.text
SYSSIZE = 0x3000
SETUPLEN = 4                    ! nombre des secteurs pour « setup »
BOOTSEG  = 0x07c0               ! Address d'origine de "boot-sector"
INITSEG  = 0x9000               ! déplacé le "boot" ici
SETUPSEG = 0x9020               ! "setup" commence ici
SYSSEG   = 0x1000               ! "system" chargé dans 0x10000
(65536).
ENDSEG   = SYSSEG + SYSSIZE     ! Fin de chargement

ROOT_DEV = 0x306

entry start
start:
      mov   ax,#BOOTSEG
      mov   ds,ax
      mov   ax,#INITSEG
      mov   es,ax
      mov   cx,#256
      sub   si,si
      sub   di,di
      rep               ! rep movsw "ES[Di]=DS[Si]" deplacement 256
      movsw
      jmpi  go,INITSEG
go:   mov   ax,cs
      mov   ds,ax
      mov   es,ax
! La pille à  0x9ff00>> 9:[ff00]
      mov   ss,ax
      mov   sp,#0xFF00        ! arbitrary value >>512

! charger "setup-sectors" après boot bloc
! Note 'es' déjà def.

load_setup:
      mov   dx,#0x0000        ! drive 0, head 0
      mov   cx,#0x0002        ! sector 2, track 0
      mov   bx,#0x0200        ! address = 512, in INITSEG
      mov   ax,#0x0200+SETUPLEN    ! service 2, nr of sectors
      int   0x13              ! read it
```

```
        jnc   ok_load_setup          ! ok - continue
        mov   dx,#0x0000
        mov   ax,#0x0000        ! reset the diskette
        int   0x13
        j     load_setup

ok_load_setup:

! Get disk drive parameters, specifically nr of sectors/track

        mov   dl,#0x00
        mov   ax,#0x0800        ! AH=8 is get drive parameters
        int   0x13
        mov   ch,#0x00
        seg cs
        mov   sectors,cx
        mov   ax,#INITSEG
        mov   es,ax

! Print message
        mov   ah,#0x03          ! read cursor pos
        xor   bh,bh
        int   0x10
        mov   cx,#24
        mov   bx,#0x0007        ! page 0, attribute 7 (normal)
        mov   bp,#msg1
        mov   ax,#0x1301        ! write string, move cursor
        int   0x10

! we want to load the system (at 0x10000)

        mov   ax,#SYSSEG
        mov   es,ax       ! segment of 0x010000
        call  read_it
        call  kill_motor

! After that we check which root-device to use. If the device is
! defined (!= 0), nothing is done and the given device is used.
! Otherwise, either /dev/PS0 (2,28) or /dev/at0 (2,8), depending
! on the number of sectors that the BIOS reports currently.

        seg cs
        mov   ax,root_dev
        cmp   ax,#0
        jne   root_defined
        seg cs
        mov   bx,sectors
        mov   ax,#0x0208        ! /dev/ps0 - 1.2Mb
        cmp   bx,#15
        je    root_defined
        mov   ax,#0x021c        ! /dev/PS0 - 1.44Mb
        cmp   bx,#18
        je    root_defined
undef_root:
      jmp undef_root
root_defined:
      seg cs
      mov   root_dev,ax
```

```
! after that (everyting loaded), we jump to
! the setup-routine loaded directly after
! the bootblock:

        jmpi  0,SETUPSEG

! This routine loads the system at address 0x10000, making sure
! no 64kB boundaries are crossed. We try to load it as fast as
! possible, loading whole tracks whenever we can.
! in: es - starting address segment (normally 0x1000)
!
sread:      .word 1+SETUPLEN ! sectors read of current track
head: .word 0                ! current head
track:      .word 0                  ! current track

read_it:
        mov ax,es
        test ax,#0x0fff
die:    jne die                 ! es must be at 64kB boundary
        xor bx,bx         ! bx is starting address within segment
rp_read:
        mov ax,es
        cmp ax,#ENDSEG          ! have we loaded all yet?
        jb ok1_read
        ret
ok1_read:
        seg cs
        mov ax,sectors
        sub ax,sread
        mov cx,ax
        shl cx,#9
        add cx,bx
        jnc ok2_read
        je ok2_read
        xor ax,ax
        sub ax,bx
        shr ax,#9
ok2_read:
        call read_track
        mov cx,ax
        add ax,sread
        seg cs
        cmp ax,sectors
        jne ok3_read
        mov ax,#1
        sub ax,head
        jne ok4_read
        inc track
ok4_read:
        mov head,ax
        xor ax,ax
ok3_read:
        mov sread,ax
        shl cx,#9
        add bx,cx
        jnc rp_read
        mov ax,es
        add ax,#0x1000
```

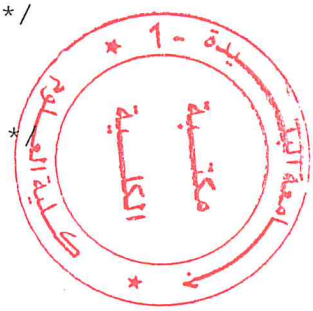
```
        mov es,ax
        xor bx,bx
        jmp rp_read

read_track:
        push ax
        push bx
        push cx
        push dx
        mov dx,track
        mov cx,sread
        inc cx
        mov ch,dl
        mov dx,head
        mov dh,dl
        mov dl,#0
        and dx,#0x0100
        mov ah,#2
        int 0x13
        jc bad_rt
        pop dx
        pop cx
        pop bx
        pop ax
        ret
bad_rt:     mov ax,#0
        mov dx,#0
        int 0x13
        pop dx
        pop cx
        pop bx
        pop ax
        jmp read_track

