

Pépin OS - Réaliser son propre système

Par michelizza

Date de publication : 12 juin 2017

CONFIRMÉ

Programmer le noyau d'un système d'exploitation est un très bon moyen pour en comprendre le fonctionnement, et c'est dans cette optique que j'ai moi-même commencé le développement de **Pépin**.

Ce tutoriel tente de guider le programmeur dans ses premiers pas en décortiquant la base de la réalisation d'un noyau et en en présentant les composantes élémentaires

Commentez

I - Introduction.....	7
I-A - Avertissements.....	7
I-B - Pourquoi ce tutoriel ?.....	7
I-C - Prérequis.....	7
I-C-1 - Compétences.....	7
I-C-2 - Outils.....	7
I-D - Ressources sur le web.....	8
II - Réaliser un secteur de boot qui affiche un message.....	8
II-A - Qu'est-ce qu'un secteur de boot ?.....	8
II-B - Comment est chargé le secteur de boot au démarrage de l'ordinateur ?.....	8
II-C - Créer un secteur de boot qui affiche un message.....	8
II-D - Que fait exactement ce programme ?.....	9
II-E - Compiler et tester le programme.....	11
III - Réaliser un secteur de boot qui charge et exécute un noyau.....	12
III-A - Le programme du boot loader.....	12
III-A-1 - Que fait exactement ce programme ?.....	13
III-B - Organiser la mémoire.....	14
III-C - Un premier noyau.....	15
III-D - Compiler et tester le boot loader et le noyau.....	16
IV - Programmer un secteur de boot qui passe en mode protégé.....	17
IV-A - Pourquoi utiliser le mode protégé ?.....	17
IV-A-1 - Le mode protégé, c'est quoi ?.....	17
IV-A-2 - Comment passer du mode réel au mode protégé.....	18
IV-B - Adresser la mémoire en mode protégé.....	18
IV-B-1 - Différents types d'adresses.....	18
IV-B-2 - Le mécanisme de segmentation.....	18
IV-B-3 - Les descripteurs de segment en détail.....	20
IV-B-3-a - Descripteur d'un segment de code.....	21
IV-B-3-b - Descripteur d'un segment de données.....	21
IV-C - Un boot loader qui passe en mode protégé.....	22
IV-C-1 - Quand passer en mode protégé ?.....	22
IV-C-2 - Le programme du boot loader.....	22
IV-C-2-a - Que fait exactement ce programme ?.....	23
IV-C-2-b - Passer en mode protégé.....	24
IV-D - Un noyau très simple.....	26
IV-D-1 - Le code du noyau.....	26
IV-D-2 - Afficher quelque chose à l'écran.....	26
IV-E - Compiler et tester le boot loader et le noyau.....	27
V - Écrire un noyau en C.....	28
V-A - Pourquoi passer de l'assembleur au C ?.....	28
V-B - Un noyau en assembleur qui fait appel à des fonctions en C.....	28
V-B-1 - Des routines pour afficher quelque chose à l'écran en C.....	28
V-B-2 - Un nouveau noyau en assembleur.....	30
V-B-3 - Compiler le noyau.....	30
V-C - Un noyau écrit entièrement en C.....	31
V-C-1 - Le code.....	31
V-C-2 - Compiler le noyau.....	31
VI - Programmer un noyau en C qui recharge la GDT.....	32
VI-A - Pourquoi changer de GDT ?.....	32
VI-B - Un noyau qui recharge la GDT.....	32
VI-B-1 - Le programme principal.....	32
VI-B-2 - Initialiser la GDT.....	33
VI-B-2-a - Les descripteurs de segment en C.....	33
VI-B-2-b - La fonction <code>init_gdt()</code>	33
VI-B-3 - Une fonction <code>main()</code> pour déjouer les pièges de gcc.....	35
VI-B-4 - Compiler le boot loader et le noyau.....	36
VII - Programmer les interruptions du processeur i386 avec le contrôleur d'interruptions 8259A.....	36
VII-A - Des interruptions pour prévenir le processeur d'un événement.....	36

VII-A-1 - Un exemple d'interruption.....	37
VII-B - Un chipset pour gérer les interruptions matérielles : le 8259A.....	37
VII-B-1 - À quoi sert le 8259A ?.....	37
VII-B-2 - Comment le 8259A traite les interruptions.....	37
VII-B-3 - Comment programmer le 8259A ?.....	37
VII-B-3-a - Registres ICW.....	38
VII-B-3-b - Registres OCW.....	39
VII-C - Exécuter la bonne routine de service grâce à la table des descripteurs d'interruptions.....	40
VII-C-1 - Descripteur système de type Interrupt Gate.....	41
VIII - Gérer les interruptions - la mise en œuvre.....	41
VIII-A - Préalable : étendre le code C du noyau à l'aide de directives en assembleur.....	41
VIII-B - Initialiser la table IDT.....	42
VIII-B-1 - Créer les descripteurs système de type Interrupt Gate.....	42
VIII-B-2 - Créer une routine d'interruption (ISR).....	42
VIII-B-3 - Initialiser l'IDT avec la fonction init_idt().....	44
VIII-C - Le programme principal du noyau.....	44
VIII-C-1 - Compiler et exécuter le noyau.....	45
IX - Gérer les interruptions du clavier.....	46
IX-A - Le chipset 8042.....	46
IX-B - La gestion du curseur.....	47
X - Créer une tâche : un noyau qui exécute une tâche utilisateur.....	48
X-A - Mode noyau et mode utilisateur.....	48
X-A-1 - Pourquoi ?.....	48
X-A-2 - Comment ?.....	49
X-B - Définir des segments de mémoire distincts.....	49
X-C - Copier le code exécutable en RAM.....	50
X-D - Créer et initialiser un TSS.....	50
X-D-1 - Descripteur de TSS.....	51
X-D-2 - Initialiser le TSS.....	51
X-E - Exécuter une tâche.....	52
X-E-1 - Simuler un retour d'interruption pour changer de tâche.....	52
X-E-1-a - Le code vu pas à pas.....	53
X-F - Sauvegarder le contexte d'une tâche.....	55
X-F-1 - Des interruptions qui sauvegardent la tâche en cours.....	55
X-G - Compiler et exécuter le noyau.....	56
XI - Les appels système : un noyau monotâche qui implémente des appels système.....	56
XI-A - Des appels système pour accéder aux services du noyau.....	57
XI-A-1 - Pourquoi ?.....	57
XI-A-2 - Appeler une routine privilégiée à l'aide d'une interruption logicielle.....	57
XI-A-3 - Une interruption logicielle qui utilise un Trap Gate.....	57
XI-A-3-a - Descripteur système de type Trap Gate.....	57
XI-A-4 - Comment passer des paramètres à l'appel système ?.....	58
XI-A-5 - La routine d'interruption (1).....	58
XI-A-6 - La routine d'interruption (2).....	58
XI-A-6-a - Que fait exactement cette fonction ?.....	59
XI-B - Charger une tâche.....	60
XI-C - Exécuter le noyau.....	60
XII - Gérer la mémoire - un noyau qui implémente la pagination.....	61
XII-A - Mémoire virtuelle et mémoire physique.....	61
XII-A-1 - Présentation.....	61
XII-A-2 - Le fonctionnement.....	62
XII-A-3 - Les entrées des répertoires et des tables de pages.....	63
XII-A-3-a - Un peu d'arithmétique.....	63
XII-A-4 - Notes sur le cache.....	63
XII-A-5 - Activer la pagination.....	63
XII-B - Une première implémentation très simple.....	64
XII-B-1 - Une organisation simple de la mémoire avec l'identity mapping.....	64
XII-B-2 - Initialiser le répertoire et les tables de pages.....	65

XII-B-2-a - Que fait exactement cette fonction ?.....	66
XII-B-3 - Une nouvelle exception pour gérer les Page Fault.....	66
XII-C - Exécuter le noyau.....	67
XIII - Gérer la mémoire - utiliser la pagination pour une tâche utilisateur.....	67
XIII-A - Préalable : gérer la mémoire physique.....	67
XIII-A-1 - Initialisation du bitmap.....	68
XIII-B - Comment organiser l'espace d'adressage d'une tâche utilisateur.....	71
XIII-C - Comment organiser l'espace noyau, pour gérer les appels système.....	72
XIII-C-1 - Note concernant la sécurité.....	73
XIII-D - Créer une tâche.....	74
XIII-D-1 - Une tâche très simple.....	74
XIII-D-2 - Charger la tâche.....	74
XIII-D-3 - Créer et initialiser le répertoire et les tables de pages de la tâche utilisateur.....	74
XIII-E - Exécuter la tâche utilisateur.....	75
XIV - Un système multitâche simple.....	77
XIV-A - Exécuter plusieurs tâches simultanément grâce à l'ordonnanceur.....	77
XIV-B - Un ordonnanceur appelé à chaque interruption d'horloge.....	77
XIV-B-1 - Sauvegarder la tâche en cours.....	79
XIV-B-2 - Choisir la nouvelle tâche.....	81
XIV-B-3 - Commuter.....	81
XIV-B-3-a - La fonction do_switch().....	82
XIV-B-4 - Note concernant les appels système.....	83
XIV-C - Charger une tâche en mémoire.....	83
XIV-D - Le contexte d'une tâche.....	84
XIV-E - Exécuter plusieurs tâches.....	84
XV - Un noyau multitâche avec des appels système préemptibles.....	85
XV-A - Un scheduler qui tient compte du contexte d'exécution.....	85
XV-A-1 - Une pile noyau par tâche : pourquoi ?.....	85
XV-A-2 - Une pile noyau par tâche : comment.....	86
XV-A-3 - L'ordonnanceur.....	87
XV-A-3-a - La fonction switch_to_task().....	92
XV-B - Utiliser les TrapGate pour rendre les appels système interruptibles.....	93
XVI - Un noyau qui boote avec Grub.....	94
XVI-A - Le standard multiboot.....	94
XVI-B - Un premier noyau minimaliste démarré par Grub.....	94
XVI-B-1 - L'en-tête.....	94
XVI-B-1-a - Le programme principal.....	95
XVI-C - Compiler et linker le noyau.....	95
XVI-D - Créer et mettre à jour une image de disquette avec Grub.....	96
XVI-E - Booter Pépin avec Grub.....	96
XVII - Gérer la mémoire physique et la mémoire virtuelle.....	97
XVII-A - Une nouvelle organisation de la mémoire physique et de la mémoire virtuelle.....	97
XVII-A-1 - Comment gérer « les » mémoires.....	97
XVII-A-2 - Utilisation de la mémoire physique.....	98
XVII-A-3 - Utilisation de la mémoire virtuelle.....	98
XVII-B - Allouer dynamiquement de la mémoire pour le noyau.....	99
XVII-B-1 - Gérer la mémoire physique.....	99
XVII-B-2 - Gérer la mémoire virtuelle.....	99
XVII-B-2-a - Le heap.....	100
XVII-B-2-b - Le heap de pages.....	100
XVII-B-2-c - Quand le noyau n'a plus assez de mémoire.....	101
XVII-C - Mettre à jour le répertoire et les tables de pages.....	101
XVII-C-1 - Rappel sur le mécanisme de pagination.....	102
XVII-C-2 - Modifier les tables de pages.....	102
XVII-C-3 - Modifier le répertoire.....	103
XVII-D - Compléments sur les répertoires de pages.....	104
XVII-E - Compléments sur la pagination.....	104
XVII-E-1 - Utiliser des pages de 4 Mo.....	104