/* This procedure turns off the floppy drive motor, so
 * that we enter the kernel in a known state, and
 * don't have to worry about it later.
 */
kill_motor:
        push dx
        mov dx,#0x3f2
        mov al,#0
        outb
        pop dx
        ret

sectors:
        .word 0

msg1:
        .byte 13,10
        .ascii "Loading system ..."
        .byte 13,10,13,10
.org 508
root_dev:
        .word ROOT_DEV
boot_flag:
        .word 0xAA55
```

```
/*
 *  linux/boot/head.s
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *  head.s contains the 32-bit startup code.
 *
 * NOTE!!! Startup happens at absolute address 0x00000000, which is
also where
 * the page directory will exist. The startup code will be overwritten
by
 * the page directory.
 */
.text
.globl _idt,_gdt,_pg_dir,_tmp_floppy_area
_pg_dir:
startup_32:
      movl $0x10,%eax
      mov %ax,%ds
      mov %ax,%es
      mov %ax,%fs
      mov %ax,%gs
      lss _stack_start,%esp
      call setup_idt
      call setup_gdt
      movl $0x10,%eax       # reload all the segment registers
      mov %ax,%ds       # after changing gdt. CS was already
      mov %ax,%es       # reloaded in 'setup_gdt'
      mov %ax,%fs
      mov %ax,%gs
      lss _stack_start,%esp
      xorl %eax,%eax
1:    incl %eax         # check that A20 really IS enabled
      movl %eax,0x000000     # loop forever if it isn't
      cmpl %eax,0x100000
      je 1b
/*
 * NOTE! 486 should set bit 16, to check for write-protect in
supervisor
 * mode. Then it would be unnecessary with the "verify_area()"-calls.
 * 486 users probably want to set the NE (#5) bit also, so as to use
 * int 16 for math errors.
 */
      movl %cr0,%eax        # check math chip
      andl $0x80000011,%eax  # Save PG,PE,ET
/* "orl $0x10020,%eax" here for 486 might be good */
      orl $2,%eax       # set MP
      movl %eax,%cr0
      call check_x87
      jmp after_page_tables

/*
 * We depend on ET to be correct. This checks for 287/387.
 */
check_x87:
      fninit
```

```
        fstsw %ax
        cmpb $0,%al
        je 1f           /* no coprocessor: have to set bits */
        movl %cr0,%eax
        xorl $6,%eax            /* reset MP, set EM */
        movl %eax,%cr0
        ret
.align 2
1:      .byte 0xDB,0xE4         /* fsetpm for 287, ignored by 387 */
        ret

/*
 *  setup_idt
 *
 *  sets up a idt with 256 entries pointing to
 *  ignore_int, interrupt gates. It then loads
 *  idt. Everything that wants to install itself
 *  in the idt-table may do so themselves. Interrupts
 *  are enabled elsewhere, when we can be relatively
 *  sure everything is ok. This routine will be over-
 *  written by the page tables.
 */
setup_idt:
        lea ignore_int,%edx
        movl $0x00080000,%eax
        movw %dx,%ax            /* selector = 0x0008 = cs */
        movw $0x8E00,%dx /* interrupt gate - dpl=0, present */

        lea _idt,%edi
        mov $256,%ecx
rp_sidt:
        movl %eax,(%edi)
        movl %edx,4(%edi)
        addl $8,%edi
        dec %ecx
        jne rp_sidt
        lidt idt_descr
        ret

/*
 *  setup_gdt
 *
 *  This routines sets up a new gdt and loads it.
 *  Only two entries are currently built, the same
 *  ones that were built in init.s. The routine
 *  is VERY complicated at two whole lines, so this
 *  rather long comment is certainly needed :-).
 *  This routine will beoverwritten by the page tables.
 */
setup_gdt:
        lgdt gdt_descr
        ret

/*
 * I put the kernel page tables right after the page directory,
 * using 4 of them to span 16 Mb of physical memory. People with
 * more than 16MB will have to expand this.
 */
```

```
.org 0x1000
pg0:

.org 0x2000
pg1:

.org 0x3000
pg2:

.org 0x4000
pg3:

.org 0x5000
/*
 * tmp_floppy_area is used by the floppy-driver when DMA cannot
 * reach to a buffer-block. It needs to be aligned, so that it isn't
 * on a 64kB border.
 */
_tmp_floppy_area:
	.fill 1024,1,0

after_page_tables:
	pushl $0		# These are the parameters to main :-)
	pushl $0
	pushl $0
	pushl $L6		# return address for main, if it decides to.
	pushl $_main
	jmp setup_paging
L6:
	jmp L6			# main should never return here, but
				# just in case, we know what happens.