XVII-E-2 - Mettre à jour le TLB.....	105
XVII-F - Structure d'une tâche.....	106
XVIII - Lire et écrire sur un disque IDE.....	106
XVIII-A - Écrire et lire sur un disque IDE avec les PIO.....	106
XVIII-B - Créer et utiliser une image d'un disque dur sous Unix.....	107
XVIII-C - Tester.....	107
XIX - Utiliser un système de fichiers Ext2FS.....	108
XIX-A - Description d'un système de fichiers Ext2FS.....	108
XIX-A-1 - Un disque subdivisé en groupes.....	108
XIX-B - Lecture dans un système de fichiers Ext2FS.....	108
XIX-B-1 - Le superbloc.....	109
XIX-B-2 - Le bloc de descripteurs de groupes.....	114
XIX-B-3 - Localiser sur le disque une inode.....	114
XIX-B-4 - Lire un fichier.....	114
XIX-C - Créer et utiliser une image d'un disque dur sous Unix avec un système de fichier Ext2FS.....	114
XX - Créer et lancer une application au format ELF à partir du système de fichiers.....	115
XX-A - Remarques.....	115
XX-B - Anatomie d'un fichier ELF.....	115
XX-C - Charger un exécutable.....	117
XX-D - Un nouvel appel système : exit().....	117
XX-E - Créer et lancer une application au format ELF à partir du système de fichiers.....	118
XXI - Booter avec Grub sur un disque IDE.....	118
XXI-A - Créer une image d'un disque IDE partitionné.....	118
XXI-A-1 - Installer grub sur une image bootable.....	119
XXI-B - Utiliser un disque logique.....	120
XXII - Quelques structures élémentaires pour gérer les fichiers.....	121
XXII-A - Organiser les fichiers.....	121
XXII-A-1 - Le disque logique.....	121
XXII-A-2 - Structure d'un fichier et arborescence.....	121
XXII-B - De nouveaux appels système.....	122
XXII-B-1 - Structure d'un processus : gérer les fichiers.....	122
XXIII - Une méthode générique pour gérer les listes chaînées.....	123
XXIII-A - Des listes de données.....	124
XXIII-B - Les listes chaînées sous FreeBSD.....	124
XXIII-B-1 - Le type SLIST.....	125
XXIII-B-2 - Le type STAILQ.....	126
XXIII-B-3 - Le type LIST.....	126
XXIII-B-4 - Le type TAILQ.....	126
XXIII-C - Les listes chaînées sous Linux.....	127
XXIII-D - Les listes chaînées sous Pépin.....	128
XXIV - Un premier shell.....	129
XXIV-A - Un processus qui utilise la console.....	130
XXIV-A-1 - Attacher une console à un processus pour gérer les entrées/sorties d'un terminal.....	130
XXIV-A-2 - Utiliser l'appel système sys_console_read() pour entrer des données au clavier.....	130
XXIV-B - Créer un nouveau processus.....	132
XXIV-B-1 - Passer des arguments à un programme.....	132
XXIV-C - Allouer dynamiquement de la mémoire à un processus avec malloc().....	132
XXIV-D - Le noyau et un premier shell.....	133
XXV - Implémenter les signaux.....	133
XXV-A - Un simple problème d'affichage, conséquences en termes d'architecture.....	134
XXV-B - Implémenter la filiation entre processus.....	136
XXV-B-1 - De nouveaux champs dans la struct process.....	136
XXV-B-2 - À la création du processus.....	136
XXV-B-3 - À la mort du processus.....	136
XXV-C - Attendre la fin d'un processus avec l'appel système sys_wait().....	137
XXV-D - Implémenter les signaux.....	137
XXV-D-1 - Traiter les signaux.....	138
XXV-D-1-a - Modifier l'algorithme de l'ordonnanceur.....	138

XXV-D-2 - Utiliser une fonction de traitement de signal personnalisée.....	138
XXV-D-2-a - Associer une fonction personnalisée avec l'appel sys_sigaction().....	138
XXV-D-2-b - Exécuter une fonction personnalisée - le problème posé.....	138
XXV-D-2-c - Sauvegarder le contexte sur la pile utilisateur.....	139
XXV-D-2-d - Revenir d'une routine avec la technique du stack-smashing.....	139
XXV-D-2-e - Revenir avec sys_sigreturn().....	139
XXVI - Annexe A : Compilation séparée en assembleur sous Unix.....	139
XXVI-A - Le programme principal.....	139
XXVI-B - La fonction.....	140
XXVI-B-1 - Compilation et édition de liens.....	140
XXVI-C - Création d'exécutables sous Unix.....	141
XXVI-C-1 - Un programme principal en assembleur.....	141
XXVI-C-2 - Un programme principal en C et une fonction en assembleur.....	141
XXVI-C-2-a - Le programme principal.....	141
XXVI-C-2-b - Compilation et linkage.....	141
XXVII - Annexe B - arithmétique en base 16.....	142
XXVII-A - Conversion entre bases.....	142
XXVIII - Annexe C - Bochs en mode debug.....	142
XXVIII-A - Commandes utiles.....	142
XXVIII-A-1 - Général.....	142
XXVIII-A-1-a - Segments, interruptions, gestion de tâches.....	143
XXVIII-A-1-b - Pagination.....	143
XXIX - Annexe D - Gérer les arguments sur la pile avec les Stack Frame.....	143
XXX - Annexe E - Déboguer le noyau avec gdb.....	145
XXX-A - Compiler avec l'option -g.....	145
XXX-B - Lancer l'émulateur.....	145
XXX-C - utiliser gdb.....	146
XXX-D - Annexe F - Booter avec Grub2.....	146
XXXI - GNU Free Documentation License.....	147
XXXII - FAQ : questions souvent posées.....	152
XXXII-A - Réponses.....	152
XXXII-A-1 - Le code ne compile pas.....	152
XXXII-A-2 - Le noyau ne fonctionne pas avec mon émulateur.....	153
XXXII-A-3 - La compilation produit l'erreur undefined reference to __stack_chk_fail.....	153
XXXII-A-4 - Pour compiler le noyau sur une plateforme 64 bits.....	153
XXXIII - Conclusion.....	153
XXXIV - Note de la rédaction de developpez.com.....	153

I - Introduction

I-A - Avertissements

J'ai essayé de rendre les explications les plus simples possible. Néanmoins, si vous vous faites des migraines en raison de mes explications filandreuses, n'hésitez pas à **me contacter**, je pourrai alors essayer de clarifier les choses.

Si le code source vous nargue en refusant de fonctionner, essayez de relire les explications. Bien sûr, vous pouvez toujours me contacter si quelque chose vous semble erroné, mais dans tous les cas, essayez d'expliquer votre démarche et de joindre le fichier de logs de votre émulateur.

I-B - Pourquoi ce tutoriel ?

Programmer le noyau d'un système d'exploitation est un très bon moyen pour en comprendre le fonctionnement, et c'est dans cette optique que j'ai moi-même commencé le développement de **Pépin**.

Ce tutoriel tente de guider le programmeur dans ses premiers pas en décortiquant la base de la réalisation d'un noyau et en en présentant les composantes élémentaires. À chaque fois que les sources d'un nouveau noyau sont proposées en exemple dans un chapitre, j'ai essayé de minimiser le nombre de lignes de code supplémentaire par rapport à celles du chapitre précédent. Vous pourrez donc utiliser la commande `diff` pour bien voir les parties modifiées ou ajoutées d'un chapitre à l'autre.

I-C - Prérequis

I-C-1 - Compétences

Une partie du noyau est codée en **assembleur i386** et le reste est codé en **langage C**. La maîtrise de ce dernier ainsi que la connaissance des notions de base en assembleur sont donc un préalable à la compréhension de ce tutoriel.

Même si les notions importantes relatives à l'architecture i386 sont expliquées ici au fur et à mesure, la lecture des documents suivants est fortement recommandée.

Toutes les informations relatives à l'architecture Intel et à certains microcontrôleurs sont téléchargeables :

- **Manuels (c)Intel**
- **8259A - Programmable Interrupt Controller**

Les spécifications permettant d'écrire un noyau respectant le standard **Multiboot** :

- http://en.wikipedia.org/wiki/Multiboot_Specification

Concernant le format **ELF** :

- https://fr.wikipedia.org/wiki/Executable_and_Linking_Format

I-C-2 - Outils

- **nasm** est un excellent assembleur open source.
- **gcc** est un compilateur portable et performant.
- **Bochs** est un émulateur d'architecture i386 complet qui est très pratique pour tracer le fonctionnement des premiers kernels. Cela permet de les déboguer plus facilement.

I-D - Ressources sur le web

- **Osdev.org**, en anglais, mais plein d'explications claires et surtout un forum très actif et de haut niveau sur les systèmes d'exploitation. À voir absolument !
- **JamesM's kernel development tutorials**
- **OS Development Series**

Et aussi, pour ceux qui débutent en administration système, je recommande l'excellentissime **Guide du Rootard**.

II - Réaliser un secteur de boot qui affiche un message



La programmation d'un noyau est difficile et ce tutoriel s'adresse en premier lieu aux programmeurs ayant déjà une bonne expérience de la programmation et une connaissance suffisante de la théorie des systèmes d'exploitation. L'ensemble des prérequis nécessaires à la bonne compréhension de ce tutoriel est détaillé dans [l'introduction](#). Si vous débutez en C, que vous ne connaissez pas l'assembleur ou bien que vous n'avez qu'une très faible idée des fonctionnalités d'un noyau, il vous sera donc très difficile de tout saisir. Quoi qu'il en soit, je vous souhaite une excellente lecture et une bonne compilation !

II-A - Qu'est-ce qu'un secteur de boot ?

Un **secteur de boot** est un programme situé sur le premier secteur d'une unité de stockage, et qui est chargé et exécuté au démarrage du PC. Le programme de ce secteur a en principe pour tâche de charger un noyau en mémoire et de l'exécuter. Ce noyau peut être présent au départ sur une disquette, un disque dur, une bande ou tout autre support magnétique. Ce chapitre détaille ce qui se passe quand on boote sur disquette, mais les principes expliqués ici restent valables pour tout autre support.

II-B - Comment est chargé le secteur de boot au démarrage de l'ordinateur ?

Au démarrage, le PC commence par initialiser et tester le processeur, la mémoire et les périphériques. C'est le Power On Self Test (**POST**). Ensuite, le PC charge et exécute un programme particulier résidant en ROM, le **BIOS** (« Basic Input/Output System » ou « Built In Operating System »). Le BIOS peut être vu comme un système d'exploitation minimal chargé automatiquement par le PC et dont l'un des rôles est de charger un véritable système d'exploitation en essayant de trouver un secteur de boot valide parmi les unités de stockage. Une fois trouvé, ce secteur est chargé en mémoire à l'adresse 0000:7C00 puis le BIOS passe la main au programme de boot fraîchement chargé.

Le secteur de boot de l'unité de stockage est appelé **Master Boot Record (MBR)**. Il contient des données, dont la **table des partitions**, et du code exécutable. La table des partitions contient des informations sur les partitions primaires du disque (où elles commencent, leur taille, etc.). Le code exécutable d'un MBR standard cherche sur la table des partitions une partition active, puis, si une telle partition existe, ce code charge le secteur de boot de cette partition (appelé aussi **VBR**). C'est souvent ce deuxième secteur de boot qui charge le noyau et lui donne la main.

II-C - Créer un secteur de boot qui affiche un message

Dans notre cas, nous allons pour commencer créer un secteur de boot très simple qui ne fera qu'afficher un message de bienvenue. Cela n'est pas bien difficile et pour vous épargner un long suspense, voici le programme :

```
[BITS 16] ; indique a Nasm que l'on travaille en 16 bits
[ORG 0x0]

; initialisation des segments en 0x07C00
mov ax, 0x07C0
```



```

mov ds, ax
mov es, ax
mov ax, 0x8000
mov ss, ax
mov sp, 0xf000    ; pile de 0x8F000 -> 0x80000

; affiche un msg
mov si, msgDebut
call afficher

end:
  jmp end

;--- Variables ---
msgDebut db "Hello World !", 13, 10, 0
;-----

;-----
; Synopsis: Affiche une chaîne de caractères se terminant par 0x0
; Entrée:   DS:SI -> pointe sur la chaîne à afficher
;-----
afficher:
  push ax
  push bx
.debut:
  lodsb      ; ds:si -> al
  cmp al, 0   ; fin chaîne ?
  jz .fin
  mov ah, 0x0E ; appel au service 0x0e, int 0x10 du bios
  mov bx, 0x07 ; bx -> attribut, al -> caractère ASCII
  int 0x10
  jmp .debut

.fin:
  pop bx
  pop ax
  ret

;--- NOP jusqu'à 510 ---
times 510-($-$$) db 144
dw 0xAA55

```

II-D - Que fait exactement ce programme ?

[BITS 16] indique à Nasm que l'on travaille en 16 bits.

On indique tout d'abord que l'on travaille sur 16 bits pour le codage des instructions et des données, c'est le mode par défaut. Ceci n'est pas encore le début du programme, c'est juste une directive de compilation pour que l'exécutable obtenu soit bien sur 16 bits et pas sur 32 bits.

[ORG 0x0] Cette directive indique l'offset à ajouter à toutes les adresses référencées.

```

; initialisation des segments en 0x07C00
mov ax, 0x07C0
mov ds, ax
mov es, ax

```

Le programme du secteur de boot est chargé par le BIOS en 0x7C00, donc toutes les données internes au programme sont situées à partir de cette adresse. Le bloc ci-dessus initialise les registres ds et es qui servent à indiquer où débute le segment de données. En mettant la valeur 0x7C00 dans le sélecteur de données, on accède en fait à l'adresse 0x07C0:0000 = 0x07C00.



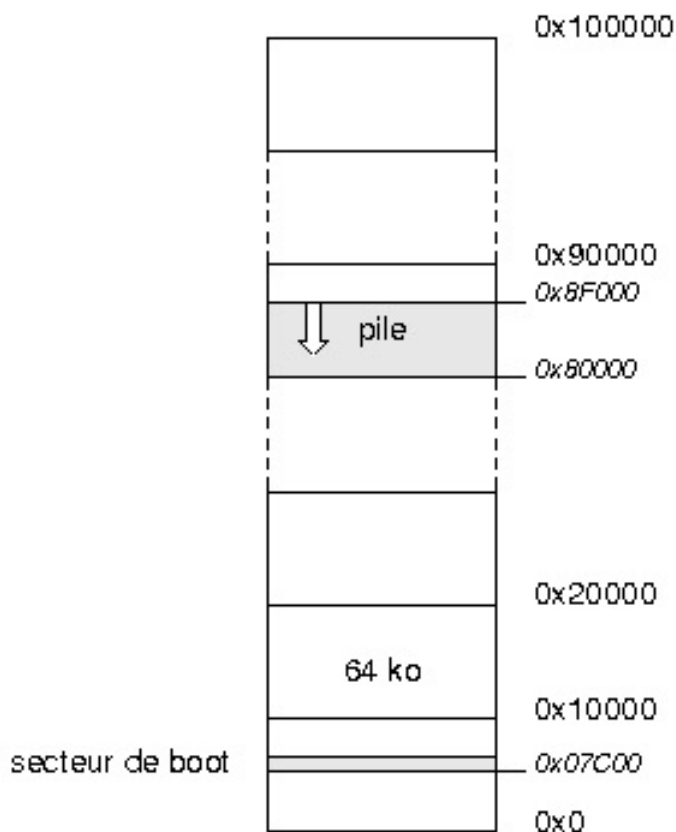
L'adressage en mode réel est particulier. La valeur du sélecteur représente les 16 bits « hauts » d'une adresse linéaire sur 20 bits. Il faut donc multiplier par 0x10 (par 16 en décimal) la valeur du sélecteur pour obtenir la base. Par exemple, l'adresse A000:1234 en mode réel correspond à l'adresse physique A0000 + 1234 = A1234. Cela signifie aussi qu'un segment en mode réel occupe 64 k, et pas un octet de plus !

```
; pile de 0x8F000 -> 0x80000
mov ax, 0x8000
mov ss, ax
mov sp, 0xf000
```

Ensuite on initialise le segment de pile SS et le pointeur de pile SP en faisant commencer la pile en 0x8F000 et en la faisant finir en 0x80000. Note : le choix de ces valeurs est arbitraire, dans le cas présent, la pile aurait pu être placée ailleurs.

À ce stade du programme :

- le secteur de boot est chargé en 0x07C00
- le segment de données est initialisé pour couvrir les adresses de 0x07C00 à 0x17C00
- la pile commence en 0x8F000 et finit en 0x80000.



```
; affiche un msg
mov si, msgDebut
call afficher
```

Une fois les principaux registres initialisés, la fonction afficher est appelée pour mettre à l'écran le message pointé par msgDebut. Cette fonction fait appel au BIOS pour gérer l'affichage du message. Plus précisément, elle appelle le service 0x0e de l'interruption logicielle 0x10 du BIOS qui permet d'afficher un caractère à l'écran en précisant ses attributs (couleur, luminosité...). Il aurait été possible d'écrire ce message à l'écran en se passant de cette facilité

offerte par le BIOS, mais cela aurait été beaucoup moins simple à réaliser (c'est d'ailleurs l'objet du chapitre suivant... patience !).

```
end:
    jmp end
```

Une fois le message affiché, le programme boucle et ne fait plus rien.

```
msgDebut db "Hello world !", 13, 10, 0
```

En fin de fichier, on définit les variables et les fonctions utilisées dans le programme. La chaîne de caractères affichée au démarrage a pour nom msgDebut.

Ensuite, la fonction afficher est définie. Elle prend en argument une chaîne de caractères pointée par les registres DS et SI. DS correspond à l'adresse du segment de données et SI est un déplacement par rapport au début de ce segment (un offset). La chaîne de caractères passée en argument doit se terminer par un octet égal à 0 (comme en C).

```
;--- NOP jusqu'à 510 ---
times 510-($-$$) db 144
dw 0xAA55
```

Cette directive ajoute du bourrage sous forme d'instructions NOP (opcode 0x90, ou 144), puis le mot 0xAA55 à la fin du code compilé afin que le binaire généré fasse 512 octets. Le mot 0xAA55 est crucial : en fin de secteur, il est une signature pour indiquer au BIOS que le secteur en question est un MBR !

II-E - Compiler et tester le programme

Le programme est écrit dans le fichier bootsect.asm. Pour le compiler avec Nasm et obtenir le binaire (exécutable au boot) bootsect, il faut utiliser la commande suivante :

```
nasm -f bin -o bootsect bootsect.asm
```

Ensuite, pour tester notre secteur de boot, nous pourrions l'écrire sur une disquette et rebooter notre machine préférée avec la disquette dedans. Mais cette démarche est très fastidieuse, car un PC met toujours du temps à rebooter et les possibilités de débogage sont très limitées à ce stade. Heureusement, il existe de très bons émulateurs de PC, tels que **Bochs** ou **qemu**. Pour ma part, j'utilise **Bochs** qui, contrairement à **qemu**, possède un mode **debug** puissant.

Bochs a la possibilité d'utiliser un fichier en faisant comme si c'était une vraie disquette. Sous Unix, pour générer une image de disquette :

```
cat bootsect /dev/zero | dd of=floppyA bs=512 count=2880
```



*Je ne sais pas trop comment tout cela peut être fait sous Windows (je ne travaille que sous **Debian GNU/Linux**). En revanche, vous trouverez d'autres développeurs Windows sur l'excellent forum **Osdev.org**.*

Ensuite, la commande pour démarrer sur cette disquette virtuelle avec Bochs devrait ressembler à ça :

```
bochs 'boot:a' 'floppya: 1_44=floppyA, status=inserted'
```



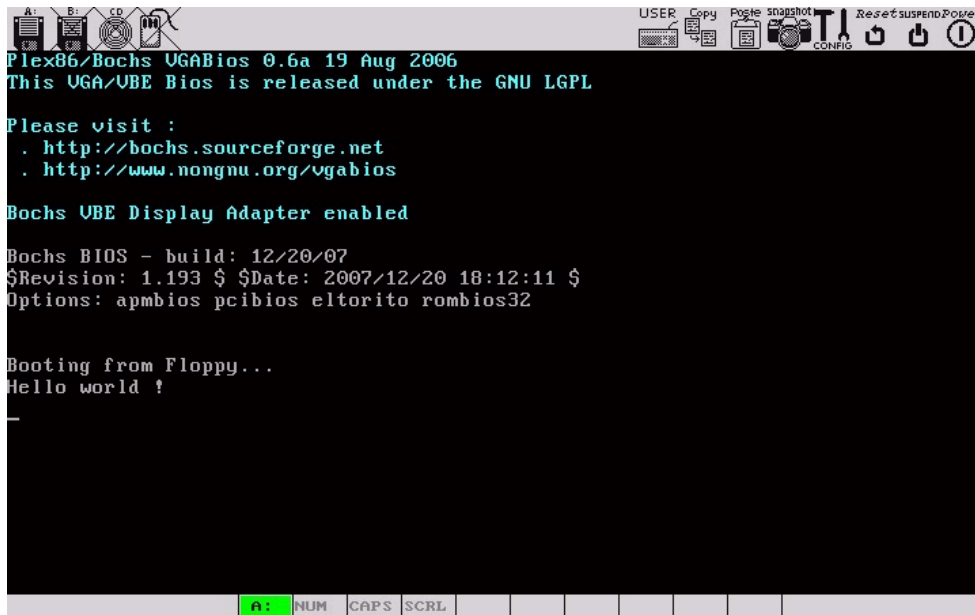
Il est assez peu probable que Bochs fonctionne du premier coup si vous ne vous êtes pas donné la peine de lire sa documentation pour créer un fichier de configuration qui lui

soit intelligible. Je sais, ça n'est pas la partie la plus passionnante, mais elle est hélas indispensable !

Pour ceux qui préfèrent utiliser **qemu**, il faut taper la commande suivante :

```
qemu -boot a -fda floppyA
```

Le résultat obtenu avec **Bochs** :



III - Réaliser un secteur de boot qui charge et exécute un noyau

- **Le programme du boot loader**
- **Organiser la mémoire**
- **Un premier noyau**
- **Compiler et tester le boot loader et le noyau**

Sources

Le package contenant les sources est téléchargeable ici : **bootsect.tgz**.

III-A - Le programme du boot loader

La partie précédente explique par l'exemple les principes de fonctionnement d'un programme de boot. Dans cette partie, nous allons voir un programme de boot plus raffiné qui, après avoir affiché un message, charge en mémoire un noyau très rudimentaire et lui passe la main. Là encore, nous allons faire au plus simple et le noyau se contentera seulement d'afficher un message.