/* This is the default interrupt "handler" :-) */
int_msg:
	.asciz "Unknown interrupt\n\r"
.align 2
ignore_int:
	pushl %eax
	pushl %ecx
	pushl %edx
	push %ds
	push %es
	push %fs
	movl $0x10,%eax
	mov %ax,%ds
	mov %ax,%es
	mov %ax,%fs
	pushl $int_msg
	call _printk
	popl %eax
	pop %fs
	pop %es
	pop %ds
	popl %edx
	popl %ecx
	popl %eax
	iret
```

```
/*
 * Setup_paging
 *
 * This routine sets up paging by setting the page bit
 * in cr0. The page tables are set up, identity-mapping
 * the first 16MB. The pager assumes that no illegal
 * addresses are produced (ie >4Mb on a 4Mb machine).
 *
 * NOTE! Although all physical memory should be identity
 * mapped by this routine, only the kernel page functions
 * use the >1Mb addresses directly. All "normal" functions
 * use just the lower 1Mb, or the local data space, which
 * will be mapped to some other place - mm keeps track of
 * that.
 *
 * For those with more memory than 16 Mb - tough luck. I've
 * not got it, why should you :-) The source is here. Change
 * it. (Seriously - it shouldn't be too difficult. Mostly
 * change some constants etc. I left it at 16Mb, as my machine
 * even cannot be extended past that (ok, but it was cheap :-)
 * I've tried to show which constants to change by having
 * some kind of marker at them (search for "16Mb"), but I
 * won't guarantee that's all :-( )
 */
.align 2
setup_paging:
      movl $1024*5,%ecx             /* 5 pages - pg_dir+4 page tables */
      xorl %eax,%eax
      xorl %edi,%edi                /* pg_dir is at 0x000 */
      cld;rep;stosl
      movl $pg0+7,_pg_dir           /* set present bit/user r/w */
      movl $pg1+7,_pg_dir+4         /*  --------- " " --------- */
      movl $pg2+7,_pg_dir+8         /*  --------- " " --------- */
      movl $pg3+7,_pg_dir+12        /*  --------- " " --------- */
      movl $pg3+4092,%edi
      movl $0xfff007,%eax           /*  16Mb - 4096 + 7 (r/w user,p) */
      std
1:    stosl             /* fill pages backwards - more efficient :-) */
      subl $0x1000,%eax
      jge 1b
      xorl %eax,%eax          /* pg_dir is at 0x0000 */
      movl %eax,%cr3          /* cr3 - page directory start */
      movl %cr0,%eax
      orl $0x80000000,%eax
      movl %eax,%cr0          /* set paging (PG) bit */
      ret               /* this also flushes prefetch-queue */

.align 2
.word 0
idt_descr:
      .word 256*8-1           # idt contains 256 entries
      .long _idt
.align 2
.word 0
gdt_descr:
      .word 256*8-1           # so does gdt (not that that's any
      .long _gdt        # magic number, but it works for me :^)
```



```
        .align 3
_idt:   .fill 256,8,0           # idt is uninitialized

_gdt:   .quad 0x0000000000000000    /* NULL descriptor */
        .quad 0x00c09a0000000fff    /* 16Mb */
        .quad 0x00c0920000000fff    /* 16Mb */
        .quad 0x0000000000000000    /* TEMPORARY - don't use */
        .fill 252,8,0               /* space for LDT's and TSS's etc */



!
!       setup.s         (C) 1991 Linus Torvalds
!
! setup.s is responsible for getting the system data from the BIOS,
! and putting them into the appropriate places in system memory.
! both setup.s and system has been loaded by the bootblock.
!
! This code asks the bios for memory/disk/other parameters, and
! puts them in a "safe" place: 0x90000-0x901FF, ie where the
! boot-block used to be. It is then up to the protected mode
! system to read them from there before the area is overwritten
! for buffer-blocks.
!

! NOTE! These had better be the same as in bootsect.s!

INITSEG  = 0x9000       ! we move boot here - out of the way
SYSSEG   = 0x1000       ! system loaded at 0x10000 (65536).
SETUPSEG = 0x9020       ! this is the current segment

.globl begtext, begdata, begbss, endtext, enddata, endbss
.text
begtext:
.data
begdata:
.bss
begbss:
.text

entry start
start:

! ok, the read went well so we get current cursor position and save it
for
! posterity.

        mov   ax,#INITSEG! this is done in bootsect already, but...
        mov   ds,ax
        mov   ah,#0x03    ! read cursor pos
        xor   bh,bh
        int   0x10        ! save it in known place, con_init fetches
        mov   [0],dx            ! it from 0x90000.

! Get memory size (extended mem, kB)

        mov   ah,#0x88
```

```
        int   0x15
        mov   [2],ax

! Get video-card data:

        mov   ah,#0x0f
        int   0x10
        mov   [4],bx          ! bh = display page
        mov   [6],ax          ! al = video mode, ah = window width

! check for EGA/VGA and some config parameters

        mov   ah,#0x12
        mov   bl,#0x10
        int   0x10
        mov   [8],ax
        mov   [10],bx
        mov   [12],cx

! Get hd0 data

        mov   ax,#0x0000
        mov   ds,ax
        lds   si,[4*0x41]
        mov   ax,#INITSEG
        mov   es,ax
        mov   di,#0x0080
        mov   cx,#0x10
        rep
        movsb

! Get hd1 data
        mov   ax,#0x0000
        mov   ds,ax
        lds   si,[4*0x46]
        mov   ax,#INITSEG
        mov   es,ax
        mov   di,#0x0090
        mov   cx,#0x10
        rep
        movsb

! Check that there IS a hd1 :-)
        mov   ax,#0x01500
        mov   dl,#0x81
        int   0x13
        jc    no_disk1
        cmp   ah,#3
        je    is_disk1
no_disk1:
        mov   ax,#INITSEG
        mov   es,ax
        mov   di,#0x0090
        mov   cx,#0x10
        mov   ax,#0x00
        rep
        stosb
is_disk1:
```

```
! now we want to move to protected mode ...

      cli             ! no interrupts allowed !