Ce programme ressemble beaucoup à celui du chapitre précédent. En fait, c'est le même avec juste quelques lignes de code en plus qui copient une partie de la disquette, contenant le noyau, en mémoire :

```
?fine BASE    0x100    ; 0x0100:0x0 = 0x1000
?fine KSIZE   1      ; nombre de secteurs de 512 octets à charger

[BITS 16]
```

```
[ORG 0x0]

jmp start
#include "UTIL.INC"
start:

; initialisation des segments en 0x07C0
mov ax, 0x07C0
mov ds, ax
mov es, ax
mov ax, 0x8000 ; stack en 0xFFFF
mov ss, ax
mov sp, 0xf000

; récupération de l'unité de boot
mov [bootdrv], dl

; affiche un msg
mov si, msgDebut
call afficher

; charger le noyau
xor ax, ax
int 0x13

push es
mov ax, BASE
mov es, ax
mov bx, 0

mov ah, 2
mov al, KSIZE
mov ch, 0
mov cl, 2
mov dh, 0
mov dl, [bootdrv]
int 0x13
pop es

; saut vers le kernel
jmp dword BASE:0

msgDebut: db "Chargement du kernel", 13, 10, 0

bootdrv: db 0

;; NOP jusqu'à 510
times 510-($-$$) db 144
dw 0xAA55
```

III-A-1 - Que fait exactement ce programme ?

```
jmp start
#include "UTIL.INC"
start:
```

Le programme commence par un saut à l'adresse `start`. La directive `include` ajoute au code du noyau le contenu du fichier `UTIL.INC` qui contient le code de la fonction `afficher` vue précédemment.

```
; récupération de l'unité de boot
mov [bootdrv], dl
```

Cette instruction met dans une variable un nombre servant à identifier le périphérique de boot (ici le lecteur de disquettes). Cette variable sera réutilisée plus tard pour indiquer à partir de quel périphérique doit être chargé le noyau.

```
; charger le noyau
xor ax, ax
int 0x13

push es
mov ax, BASE
mov es, ax
mov bx, 0

mov ah, 2
mov al, KSIZE
mov ch, 0
mov cl, 2
mov dh, 0
mov dl, [bootdrv]
int 0x13
pop es
```

Le bout de code ci-dessus charge le noyau en mémoire en faisant appel à l'interruption **0x13** du BIOS qui permet de copier un ou plusieurs secteurs d'une disquette en mémoire. Dans notre cas, le noyau se situe au début du second secteur de la disquette et le programme recopie ce secteur à l'adresse 0x1000 en RAM. Notez que le choix de cette adresse est arbitraire et on aurait très bien pu choisir une autre valeur.



L'adresse physique 0x1000 est adressée ici en mettant le sélecteur à 0x0100. Mais on aurait très bien pu procéder autrement, en utilisant un sélecteur à 0 et un offset de 0x1000, ça aurait aussi fonctionné !

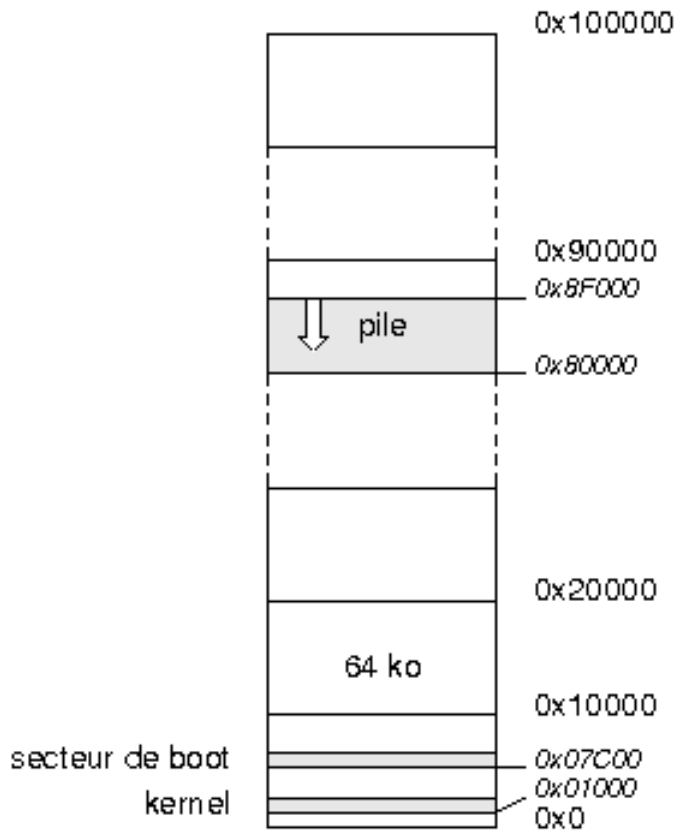
La variable KSIZE définit le nombre de secteurs à charger pour que tout le noyau soit bien recopié en mémoire. Ce premier noyau, qui est décrit dans la partie suivante, est très court (pas plus de 100 octets). On peut donc se contenter de mettre la valeur de KSIZE à 1 pour copier un seul secteur.

```
; saut vers le noyau
jmp dword BASE:0
```

Ensuite, une instruction de saut donne la main au code du noyau.

III-B - Organiser la mémoire

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 feront parfois économiser des heures de débogage ! Dans notre cas, la mémoire est occupée de la façon suivante après le chargement du noyau :



III-C - Un premier noyau

Voici le programme du noyau :

```
[BITS 16]
[ORG 0x0]

jmp start

#include "UTIL.INC"

start:
; initialisation des segments en 0x100
mov ax, 0x100
mov ds, ax
mov es, ax

; initialisation du segment de pile
mov ax, 0x8000
mov ss, ax
mov sp, 0xf000

; affiche un msg
mov si, msg00
call afficher

end:
jmp end

msg00: db 'Kernel is speaking !', 10, 0
```

Ce programme initialise le registre de code et les registres de données afin qu'ils pointent sur la bonne zone mémoire, en 0x1000. Ensuite, un message est affiché pour attester de la réussite des opérations. À la ligne suivante, le noyau ne fait vraiment pas grand-chose : il boucle indéfiniment.

III-D - Compiler et tester le boot loader et le noyau

Le package contenant les sources est téléchargeable ici : [bootsect.tgz](#).

Le code se décompose en :

- un fichier bibliothèque, UTIL.INC, qui contient la fonction `afficher` ;
- un fichier qui contient le code du secteur de boot ;
- un fichier qui contient le code du noyau.

```
$ ls
UTIL.INC bootsect.asm kernel.asm
```

On compile le boot loader et le noyau séparément :

```
nasm -f bin -o bootsect bootsect.asm
nasm -f bin -o kernel kernel.asm
```

On remarque que le binaire du secteur de boot fait comme prévu 512 octets :

```
$ ls -l
total 14
-rw-r--r--  1 am      users    492 Jan 17 17:20 UTIL.INC
-rw-r--r--  1 am      users    512 Jan 19 18:16 bootsect
-rw-r--r--  1 am      users    715 Jan 17 17:22 bootsect.asm
-rw-r--r--  1 am      users    297 Jan 17 17:50 kernel.asm
-rw-r--r--  1 am      users     69 Jan 19 18:16 kernel
```

On remarque aussi que le noyau fait seulement 69 octets, soit moins d'un secteur de disquette (512 octets), ce qui permet d'utiliser une valeur très basse pour KSIZE. Au cas où le noyau occupe davantage de place, il faut augmenter cette valeur. Malheureusement, cela fonctionne dans une certaine limite, car la lecture de données sur disque est complexe. Pour en savoir plus :

- http://en.wikipedia.org/wiki/INT_13, décrit l'utilisation de l'interruption 0x13 du BIOS ;
- <https://fr.wikipedia.org/wiki/Cylinder/Head/Sector>, décrit la structure des disques et des disquettes pour PC.

La disquette que nous allons faire aura le noyau placé sur le deuxième secteur. On réalise une image de la disquette avec la commande suivante :

```
cat bootsect kernel /dev/zero | dd of=floppyA bs=512 count=2880

ls -l floppyA
-rw-r--r--  1 am      users  1474117 Jan 19 18:27 floppyA
```



IV - Programmer un secteur de boot qui passe en mode protégé

- **Pourquoi utiliser le mode protégé ?**
- **Adresser la mémoire en mode protégé**
- **Un boot loader qui passe en mode protégé**
- **Un noyau très simple**

Sources

Le package contenant les sources est téléchargeable ici : [bootsect_PMode.tgz](#).

IV-A - Pourquoi utiliser le mode protégé ?

IV-A-1 - Le mode protégé, c'est quoi ?

Le microprocesseur d'un PC possède trois modes de fonctionnement :

- le **mode réel**, qui est le mode par défaut, fournit les mêmes fonctionnalités que le 8086. Mais cela a certaines conséquences comme l'impossibilité d'adresser plus de 1 Mo de mémoire ;
- le **mode protégé** (voir aussi sur [Wikipédia](#)) 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 multitâches et multiutilisateurs ;
- le **mode virtuel** est un mode particulier très peu utilisé.

Notre objectif étant de réaliser un noyau multiutilisateur, multitâche et pouvant adresser toute la mémoire, il nous faudra basculer le microprocesseur en mode protégé. Mais cela a d'importantes conséquences pour le programmeur :

- le mécanisme d'adressage en mode protégé est très différent de celui en mode réel ;
- le jeu d'instruction n'est plus sur 16 bits, mais sur 32 bits et 64 bits sur les processeurs actuellement utilisés ;
- il n'est pas possible avec ce mode de s'appuyer sur les routines du BIOS pour accéder aux périphériques. Tous les drivers sont donc à réécrire !

IV-A-2 - Comment passer du mode réel au mode protégé

Passer du mode réel au mode protégé est très simple, il suffit de mettre le bit 0 du registre CR0 à 1 :

```
; PE mis a 1 (CR0)
mov eax, cr0
or ax, 1
mov cr0, eax
```

Mais si cela suffit pour changer de mode, cela ne suffit pas à ce qu'un programme continue de fonctionner une fois le mode protégé établi. Pourquoi ? L'adressage en mode protégé, qui diffère de l'adressage en mode réel, s'appuie sur des structures qui doivent être correctement initialisées lors du changement de mode pour que le processeur continue d'adresser correctement la mémoire (et notamment le segment de données, la pile et le pointeur d'instruction).

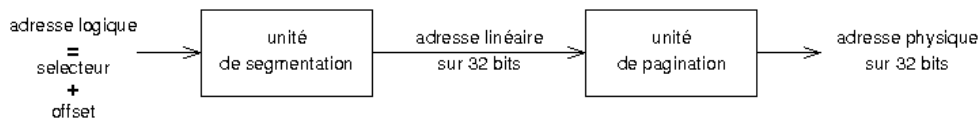
IV-B - Adresser la mémoire en mode protégé

IV-B-1 - Différents types d'adresses

En mode protégé, il existe pour le programmeur 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.

Le schéma ci-dessous résume le principe de l'adressage en mode protégé :



Dans un premier temps, nous allons utiliser uniquement le mécanisme de segmentation sans le mécanisme de pagination, plus délicat à mettre en œuvre.

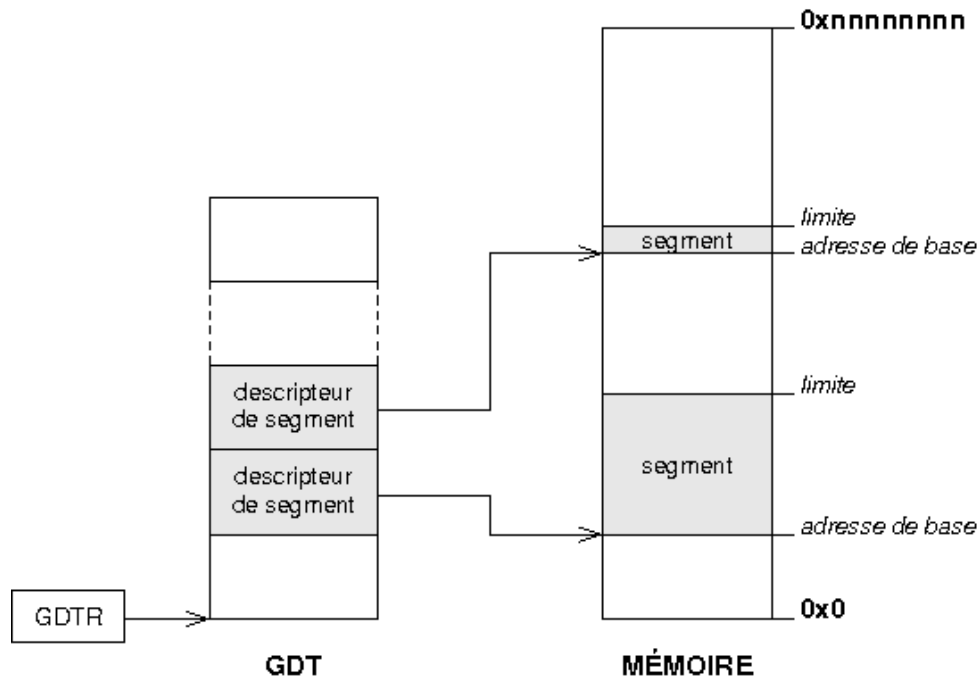
IV-B-2 - Le mécanisme de segmentation

Une adresse logique est constituée par un sélecteur de segment et un offset. Le sélecteur sélectionne un bloc mémoire d'une certaine taille, appelé **segment**, qui définit en quelque sorte l'espace de travail du programme. L'offset est un déplacement par rapport au début de ce bloc.

Un segment est décrit par une structure de 64 bits appelée **descripteur de segment** qui précise :

- sa **base**, l'endroit en mémoire où commence le segment ;
- sa **limite**, la taille du segment exprimée en octets ou en blocs de 4 ko ;
- son type (code, données, pile ou autre).

Les descripteurs sont stockés dans la **Global Descriptor Table (GDT)**. Cette table peut résider n'importe où en mémoire. Son adresse en mémoire physique est renseignée au processeur grâce à un registre particulier : le **GDTR** :



Le **sélecteur de segment** est un registre de 16 bits directement manipulé par le programmeur qui pointe sur un descripteur de segment dans la GDT et indique de ce fait dans quel segment on se situe. Ces registres sont bien connus de ceux qui ont déjà programmé en mode réel :

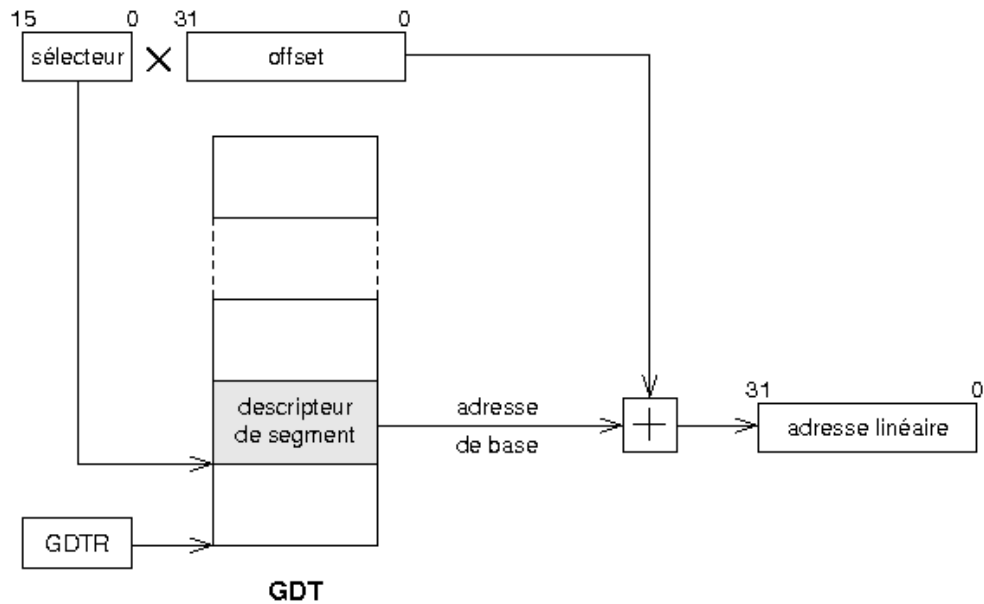
- CS est le sélecteur de segment de code ;
- DS est le sélecteur de segment de données ;
- ES, FS et GS sont des sélecteurs de segments généraux ;
- SS est le sélecteur de segment de pile.

Sur les processeurs 64 bits, la segmentation n'est plus utilisée sauf pour les segments FS et GS. En mode compatibilité, la segmentation est toujours présente :



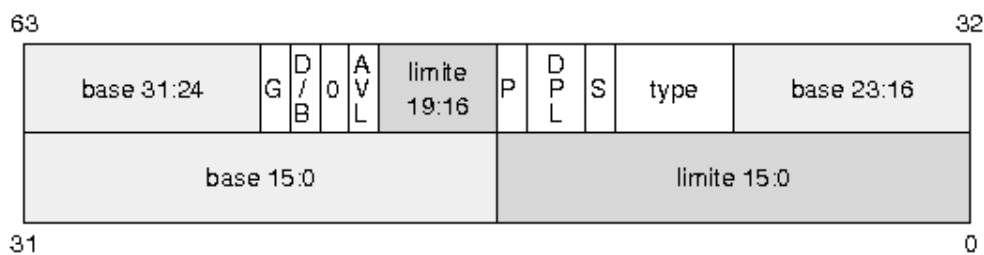
Extrait du manuel Intel.

Le sélecteur pointe sur un descripteur qui donne l'adresse où commence le segment. En ajoutant l'offset à cette base, on obtient une adresse linéaire sur 32 bits :



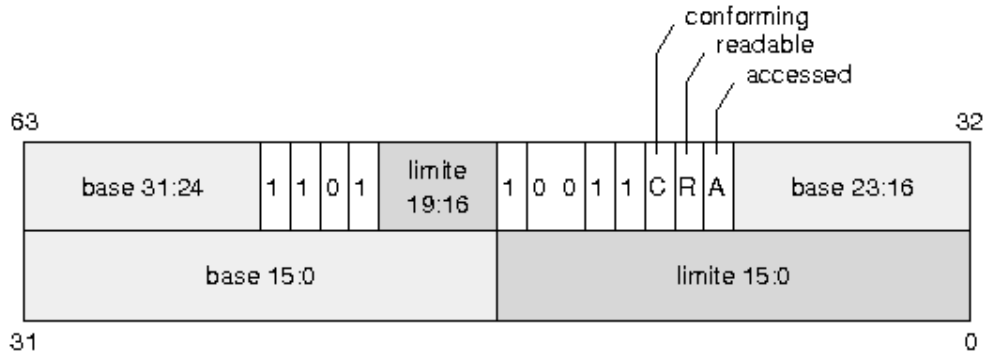
IV-B-3 - Les descripteurs de segment en détail

Le schéma ci-dessous décrit la structure générale d'un descripteur de segment :



- la **base**, sur 32 bits, est l'adresse linéaire où débute le segment en mémoire ;
- la **limite**, sur 20 bits, définit la longueur du segment ;
- si le **bit G** est à 0, la limite est exprimée en octets, sinon, elle est exprimée en nombre de pages de 4 ko ;
- le **bit D/B** précise la taille des instructions et des données manipulées. Il est mis à 1 pour 32 bits ;
- le **bit AVL** est librement disponible ;
- le **bit P** est utilisé pour déterminer si le segment est présent en mémoire physique. Il est à 1 si c'est le cas ;
- le **DPL** indique le niveau de privilège du segment. Le niveau 0 correspond au mode super-utilisateur ;
- le **bit S** est mis à 1 pour un descripteur de segment et à 0 pour un descripteur système (un genre particulier de descripteur que nous verrons plus tard) ;
- le **type** définit le type de segment (code, données ou pile).

IV-B-3-a - Descripteur d'un segment de code

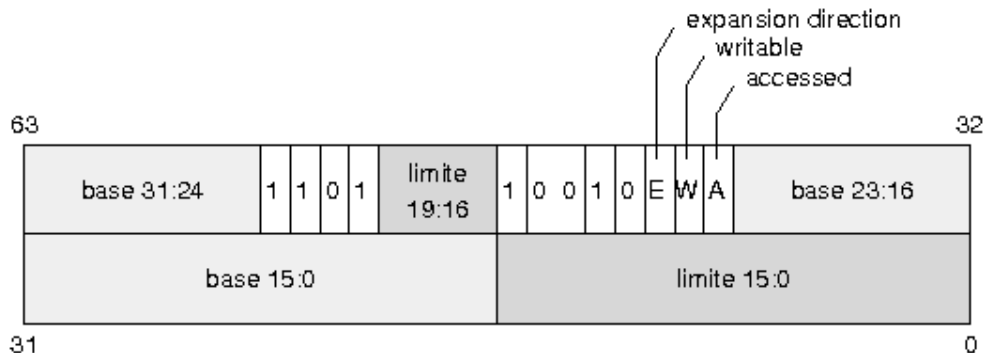


- Le bit G est mis à 1 (limite exprimée en pages).
- Le bit D/B est mis à 1 (code sur 32 bits).
- Le bit P est mis à 1 (page présente en mémoire).
- Le niveau de privilège est mis à 0 (mode super-utilisateur).
- Le bit S est mis à 1 (descripteur de segment).
- Le premier bit à 1 pour le type indique que l'on a affaire à un segment de code.
- Pour pouvoir adresser toute la mémoire, la base du segment doit être à 0x0 et sa limite doit être à 0xFFFFF avec le bit de granularité à 1.
- Le bit C indique si le segment de code est « conforment » ou non. Pour le moment, ce bit sera mis à 0.

Ce flag est complexe à manipuler. Il est en lien avec les différents niveaux de privilèges et de protection mis en œuvre par les microprocesseurs de type i386. La plupart des segments de code sont non conformants, ce qui signifie qu'ils peuvent transférer le contrôle (via un call ou un jmp) seulement à des segments de même privilège. La gestion des niveaux de protection sur architecture i386 est rendue très complexe par un foisonnement de mécanismes impossibles à résumer ici. Pour une étude approfondie, l'étude de la documentation de référence est indispensable...

- Le bit R est mis à 1 pour indiquer que le segment de code est accessible en lecture (en plus de l'être en exécution).
- Le bit A est mis à 1 par le processeur quand le segment est utilisé.

IV-B-3-b - Descripteur d'un segment de données



- Le segment de données se distingue du segment de code par le champ **type** avec le premier bit qui est mis à 0.

- Le bit E indique le sens d'expansion des données. Il est mis à 1 pour un segment de type « pile » dans lequel les données s'accumulent vers le début de la mémoire (**expand-down segment**).



Selon que le bit E est à 0 ou à 1, la limite s'interprète différemment. Pour un segment sur 32 bits, si le bit E est à 1, la limite supérieure de la plage de données est en 0xFFFFFFFF et la limite inférieure est à l'adresse indiquée par le champ limite. Note : à confirmer, mais il semble que la base soit dans ce cas purement ignorée.

- Le bit W est mis à 1 pour indiquer que le segment est accessible en écriture (en plus de l'être en lecture).
- Le bit A est mis à 1 par le processeur quand le segment est utilisé.

IV-C - Un boot loader qui passe en mode protégé

IV-C-1 - Quand passer en mode protégé ?

Il est possible de passer en mode protégé à plusieurs moments : lors de l'exécution du secteur de boot ou du noyau. Notre noyau va utiliser un jeu d'instructions sur 32 bits, seulement utilisable en mode protégé. Le plus simple est donc de passer en mode protégé pendant le boot, avant que le noyau ne s'exécute. Il est cependant possible que ce soit le noyau qui effectue la commutation, mais cela complique inutilement son écriture, car le code du noyau doit alors être en partie sur 16 bits et en partie sur 32 bits.