! first we move the system to it's rightful place

      mov   ax,#0x0000
      cld             ! 'direction'=0, movs moves forward
do_move:
      mov   es,ax     ! destination segment
      add   ax,#0x1000
      cmp   ax,#0x9000
      jz    end_move
      mov   ds,ax     ! source segment
      sub   di,di
      sub   si,si
      mov   cx,#0x8000
      rep
      movsw
      jmp   do_move

! then we load the segment descriptors

end_move:
      mov   ax,#SETUPSEG     ! right, forgot this at first. didn't work
:-)
      mov   ds,ax
      lidt  idt_48           ! load idt with 0,0
      lgdt  gdt_48           ! load gdt with whatever appropriate

! that was painless, now we enable A20

      call  empty_8042
      mov   al,#0xD1         ! command write
      out   #0x64,al
      call  empty_8042
      mov   al,#0xDF         ! A20 on
      out   #0x60,al
      call  empty_8042

! well, that went ok, I hope. Now we have to reprogram the interrupts
:-(
! we put them right after the intel-reserved hardware interrupts, at
! int 0x20-0x2F. There they won't mess up anything. Sadly IBM really
! messed this up with the original PC, and they haven't been able to
! rectify it afterwards. Thus the bios puts interrupts at 0x08-0x0f,
! which is used for the internal hardware interrupts as well. We just
! have to reprogram the 8259's, and it isn't fun.

      mov   al,#0x11         ! initialization sequence
      out   #0x20,al         ! send it to 8259A-1
      .word 0x00eb,0x00eb          ! jmp $+2, jmp $+2
      out   #0xA0,al         ! and to 8259A-2
      .word 0x00eb,0x00eb
      mov   al,#0x20         ! start of hardware int's (0x20)
      out   #0x21,al
      .word 0x00eb,0x00eb
      mov   al,#0x28         ! start of hardware int's 2 (0x28)
```

```
        out   #0xA1,al
        .word 0x00eb,0x00eb
        mov   al,#0x04          ! 8259-1 is master
        out   #0x21,al
        .word 0x00eb,0x00eb
        mov   al,#0x02          ! 8259-2 is slave
        out   #0xA1,al
        .word 0x00eb,0x00eb
        mov   al,#0x01          ! 8086 mode for both
        out   #0x21,al
        .word 0x00eb,0x00eb
        out   #0xA1,al
        .word 0x00eb,0x00eb
        mov   al,#0xFF          ! mask off all interrupts for now
        out   #0x21,al
        .word 0x00eb,0x00eb
        out   #0xA1,al

! well, that certainly wasn't fun :-(. Hopefully it works, and we don't
! need no steenking BIOS anyway (except for the initial loading :-).
! The BIOS-routine wants lots of unnecessary data, and it's less
! "interesting" anyway. This is how REAL programmers do it.
!
! Well, now's the time to actually move into protected mode. To make
! things as simple as possible, we do no register set-up or anything,
! we let the gnu-compiled 32-bit programs do that. We just jump to
! absolute address 0x00000, in 32-bit protected mode.
        mov   ax,#0x0001  ! protected mode (PE) bit
        lmsw  ax          ! This is it!
        jmpi  0,8         ! jmp offset 0 of segment 8 (cs)
! This routine checks that the keyboard command queue is empty
! No timeout is used - if this hangs there is something wrong with
! the machine, and we probably couldn't proceed anyway.
empty_8042:
        .word 0x00eb,0x00eb
        in    al,#0x64    ! 8042 status port
        test  al,#2       ! is input buffer full?
        jnz   empty_8042  ! yes - loop
        ret

gdt:
        .word 0,0,0,0           ! dummy
        .word 0x07FF            ! 8Mb - limit=2047 (2048*4096=8Mb)
        .word 0x0000            ! base address=0
        .word 0x9A00            ! code read/exec
        .word 0x00C0            ! granularity=4096, 386

        .word 0x07FF            ! 8Mb - limit=2047 (2048*4096=8Mb)
        .word 0x0000            ! base address=0
        .word 0x9200            ! data read/write
        .word 0x00C0            ! granularity=4096, 386
idt_48:
        .word 0                 ! idt limit=0
        .word 0,0               ! idt base=0L
gdt_48:
        .word 0x800         ! gdt limit=2048, 256 GDT entries
        .word 512+gdt,0x9! gdt base = 0X9xxxx
```

```
/*
 *  linux/init/main.c
 *
 *  (C) 1991  Linus Torvalds
 */

#define __LIBRARY__
#include <unistd.h>
#include <time.h>

/*
 * we need this inline - forking from kernel space will result
 * in NO COPY ON WRITE (!!!), until an execve is executed. This
 * is no problem, but for the stack. This is handled by not letting
 * main() use the stack at all after fork(). Thus, no function
 * calls - which means inline code for fork too, as otherwise we
 * would use the stack upon exit from 'fork()'.
 *
 * Actually only pause and fork are needed inline, so that there
 * won't be any messing with the stack from main(), but we define
 * some others too.
 */
static inline _syscall0(int,fork)
static inline _syscall0(int,pause)
static inline _syscall1(int,setup,void *,BIOS)
static inline _syscall0(int,sync)

#include <linux/tty.h>
#include <linux/sched.h>
#include <linux/head.h>
#include <asm/system.h>
#include <asm/io.h>

#include <stddef.h>
#include <stdarg.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>

#include <linux/fs.h>

static char printbuf[1024];

extern int vsprintf();
extern void init(void);
extern void blk_dev_init(void);
extern void chr_dev_init(void);
extern void hd_init(void);
extern void floppy_init(void);
extern void mem_init(long start, long end);
extern long rd_init(long mem_start, int length);
extern long kernel_mktime(struct tm * tm);
extern long startup_time;

/*
 * This is set up by the setup-routine at boot-time
 */
#define EXT_MEM_K (*(unsigned short *)0x90002)
```

```
#define DRIVE_INFO (*(struct drive_info *)0x90080)
#define ORIG_ROOT_DEV (*(unsigned short *)0x901FC)

/*
 * Yeah, yeah, it's ugly, but I cannot find how to do this correctly
 * and this seems to work. I anybody has more info on the real-time
 * clock I'd be interested. Most of this was trial and error, and some
 * bios-listing reading. Urghh.
 */

#define CMOS_READ(addr) ({ \
outb_p(0x80|addr,0x70); \
inb_p(0x71); \
})

#define BCD_TO_BIN(val) ((val)=((val)&15) + ((val)>>4)*10)

static void time_init(void)
{
      struct tm time;

      do {
            time.tm_sec = CMOS_READ(0);
            time.tm_min = CMOS_READ(2);
            time.tm_hour = CMOS_READ(4);
            time.tm_mday = CMOS_READ(7);
            time.tm_mon = CMOS_READ(8);
            time.tm_year = CMOS_READ(9);
      } while (time.tm_sec != CMOS_READ(0));
      BCD_TO_BIN(time.tm_sec);
      BCD_TO_BIN(time.tm_min);
      BCD_TO_BIN(time.tm_hour);
      BCD_TO_BIN(time.tm_mday);
      BCD_TO_BIN(time.tm_mon);
      BCD_TO_BIN(time.tm_year);
      time.tm_mon--;
      startup_time = kernel_mktime(&time);
}

static long memory_end = 0;
static long buffer_memory_end = 0;
static long main_memory_start = 0;

struct drive_info { char dummy[32]; } drive_info;

void main(void)         /* This really IS void, no error here. */
{                       /* The startup routine assumes (well, ...) this */
/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
      ROOT_DEV = ORIG_ROOT_DEV;
      drive_info = DRIVE_INFO;
      memory_end = (1<<20) + (EXT_MEM_K<<10);
      memory_end &= 0xfffff000;
      if (memory_end > 16*1024*1024)
            memory_end = 16*1024*1024;
      if (memory_end > 12*1024*1024)
```

```
            buffer_memory_end = 4*1024*1024;
        else if (memory_end > 6*1024*1024)
            buffer_memory_end = 2*1024*1024;
        else
            buffer_memory_end = 1*1024*1024;
        main_memory_start = buffer_memory_end;
#ifdef RAMDISK
        main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
#endif
        mem_init(main_memory_start,memory_end);
        trap_init();
        blk_dev_init();
        chr_dev_init();
        tty_init();
        time_init();
        sched_init();
        buffer_init(buffer_memory_end);
        hd_init();
        floppy_init();
        sti();
        move_to_user_mode();
        if (!fork()) {          /* we count on this going ok */
            init();
        }
/*
 *   NOTE!!   For any other task 'pause()' would mean we have to get a
 * signal to awaken, but task0 is the sole exception (see 'schedule()')
 * as task 0 gets activated at every idle moment (when no other tasks
 * can run). For task0 'pause()' just means we go check if some other
 * task can run, and if not we return here.
 */
        for(;;) pause();
}

static int printf(const char *fmt, ...)
{
        va_list args;
        int i;

        va_start(args, fmt);
        write(1,printbuf,i=vsprintf(printbuf, fmt, args));
        va_end(args);
        return i;
}

static char * argv_rc[] = { "/bin/sh", NULL };
static char * envp_rc[] = { "HOME=/", NULL };

static char * argv[] = { "-/bin/sh",NULL };
static char * envp[] = { "HOME=/usr/root", NULL };

void init(void)
{
        int pid,i;

        setup((void *) &drive_info);
        (void) open("/dev/tty0",O_RDWR,0);
        (void) dup(0);
```

```
        (void) dup(0);
        printf("%d buffers = %d bytes buffer space\n\r",NR_BUFFERS,
                NR_BUFFERS*BLOCK_SIZE);
        printf("Free mem: %d bytes\n\r",memory_end-main_memory_start);
        if (!(pid=fork())) {
                close(0);
                if (open("/etc/rc",O_RDONLY,0))
                        _exit(1);
                execve("/bin/sh",argv_rc,envp_rc);
                _exit(2);
        }
        if (pid>0)
                while (pid != wait(&i))
                        /* nothing */;
        while (1) {
                if ((pid=fork())<0) {
                        printf("Fork failed in init\r\n");
                        continue;
                }
                if (!pid) {
                        close(0);close(1);close(2);
                        setsid();
                        (void) open("/dev/tty0",O_RDWR,0);
                        (void) dup(0);
                        (void) dup(0);
                        _exit(execve("/bin/sh",argv,envp));
                }
                while (1)
                        if (pid == wait(&i))
                                break;
                printf("\n\rchild %d died with code %04x\n\r",pid,i);
                sync();
        }
        _exit(0);   /* NOTE! _exit, not exit() */
}
```

```
/*
 *  linux/mm/page.s
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * page.s contains the low-level page-exception code.
 * the real work is done in mm.c
 */

.globl _page_fault

_page_fault:
      xchgl %eax,(%esp)
      pushl %ecx
      pushl %edx
      push %ds
      push %es
      push %fs
      movl $0x10,%edx
      mov %dx,%ds
      mov %dx,%es
      mov %dx,%fs
      movl %cr2,%edx
      pushl %edx
      pushl %eax
      testl $1,%eax
      jne 1f
      call _do_no_page
      jmp 2f
1:    call _do_wp_page
2:    addl $8,%esp
      pop %fs
      pop %es
      pop %ds
      popl %edx
      popl %ecx
      popl %eax
      iret