IV-C-2 - Le programme du boot loader

```
?fine BASE      0x100    ; 0x0100:0x0 = 0x1000
?fine KSIZE     50       ; nombre de secteurs à charger

[BITS 16]
[ORG 0x0]

jmp start
%include "UTIL.INC"
start:

; initialisation des segments en 0x07C0
mov ax, 0x07C0
mov ds, ax
mov es, ax
mov ax, 0x8000    ; stack en 0xFFFF
mov ss, ax
mov sp, 0xf000

; récupération de l'unité de boot
mov [bootdrv], dl

; affiche un msg
mov si, msgDebut
call afficher

; charger le noyau
xor ax, ax
int 0x13

push es
mov ax, BASE
mov es, ax
mov bx, 0
mov ah, 2
mov al, KSIZE
mov ch, 0
mov cl, 2
mov dh, 0
```



```

mov dl, [bootdrv]
int 0x13
pop es

; initialisation du pointeur sur la GDT
mov ax, gdtend      ; calcule la limite de GDT
mov bx, gdt
sub ax, bx
mov word [gdtptr], ax

xor eax, eax        ; calcule l'adresse linéaire de GDT
xor ebx, ebx
mov ax, ds
mov ecx, eax
shl ecx, 4
mov bx, gdt
add ecx, ebx
mov dword [gdtptr+2], ecx

; passage en modep
cli
lgdt [gdtptr]       ; charge la gdt
mov eax, cr0
or ax, 1
mov cr0, eax        ; PE mis a 1 (CR0)

jmp next
next:
mov ax, 0x10        ; segment de donne
mov ds, ax
mov fs, ax
mov gs, ax
mov es, ax
mov ss, ax
mov esp, 0x9F000

jmp dword 0x8:0x1000 ; réinitialise le segment de code

;-----
bootdrv: db 0
msgDebut: db "Chargement du kernel", 13, 10, 0
;-----
gdt:
db 0, 0, 0, 0, 0, 0, 0, 0, 0
gdt_cs:
db 0xFF, 0xFF, 0x0, 0x0, 0x0, 10011011b, 11011111b, 0x0
gdt_ds:
db 0xFF, 0xFF, 0x0, 0x0, 0x0, 10010011b, 11011111b, 0x0
gdtend:
;-----
gdtptr:
dw 0 ; limite
dd 0 ; base
;-----

;; NOP jusqu'a 510
times 510-($-$$) db 144
dw 0xAA55

```

IV-C-2-a - Que fait exactement ce programme ?

Ce programme est identique à celui du chapitre précédent avec en plus des instructions qui basculent le microprocesseur en mode protégé. Pour résumer, le numéro de périphérique de boot est placé dans une variable, les registres relatifs aux segments de code et de données sont initialisés, puis le noyau est chargé en mémoire à l'adresse 0x1000 (on note que la variable KSIZE a été augmentée afin de charger un noyau plus volumineux). Ensuite, la GDT est initialisée et chargée en mémoire, puis on bascule en mode protégé. Enfin, le noyau est exécuté.



Ce secteur de boot peut charger seulement des noyaux d'une taille limitée et bien inférieure à la capacité maximale d'une disquette. Nous verrons plus tard comment charger notre noyau à l'aide de GRUB.

IV-C-2-b - Passer en mode protégé

Avant de passer en mode protégé, il faut initialiser la GDT de façon à ce qu'il n'y ait pas de problème d'adressage après le changement de mode. La GDT doit contenir des descripteurs pour les segments de code, de données et de pile. Les directives ci-dessous déclarent et initialisent la GDT :

```
gdt:
    db 0, 0, 0, 0, 0, 0, 0, 0, 0
gdt_cs:
    db 0xFF, 0xFF, 0x0, 0x0, 0x0, 10011011b, 11011111b, 0x0
gdt_ds:
    db 0xFF, 0xFF, 0x0, 0x0, 0x0, 10010011b, 11011111b, 0x0
gdtend:
```

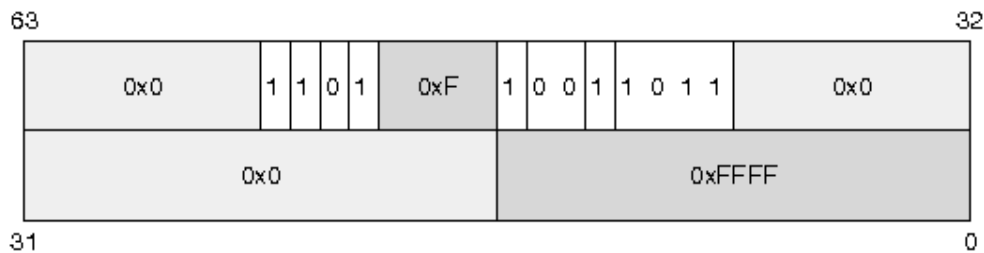
L'étiquette gdt: est un pointeur sur le début du tableau qui contient trois descripteurs :

- le premier descripteur ne doit pas être utilisé et les données sont mises à zéro. Il s'agit du descripteur NULL ;
- le deuxième descripteur, avec l'étiquette gdt_cs: décrit le segment de code ;
- le troisième descripteur, avec l'étiquette gdt_ds: décrit le segment de données.

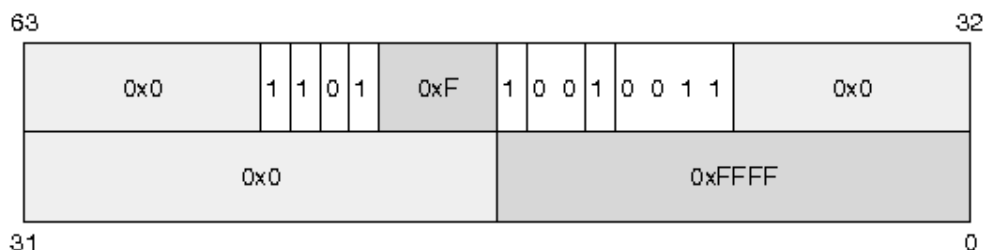
Chaque descripteur est initialisé de façon à pouvoir adresser l'ensemble de la RAM. La base de ces segments est à 0x0 avec une limite de 0xFFFF pages (le bit G est à 1).

Les schémas ci-dessous résument la façon dont sont initialisés les descripteurs.

Descripteur du segment de code



Descripteur du segment de données



La GDT est directement initialisée, mais avant de basculer en mode protégé, il faut renseigner le processeur pour qu'il prenne en compte la GDT. Cela se fait en mettant à jour le registre **GDT**, de 6 octets, qui contient l'adresse de la GDT et sa taille (on parle aussi de limite). On charge ce registre spécial avec l'instruction lgdt.

Dans le programme, `gdtptr` est un pointeur sur une structure qui contient les informations à charger dans le registre GDTR. La structure `gdtptr` est d'abord déclarée et initialisée à zéro (comme une variable classique en C) :

```
gdtptr:
    dw 0 ; limite
    dd 0 ; base
```

Ensuite, on calcule les valeurs pour mettre dans cette structure. Le code suivant calcule la taille de la GDT et stocke la valeur dans le premier champ de `gdtptr` :

```
; initialisation du pointeur sur la GDT
mov ax, gdtend ; calcule la limite de GDT
mov bx, gdt
sub ax, bx
mov word [gdtptr], ax
```

Ce code calcule l'adresse physique de la GDT en se basant sur les valeurs du segment de données `ds` et de l'adresse de l'étiquette `gdt`. Le résultat de ce calcul est stocké dans le second champ de `gdtptr` :

```
; calcule l'adresse linéaire de GDT
xor eax, eax
xor ebx, ebx
mov ax, ds
mov ecx, eax
shl ecx, 4
mov bx, gdt
add ecx, ebx
mov dword [gdtptr+2], ecx
```

Notre structure est donc maintenant correctement initialisée. Nous sommes maintenant presque prêt à passer en mode protégé. Avant cela, il faut inhiber les interruptions, car comme le système d'adressage va changer, les routines appelées par les interruptions ne seront plus valides après la bascule (il faudra les reprogrammer) :

```
; désactivation des interruptions
cli
```

Le registre GDTR est chargé avec l'instruction `lgdt` pour indiquer au microprocesseur où se trouve la GDT :

```
; charge la gdt
lgdt [gdtptr]
```

On peut maintenant passer en mode protégé :

```
; PE mis a 1 (CR0)
mov eax, cr0
or ax, 1
mov cr0, eax
```

Enfin ! :-)

Notre tâche semble terminée. Mais au fait... il reste encore à réinitialiser les sélecteurs de segment de code et de données ! La commande qui suit doit impérativement être la suivante afin de vider les caches internes du processeur :

```
jmp gdt_next
next:
```



En principe, il faudrait faire un far jump à la place du near jump ci-dessus pour réinitialiser le sélecteur de segment de code. Oui, mais voilà, le manuel spécifie : « When the processor is switched into protected mode, the original code segment base-address value of FFFF0000H (located in the hidden part of the CS register) is retained and execution continues from the

current offset in the EIP register. The processor will thus continue to execute code in the EPROM until a far jump or call is made to a new code segment, at which time, the base address in the CS register will be changed. »

Ensuite, on réinitialise les sélecteurs de données :

```
; segment de données
mov ax, 0x10
mov ds, ax
mov fs, ax
mov gs, ax
mov es, ax
```

Puis le segment de pile :

```
; la pile
mov ss, ax
mov esp, 0x9F000
```

L'instruction suivante réinitialise le sélecteur de code et exécute le noyau situé à l'adresse physique 0x1000. Cette instruction est essentielle, car elle permet, outre l'exécution du code du noyau, la réinitialisation correcte du sélecteur de code sur le bon descripteur (offset 0x8 dans la GDT) :

```
; réinitialise le segment de code
jmp dword 0x8:0x1000
```

Ensuite, le code du noyau s'exécute...

IV-D - Un noyau très simple

Ce noyau affiche juste un message de bienvenue et boucle ensuite indéfiniment. À ce stade, les routines du BIOS permettant d'afficher des caractères à l'écran ne sont plus utilisables, il faut donc que nous gérons nous-même l'affichage.

IV-D-1 - Le code du noyau

```
[BITS 32]
[ORG 0x1000]

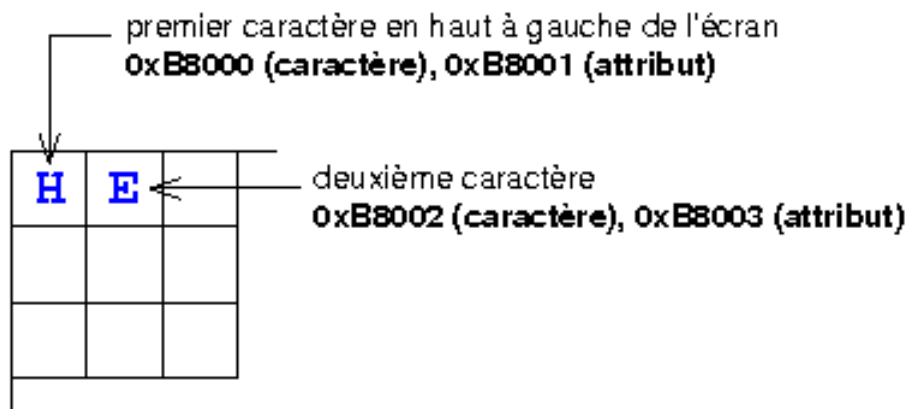
; Affichage d'un message par écriture dans la RAM vidéo
mov byte [0xB8A00], 'H'
mov byte [0xB8A01], 0x57
mov byte [0xB8A02], 'E'
mov byte [0xB8A03], 0x0A
mov byte [0xB8A04], 'L'
mov byte [0xB8A05], 0x4E
mov byte [0xB8A06], 'L'
mov byte [0xB8A07], 0x62
mov byte [0xB8A08], 'O'
mov byte [0xB8A09], 0x0E

end:
jmp end
```