/*
 *  linux/mm/memory.c
 *  (C) 1991  Linus Torvalds
 * demand-loading started 01.12.91 - seems it is high on the list of
 * things wanted, and it should be easy to implement. - Linus
 * Ok, demand-loading was easy, shared pages a little bit tricker.
Shared
 * pages started 02.12.91, seems to work. - Linus.
 *
 * Tested sharing by executing about 30 /bin/sh: under the old kernel
it
 * would have taken more than the 6M I have free, but it worked well as
 * far as I could see.
 *
 * Also corrected some "invalidate()"s - I wasn't doing enough of them.
 */
```

```
#include <signal.h>
#include <asm/system.h>
#include <linux/sched.h>
#include <linux/head.h>
#include <linux/kernel.h>
volatile void do_exit(long code);

static inline volatile void oom(void)
{
      printk("out of memory\n\r");
      do_exit(SIGSEGV);
}
#define invalidate() \
__asm__("movl %%eax,%%cr3"::"a" (0))

/* these are not to be changed without changing head.s etc */
#define LOW_MEM 0x100000
#define PAGING_MEMORY (15*1024*1024)
#define PAGING_PAGES (PAGING_MEMORY>>12)
#define MAP_NR(addr) (((addr)-LOW_MEM)>>12)
#define USED 100
#define CODE_SPACE(addr) ((((addr)+4095)&~4095) < \
current->start_code + current->end_code)

static long HIGH_MEMORY = 0;

#define copy_page(from,to) \
__asm__("cld ; rep ; movsl"::"S" (from),"D" (to),"c"
(1024):"cx","di","si")

static unsigned char mem_map [ PAGING_PAGES ] = {0,};

/*
 * Get physical address of first (actually last :-) free page, and mark
it
 * used. If no free pages left, return 0.
 */
unsigned long get_free_page(void)
{
register unsigned long __res asm("ax");

__asm__("std ; repne ; scasb\n\t"
      "jne 1f\n\t"
      "movb $1,1(%%edi)\n\t"
      "sall $12,%%ecx\n\t"
      "addl %2,%%ecx\n\t"
      "movl %%ecx,%%edx\n\t"
      "movl $1024,%%ecx\n\t"
      "leal 4092(%%edx),%%edi\n\t"
      "rep ; stosl\n\t"
      "movl %%edx,%%eax\n"
      "1:"
      :"=a" (__res)
      :"0" (0),"i" (LOW_MEM),"c" (PAGING_PAGES),
      "D" (mem_map+PAGING_PAGES-1)
      :"di","cx","dx");
return __res;
}
```

```
/*
 * Free a page of memory at physical address 'addr'. Used by
 * 'free_page_tables()'
 */
void free_page(unsigned long addr)
{
      if (addr < LOW_MEM) return;
      if (addr >= HIGH_MEMORY)
            panic("trying to free nonexistent page");
      addr -= LOW_MEM;
      addr >>= 12;
      if (mem_map[addr]--) return;
      mem_map[addr]=0;
      panic("trying to free free page");
}

/*
 * This function frees a continuos block of page tables, as needed
 * by 'exit()'. As does copy_page_tables(), this handles only 4Mb
blocks.
 */
int free_page_tables(unsigned long from,unsigned long size)
{
      unsigned long *pg_table;
      unsigned long * dir, nr;

      if (from & 0x3fffff)
            panic("free_page_tables called with wrong alignment");
      if (!from)
            panic("Trying to free up swapper memory space");
      size = (size + 0x3fffff) >> 22;
      dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */
      for ( ; size-->0 ; dir++) {
            if (!(1 & *dir))
                  continue;
            pg_table = (unsigned long *) (0xfffff000 & *dir);
            for (nr=0 ; nr<1024 ; nr++) {
                  if (1 & *pg_table)
                        free_page(0xfffff000 & *pg_table);
                  *pg_table = 0;
                  pg_table++;
            }
            free_page(0xfffff000 & *dir);
            *dir = 0;
      }
      invalidate();
      return 0;
}
/*
 *  Well, here is one of the most complicated functions in mm. It
 * copies a range of linerar addresses by copying only the pages.
 * Let's hope this is bug-free, 'cause this one I don't want to debug
:-)
 *
 * Note! We don't copy just any chunks of memory - addresses have to
 * be divisible by 4Mb (one page-directory entry), as this makes the
 * function easier. It's used only by fork anyway.
```

```
 *
 * NOTE 2!! When from==0 we are copying kernel space for the first
 * fork(). Then we DONT want to copy a full page-directory entry, as
 * that would lead to some serious memory waste - we just copy the
 * first 160 pages - 640kB. Even that is more than we need, but it
 * doesn't take any more memory - we don't copy-on-write in the low
 * 1 Mb-range, so the pages can be shared with the kernel. Thus the
 * special case for nr=xxxx.
 */
int copy_page_tables(unsigned long from,unsigned long to,long size)
{
      unsigned long * from_page_table;
      unsigned long * to_page_table;
      unsigned long this_page;
      unsigned long * from_dir, * to_dir;
      unsigned long nr;

      if ((from&0x3fffff) || (to&0x3fffff))
            panic("copy_page_tables called with wrong alignment");
      from_dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0
*/
      to_dir = (unsigned long *) ((to>>20) & 0xffc);
      size = ((unsigned) (size+0x3fffff)) >> 22;
      for( ; size-->0 ; from_dir++,to_dir++) {
            if (1 & *to_dir)
                  panic("copy_page_tables: already exist");
            if (!(1 & *from_dir))
                  continue;
            from_page_table = (unsigned long *) (0xfffff000 &
*from_dir);
            if (!(to_page_table = (unsigned long *) get_free_page()))
                  return -1; /* Out of memory, see freeing */
            *to_dir = ((unsigned long) to_page_table) | 7;
            nr = (from==0)?0xA0:1024;
            for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
                  this_page = *from_page_table;
                  if (!(1 & this_page))
                        continue;
                  this_page &= ~2;
                  *to_page_table = this_page;
                  if (this_page > LOW_MEM) {
                        *from_page_table = this_page;
                        this_page -= LOW_MEM;
                        this_page >>= 12;
                        mem_map[this_page]++;
                  }
            }
      }
      invalidate();
      return 0;
}

/*
 * This function puts a page in memory at the wanted address.
 * It returns the physical address of the page gotten, 0 if
 * out of memory (either when trying to access page-table or
 * page.)
 */
```

```
unsigned long put_page(unsigned long page,unsigned long address)
{
	unsigned long tmp, *page_table;

/* NOTE !!! This uses the fact that _pg_dir=0 */

	if (page < LOW_MEM || page >= HIGH_MEMORY)
		printk("Trying to put page %p at %p\n",page,address);
	if (mem_map[(page-LOW_MEM)>>12] != 1)
		printk("mem_map disagrees with %p at %p\n",page,address);
	page_table = (unsigned long *) ((address>>20) & 0xffc);
	if ((*page_table)&1)
		page_table = (unsigned long *) (0xfffff000 & *page_table);
	else {
		if (!(tmp=get_free_page()))
			return 0;
		*page_table = tmp|7;
		page_table = (unsigned long *) tmp;
	}
	page_table[(address>>12) & 0x3ff] = page | 7;
/* no need for invalidate */
	return page;
}

void un_wp_page(unsigned long * table_entry)
{
	unsigned long old_page,new_page;

	old_page = 0xfffff000 & *table_entry;
	if (old_page >= LOW_MEM && mem_map[MAP_NR(old_page)]==1) {
		*table_entry |= 2;
		invalidate();
		return;
	}
	if (!(new_page=get_free_page()))
		oom();
	if (old_page >= LOW_MEM)
		mem_map[MAP_NR(old_page)]--;
	*table_entry = new_page | 7;
	invalidate();
	copy_page(old_page,new_page);
}

/*
 * This routine handles present pages, when users try to write
 * to a shared page. It is done by copying the page to a new address
 * and decrementing the shared-page counter for the old page.
 *
 * If it's in code space we exit with a segment error.
 */
void do_wp_page(unsigned long error_code,unsigned long address)
{
#if 0
/* we cannot do this yet: the estdio library writes to code space */
/* stupid, stupid. I really want the libc.a from GNU */
	if (CODE_SPACE(address))
		do_exit(SIGSEGV);
#endif
```

```
	un_wp_page((unsigned long *)
		(((address>>10) & 0xffc) + (0xfffff000 &
		*((unsigned long *) ((address>>20) &0xffc)))));

}

void write_verify(unsigned long address)
{
	unsigned long page;

	if (!( (page = *((unsigned long *) ((address>>20) & 0xffc)) )&1))
		return;
	page &= 0xfffff000;
	page += ((address>>10) & 0xffc);
	if ((3 & *(unsigned long *) page) == 1)  /* non-writeable, present
*/
		un_wp_page((unsigned long *) page);
	return;
}

void get_empty_page(unsigned long address)
{
	unsigned long tmp;

	if (!(tmp=get_free_page()) || !put_page(tmp,address)) {
		free_page(tmp);		/* 0 is ok - ignored */
		oom();
	}
}

/*
 * try_to_share() checks the page at address "address" in the task "p",
 * to see if it exists, and if it is clean. If so, share it with the
current
 * task.
 *
 * NOTE! This assumes we have checked that p != current, and that they
 * share the same executable.
 */
static int try_to_share(unsigned long address, struct task_struct * p)
{
	unsigned long from;
	unsigned long to;
	unsigned long from_page;
	unsigned long to_page;
	unsigned long phys_addr;

	from_page = to_page = ((address>>20) & 0xffc);
	from_page += ((p->start_code>>20) & 0xffc);
	to_page += ((current->start_code>>20) & 0xffc);
/* is there a page-directory at from? */
	from = *(unsigned long *) from_page;
	if (!(from & 1))
		return 0;
	from &= 0xfffff000;
	from_page = from + ((address>>10) & 0xffc);
	phys_addr = *(unsigned long *) from_page;
/* is the page clean and present? */
```

```
	if ((phys_addr & 0x41) != 0x01)
		return 0;
	phys_addr &= 0xfffff000;
	if (phys_addr >= HIGH_MEMORY || phys_addr < LOW_MEM)
		return 0;
	to = *(unsigned long *) to_page;
	if (!(to & 1))
		if (to = get_free_page())
			*(unsigned long *) to_page = to | 7;
		else
			oom();
	to &= 0xfffff000;
	to_page = to + ((address>>10) & 0xffc);
	if (1 & *(unsigned long *) to_page)
		panic("try_to_share: to_page already exists");
/* share them: write-protect */
	*(unsigned long *) from_page &= ~2;
	*(unsigned long *) to_page = *(unsigned long *) from_page;
	invalidate();
	phys_addr -= LOW_MEM;
	phys_addr >>= 12;
	mem_map[phys_addr]++;
	return 1;
}