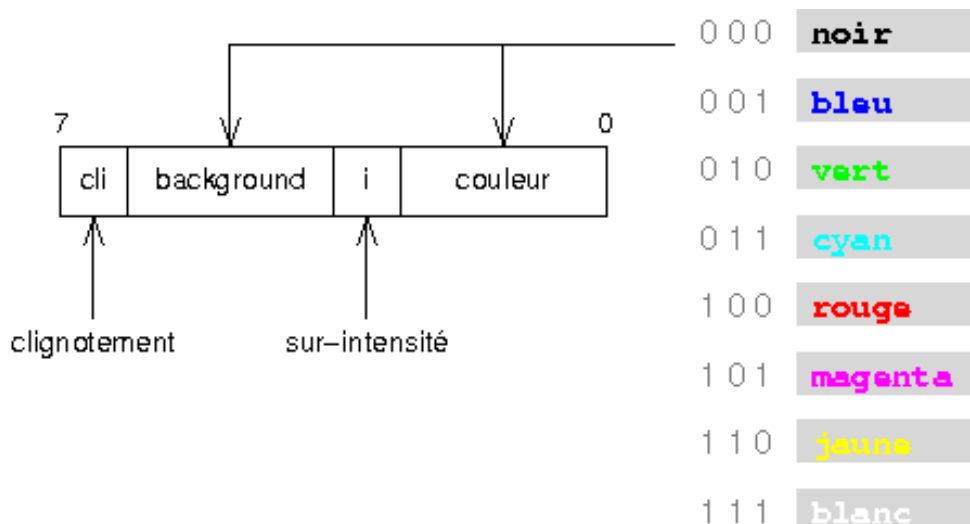
IV-D-2 - Afficher quelque chose à l'écran

La mémoire vidéo est mappée en mémoire à l'adresse physique 0xB8000. On peut donc afficher des informations en manipulant directement les octets débutant à cette adresse :

- la console d'affichage comprend 25 lignes et 80 colonnes.
- chaque caractère est décrit par 2 octets. Le premier contient le code ASCII du caractère à afficher et le suivant contient ses attributs (couleur, clignotement...)



Les attributs sont codés de la façon suivante :



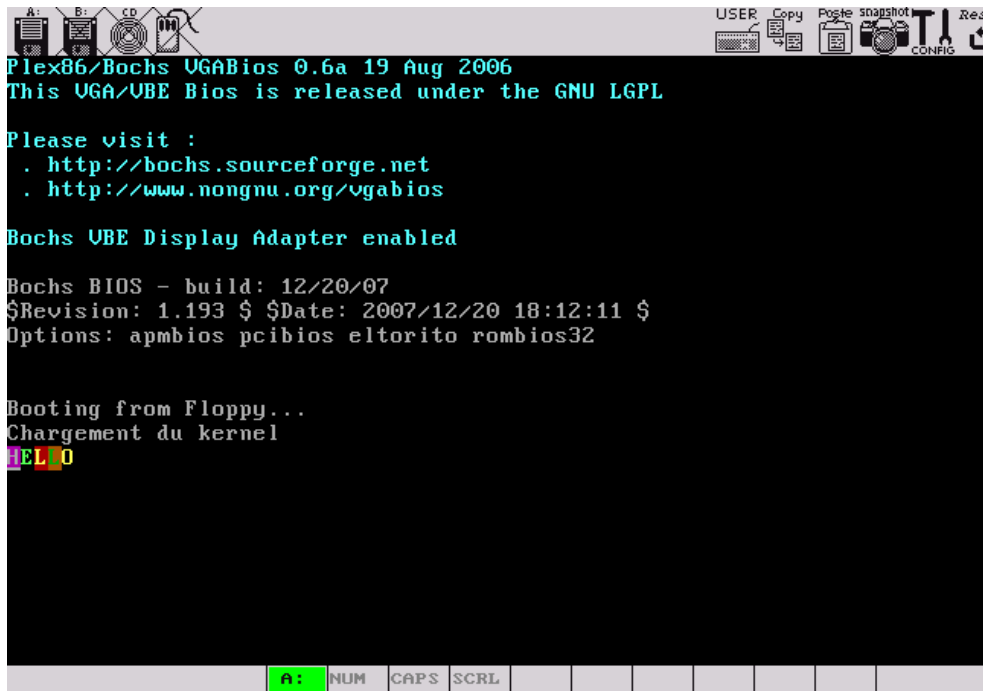
Par exemple, le code suivant affiche le caractère 'H' en blanc sur fond magenta en haut à gauche de l'écran :

```
mov byte [0xB8000], 'H'
mov byte [0xB8001], 0x57
```

IV-E - Compiler et tester le boot loader et le noyau

On compile le boot loader et le noyau séparément puis on crée la disquette :

```
nasm -f bin -o bootsect bootsect.asm
nasm -f bin -o kernel kernel.asm
cat bootsect kernel /dev/zero | dd of=floppyA bs=512 count=2880
```



V - Écrire un noyau en C

- Pourquoi passer de l'assembleur au C ?
- Un noyau en assembleur qui fait appel à des fonctions en C
- Un noyau écrit entièrement en C

Sources

Le package contenant toutes les sources est téléchargeable ici : [bootsect_kernelC.tgz](#).

V-A - Pourquoi passer de l'assembleur au C ?

La programmation en C offre par rapport à l'assembleur des avantages incontournables :

- concision de l'écriture ;
- facilité du débogage ;
- portabilité sur d'autres architectures.

Il est possible d'utiliser à peu près n'importe quel langage compilé pour écrire le code d'un noyau et chacun choisira le langage avec lequel il se sent le plus à l'aise (Pascal, C++...), la seule contrainte étant que ce langage doit permettre d'insérer des routines en assembleur et de manipuler directement les adresses en mémoire.

V-B - Un noyau en assembleur qui fait appel à des fonctions en C

V-B-1 - Des routines pour afficher quelque chose à l'écran en C

Le fichier `screen.c` contient des fonctions permettant d'afficher des caractères à l'écran :

```
#include "types.h"

#define RAMSCREEN 0xB8000 /* début de la mémoire vidéo */
#define SIZESCREEN 0xFA0 /* 4000, nombres d'octets d'une page texte */
```

```

#define SCREENLIM 0xB8FA0

char kX = 0;                               /* position courante du curseur à l'écran */
char kY = 17;
char kattr = 0x0E;                         /* attributs vidéo des caractères à afficher */

/*
 * 'scrollup' scrolle l'écran (la console mappée en ram) vers le haut
 * de n lignes (de 0 a 25).
 */
void scrollup(unsigned int n)
{
    unsigned char *video, *tmp;

    for (video = (unsigned char *) RAMSCREEN;
         video < (unsigned char *) SCREENLIM; video += 2) {
        tmp = (unsigned char *) (video + n * 160);

        if (tmp < (unsigned char *) SCREENLIM) {
            *video = *tmp;
            *(video + 1) = *(tmp + 1);
        } else {
            *video = 0;
            *(video + 1) = 0x07;
        }
    }

    kY -= n;
    if (kY < 0)
        kY = 0;
}

void putcar(uchar c)
{
    unsigned char *video;
    int i;

    if (c == 10) {                          /* CR-NL */
        kX = 0;
        kY++;
    } else if (c == 9) {                    /* TAB */
        kX = kX + 8 - (kX % 8);
    } else if (c == 13) {                  /* CR */
        kX = 0;
    } else {                                /* autres caractères */
        video = (unsigned char *) (RAMSCREEN + 2 * kX + 160 * kY);
        *video = c;
        *(video + 1) = kattr;

        kX++;
        if (kX > 79) {
            kX = 0;
            kY++;
        }
    }

    if (kY > 24)
        scrollup(kY - 24);
}

/*
 * 'print' affiche à l'écran, à la position courante du curseur, une chaîne
 * de caractères terminée par \0.
 */
void print(char *string)
{
    while (*string != 0) { /* tant que le caractère est différent de 0x0 */
        putcar(*string);
        string++;
    }
}

```


- Les variables `kX` et `kY` stockent en mémoire l'emplacement du curseur à l'écran. La variable `kattr` contient les attributs vidéo des caractères affichés.
- La fonction `scrollup()` prend en argument un entier `n` et scrolle l'écran de `n` lignes.
- La fonction `putcar()` affiche un caractère à l'écran.
- La fonction `print()` affiche une chaîne de caractères.

Notez que certains types ont été définis dans le fichier `types.h` :

```
#ifndef _I386_TYPE_
#define _I386_TYPE_

typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned int u32;
typedef unsigned char uchar;

#endif
```

V-B-2 - Un nouveau noyau en assembleur

Le code du noyau ci-dessous fait appel aux fonctions d'affichage définies ci-dessus. Cet exemple est assez intéressant, car il montre comment un programme en assembleur fait appel à une fonction écrite en C (ce sujet est développé dans l'annexe sur la [compilation séparée en assembleur sous Unix](#)) :

```
[BITS 32]

EXTERN scrollup, print
GLOBAL _start

_start:

    mov eax, msg
    push eax
    call print
    pop eax

    mov eax, msg2
    push eax
    call print
    pop eax

    mov eax, 2
    push eax
    call scrollup

end:
    jmp end

msg db 'un premier message', 10, 0
msg2 db 'un deuxième message', 10, 0
```

Par rapport aux précédents noyaux, on note surtout :

- l'absence de directive `ORG`, qui sert au calcul des adresses en fonction de l'endroit où le code est relogé. Ce calcul est maintenant réalisé par le linker `ld` ;
- la présence du point d'entrée `_start`, indispensable à `ld`.

V-B-3 - Compiler le noyau

```
gcc -c screen.c
nasm -f elf -o kernel.o kernel.asm
ld --oformat binary -Ttext 1000 kernel.o screen.o -o kernel
```

L'option `-Ttext` indique l'adresse linéaire à partir de laquelle le code commence. Par défaut, `ld` suppose que le code commence à l'adresse `0x0`. Ici, ce paramètre est indispensable, car le code du noyau est recopié par le secteur de boot en `0x1000`. La même fonctionnalité était implémentée par la directive `[ORG 0x1000]` dans les noyaux précédents en assembleur.

L'option `-Tdata`, qui sert à indiquer l'offset de début de la section de données, n'est pas utilisée. Par défaut, le linker considère que la zone de données suit la zone de texte (on remarque qu'elle est relogée une page mémoire plus loin).

V-C - Un noyau écrit entièrement en C

V-C-1 - Le code

```
extern void scrollup(unsigned int);
extern void print(char *);

extern kY;
extern kattr;

void _start(void)
{
    kY = 18;
    kattr = 0x5E;
    print("un message\n");

    kattr = 0x4E;
    print("un autre message\n");

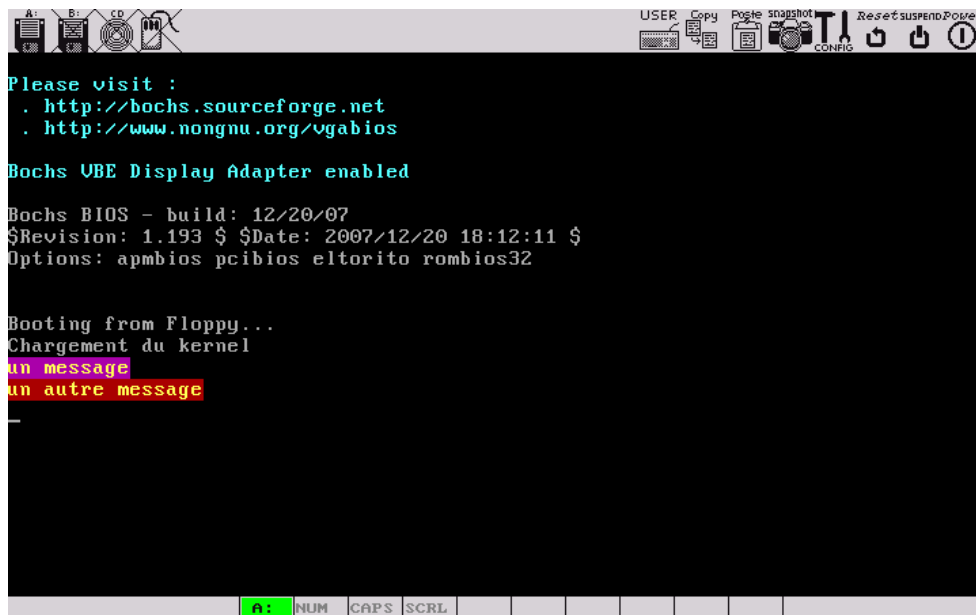
    scrollup(2);

    while (1);
}
```

Le code du noyau en C reprend de façon fidèle le code exprimé plus haut en assembleur.