/*share_page() tries to find a process that could share a page with
 * the current one. Address is the address of the wanted page relative
 * to the current data space.
 *
 * We first check if it is at all feasible by checking executable->i_count.
 * It should be >1 if there are other tasks sharing this inode.
 */
static int share_page(unsigned long address)
{
	struct task_struct ** p;

	if (!current->executable)
		return 0;
	if (current->executable->i_count < 2)
		return 0;
	for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
		if (!*p)
			continue;
		if (current == *p)
			continue;
		if ((*p)->executable != current->executable)
			continue;
		if (try_to_share(address,*p))
			return 1;
	}
	return 0;
}
void do_no_page(unsigned long error_code,unsigned long address)
{
	int nr[4];
	unsigned long tmp;
	unsigned long page;
```

```
        int block,i;
        address &= 0xfffff000;
        tmp = address - current->start_code;
        if (!current->executable || tmp >= current->end_data) {
                get_empty_page(address);
                return;
        }
        if (share_page(tmp))
                return;
        if (!(page = get_free_page()))
                oom();
/* remember that 1 block is used for header */
        block = 1 + tmp/BLOCK_SIZE;
        for (i=0 ; i<4 ; block++,i++)
                nr[i] = bmap(current->executable,block);
        bread_page(page,current->executable->i_dev,nr);
        i = tmp + 4096 - current->end_data;
        tmp = page + 4096;
        while (i-- > 0) {
                tmp--;
                *(char *)tmp = 0;
        }
        if (put_page(page,address))
                return;
        free_page(page);
        oom();
}
void mem_init(long start_mem, long end_mem)
{
        int i;

        HIGH_MEMORY = end_mem;
        for (i=0 ; i<PAGING_PAGES ; i++)
                mem_map[i] = USED;
        i = MAP_NR(start_mem);
        end_mem -= start_mem;
        end_mem >>= 12;
        while (end_mem-->0)
                mem_map[i++]=0;
}
void calc_mem(void)
{
        int i,j,k,free=0;
        long * pg_tbl;

        for(i=0 ; i<PAGING_PAGES ; i++)
                if (!mem_map[i]) free++;
        printk("%d pages free (of %d)\n\r",free,PAGING_PAGES);
        for(i=2 ; i<1024 ; i++) {
                if (1&pg_dir[i]) {
                        pg_tbl=(long *) (0xfffff000 & pg_dir[i]);
                        for(j=k=0 ; j<1024 ; j++)
                                if (pg_tbl[j]&1)
                                        k++;
                        printk("Pg-dir[%d] uses %d pages\n",i,k);
                }
        }
}
```

CHAPITRE 4

Implimentation

1- Introduction

Pour l'implémenté un système il faut avoir les moyennes de création et les moyennes de transfère sur un disque de démarrage.

2- Les moyennes de programmations

On a utilisé le langage assembleur GNU/Assembleur sur le système d'exploitation linux RedHat « as »

Pour l'éditeur de lien et la création du programme binaire sur le même système on a utilisé « ld ».

Pour le transfert sur le disque on a utilisé « dd » sous le même système d'exploitation.

Pour plus de souplesse dans l'implémentation et testes nous avons travaillé sur la machine virtuel « VMWAR » on utilise deux disques virtuels : un pour le système linux et l'autre pour installer notre système

Fonctionnement du VMWAR :

VMware crée un environnement clos dans lequel sont disponibles un, deux, quatre ou huit (vSphere) processeur(s), des périphériques et un BIOS virtuel.

Selon les concepteurs, le microprocesseur est émulé seulement lorsque c'est nécessaire. Par exemple, les instructions initiées dans la VM (machine virtuelle) en mode user ou en mode virtuel 8086 ne sont pas toujours émulées, elles sont passées directement à l'OS hôte. Par contre, pour les instructions initiées en mode noyau ou en mode réel, VMWare va utiliser la technique dite de translation de code. Tout cela permet à VMware d'être plus rapide que des solutions multiplateformes qui émulent tout.

VMDK (Virtual Machine Disk) est un format de fichier ouvert permettant de simuler un disque dur virtuel pour les machines virtuelles telles que VMware, Virtualbox ou pour les émulateurs. Il a été créé à la base par les produits VMware

Conclusion générale

Comprendre le système d'exploitation c'est comprendre tout le monde interne de l'informatique, et de maitriser le contrôle de la machine pour profiter le maximum de ses capacités.

Le projet réalisé nous a permis de connaitre le détail de la programmation en assembleur et la création du code binaire avec tous les problèmes techniques de réalisation et testes

L'utilisation de la machine virtuelle ouvre des horizons très vastes pour le développement des systèmes.

Comme perspectives : le développement du système ne s'arrête jamais vu le l'apparition continu des équipements et nouvelles applications.

Bibliographies

[1] Intel® 64 and IA-32 Architectures, Software Developer's Manual, Volume 3A: System Programming Guide, Part 1

[2] Programmation en assembleur Gnu sur des microprocesseurs de la gamme Intel (du 80386 au Pentium-Pro)

[3] Introduction à l'Assembleur GNU Sous GNU/Linux sur Intel 80386. Issam Abdallah 2014

[4] ABCs of IBM z/OS System Programming Volume 1, Karan Singh, Paul Rogers, August 2014

[5] www.linux.org

[6] www.vmwar.com