V-C-2 - Compiler le noyau

```
gcc -c screen.c
gcc -c kernel.c
ld --oformat binary -Ttext 1000 kernel.o screen.o -o kernel
```



VI - Programmer un noyau en C qui recharge la GDT

- Pourquoi changer de GDT ?
- Un noyau qui recharge la GDT

Sources :

Le package contenant les sources est téléchargeable ici : [kernel_ReloadGDT.tgz](#)

VI-A - Pourquoi changer de GDT ?

Au boot, le programme du MBR commute le PC en mode protégé afin de pouvoir charger et exécuter un noyau 32 bits. Le problème est que la GDT initialisée par le secteur de boot ne correspond pas forcément à celle que l'on souhaite pour le noyau. Par exemple, si on démarre notre noyau à l'aide de LILO ou d'un autre boot loader, on ne sait pas à l'avance où sera la GDT ni comment elle sera constituée. Un noyau doit donc initialiser et charger sa propre GDT.

VI-B - Un noyau qui recharge la GDT

VI-B-1 - Le programme principal

Il affiche un message, initialise la nouvelle GDT et la charge en mémoire. Après avoir réinitialisé la pile, le noyau passe la main à la fonction `main()` qui affiche un message et boucle indéfiniment :

```
#include "types.h"
#include "gdt.h"
#include "screen.h"

int main(void);

void _start(void)
{
    kY = 18;
    kattr = 0x5E;
    print("kernel : loading new gdt...\n");

    /* initialisation de la GDT et des segments */
}
```

```

init_gdt();

/* Initialisation du pointeur de pile %esp */
asm("    movw $0x18, %ax \n \
      movw %ax, %ss \n \
      movl $0x20000, %esp");

main();
}

int main(void)
{
    kattr = 0x4E;
    print("kernel : new gdt loaded !\n");

    while (1);
}

```

VI-B-2 - Initialiser la GDT

VI-B-2-a - Les descripteurs de segment en C

Les structures ci-dessous, définies dans le fichier gdt.h, servent à créer les descripteurs de segment et le registre GDTR :

```

/* Descripteur de segment */
struct gtdesc {
    u16 lim0_15;
    u16 base0_15;
    u8 base16_23;
    u8 acces;
    u8 lim16_19 : 4;
    u8 other : 4;
    u8 base24_31;
} __attribute__((packed));

/* Registre GDTR */
struct gdtr {
    u16 limite ;
    u32 base ;
} __attribute__((packed));

```

La directive `__attribute__((packed))` indique à gcc que la structure en question doit occuper le moins de place possible en mémoire. Sans cette directive, le compilateur insère des octets entre les champs de la structure afin de les aligner pour optimiser la vitesse d'accès. Or dans notre cas, nous voulons que la structure décrive exactement l'occupation en mémoire des données.

Pour ces mêmes raisons, il faut définir de nouveaux types de données afin de maîtriser les allocations en mémoire. Ces types sont définis dans le fichier types.h.

```

#ifndef _I386_TYPE_
#define _I386_TYPE_

typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned int u32;
typedef unsigned char uchar;

#endif

```

VI-B-2-b - La fonction init_gdt()

Le fichier gdt.c contient la fonction `init_gdt()` qui initialise les descripteurs de segments et charge la nouvelle GDT :

```
#include "types.h"
#include "lib.h"

#define __GDT__
#include "gdt.h"

/*
 * 'init_desc' initialise un descripteur de segment situe en gdt ou en ldt.
 * 'desc' est l'adresse linéaire du descripteur à initialiser.
 */
void init_gdt_desc(u32 base, u32 limite, u8 acces, u8 other,
                  struct gtdesc *desc)
{
    desc->lim0_15 = (limite & 0xffff);
    desc->base0_15 = (base & 0xffff);
    desc->base16_23 = (base & 0xff0000) >> 16;
    desc->acces = acces;
    desc->lim16_19 = (limite & 0xf0000) >> 16;
    desc->other = (other & 0xf);
    desc->base24_31 = (base & 0xff000000) >> 24;
    return;
}

/*
 * Cette fonction initialise la GDT après que le kernel soit chargé
 * en mémoire. Une GDT est déjà opérationnelle, mais c'est celle qui
 * a été initialisée par le secteur de boot et qui ne correspond
 * pas forcément à celle que l'on souhaite.
 */
void init_gdt(void)
{
    /* initialisation des descripteurs de segment */
    init_gdt_desc(0x0, 0x0, 0x0, 0x0, &kgdt[0]);
    init_gdt_desc(0x0, 0xFFFFF, 0x9B, 0x0D, &kgdt[1]); /* code */
    init_gdt_desc(0x0, 0xFFFFF, 0x93, 0x0D, &kgdt[2]); /* data */
    init_gdt_desc(0x0, 0x0, 0x97, 0x0D, &kgdt[3]); /* stack */

    /* initialisation de la structure pour GDTR */
    kgdtr.limite = GDTSIZE * 8;
    kgdtr.base = GDTBASE;

    /* recopie de la GDT à son adresse */
    memcpy((char *) kgdtr.base, (char *) kgdt, kgdtr.limite);

    /* chargement du registre GDTR */
    asm("lgdtl (kgdtr)");

    /* initialisation des segments */
    asm("    movw $0x10, %ax \n \
        movw %ax, %ds      \n \
        movw %ax, %es      \n \
        movw %ax, %fs      \n \
        movw %ax, %gs      \n \
        ljmp $0x08, $next   \n \
        next:               \n");
}
```

Les descripteurs sont initialisés et copiés dans le tableau kgdt[] :

```
init_gdt_desc(0x0, 0xFFFFF, 0x9B, 0x0D, &kgdt[1]); /* code */
init_gdt_desc(0x0, 0xFFFFF, 0x93, 0x0D, &kgdt[2]); /* data */
init_gdt_desc(0x0, 0x0, 0x97, 0x0D, &kgdt[3]); /* stack */
```

- Les segments de code et de données ont leur base qui est à 0 et leur limite est de $0xFFFFF + 1 = 0x100000$ pages de 4 ko, soient 4 Go. Autrement dit, ils adressent l'ensemble de la mémoire.
- De façon un peu étonnante, le segment de pile a une base et une limite qui sont à 0 ! Dans un segment de pile (**expand down**), la base n'est pas interprétée, elle est donc ici mise à 0. La limite se calcule comme

pour les autres segments, mais elle doit être interprétée comme la limite inférieure du segment. Dans le cas présent, le segment de pile recouvre donc toute la mémoire.

Une fois le tableau rempli, il est recopié à l'endroit en mémoire où la GDT doit résider :

```
/* recopie de la GDT a son adresse */
memcpy((char *) kgdtr.base, (char *) kgdt, kgdtr.limite);
```

Ensuite, la structure `kgdtr` est initialisée puis chargée dans le registre GDTR. À ce moment-là, le changement de GDT est effectif :

```
/* chargement du registre GDTR */
asm("lgdtl (%kgdtr)");
```

Une fois la nouvelle GDT chargée, il faut mettre à jour les sélecteurs de segments de données (`ds`, `es`, `fs`, `gs` et `ss`). Un **long jump** permet de mettre à jour le sélecteur du segment de code :

```
/* initialisation des segments */
asm("    movw $0x10, %ax    \n \
      movw %ax, %ds \n \
      movw %ax, %es \n \
      movw %ax, %fs \n \
      movw %ax, %gs \n \
      ljmp $0x08, $next    \n \
      next:                \n");
```

***I** Vous avez sans doute remarqué que le pointeur de pile est initialisé après l'appel à la fonction `init_gdt()`. Pourquoi n'est-il pas initialisé dans `init_gdt()` comme tous les autres ? Parce que l'instruction assembleur `leave`, en fin de fonction, écrase le registre `esp` avec la valeur de `ebp`. Tout serait alors à refaire ! Une solution serait de forcer la valeur de `ebp` de façon à ce qu'elle coïncide avec celle de `esp`, mais cela ne ferait que repousser le problème : n'oubliez pas qu'en changeant la pile, on perd l'adresse sauvegardée du compteur ordinal (`eip`) qui permet le retour.*

VI-B-3 - Une fonction `main()` pour déjouer les pièges de gcc

```
/* initialisation de la GDT et des segments */
init_gdt();

/* Initialisation du pointeur de pile %esp */
asm("    movw $0x18, %ax \n \
      movw %ax, %ss \n \
      movl $0x20000, %esp");

main();
```

Après avoir initialisé la GDT, la pile est initialisée pour pointer en `0x20000`. Cette adresse est arbitraire, j'aurais pu choisir autre chose... mais attention à prendre une valeur où la pile ne risque pas de corrompre le code ou des données !

Après ces initialisations, une fonction `main()` est appelée. La création d'une nouvelle fonction peut sembler luxueuse quand on voit ce qu'elle réalise : juste afficher un message et boucler indéfiniment. N'aurait-on pas pu placer l'intégralité du code dans la fonction `_start()` ? Non, car l'appel à la fonction `print` est réalisé par gcc de cette façon :

```
movl    $.LC1, (%esp)
call    print
```

On remarque que le passage d'argument ne se fait pas par un `push`, mais par un `mov` qui écrit l'adresse de la chaîne de caractères à afficher directement en `0x20000`, ce qui est au-delà du sommet de la pile ! En principe, pour passer un paramètre à une fonction, un compilateur doit utiliser `push` qui décrémente d'abord la valeur de `esp` avant d'écrire sur la pile, mais `gcc` procède autrement en réservant au début de la fonction suffisamment de place sur la pile. Mais comme nous avons modifié entretemps la structure de la pile, et donc de la frame associée à la fonction `_start`, cela ne peut plus fonctionner ! La fonction `main()` permet de repartir sur une **frame** propre.

VI-B-4 - Compiler le bootloader et le noyau

```
tar xfz kernel_ReloadGDT.tgz
cd ReloadGDT
make
```



VII - Programmer les interruptions du processeur i386 avec le contrôleur d'interruptions 8259A

- **Des interruptions pour prévenir le processeur d'un événement**
- **Un chipset pour gérer les interruptions matérielles : le 8259A**
- **Exécuter la bonne routine de service grâce à la table des descripteurs d'interruptions**

VII-A - Des interruptions pour prévenir le processeur d'un événement

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

- 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 ;
- les **interruptions logicielles** sont déclenchées volontairement par le programme. Elles permettent de gérer les appels système ;
- les **exceptions** sont déclenchées par le processeur en cas de faute (division par zéro, défaut de page, etc.).

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.

VII-A-1 - Un exemple d'interruption

Quand on presse ou qu'on relâche une touche du clavier d'un PC, le contrôleur du clavier envoie une interruption à un chipset chargé de multiplexer les interruptions : le **contrôleur d'interruptions**. Si l'interruption en question n'est pas masquée, le contrôleur déclenche une interruption matérielle pour prévenir le processeur. Le processeur exécute alors une routine pour traiter l'événement (touche pressée ou relâchée). En général, cette routine va interroger le contrôleur du clavier pour savoir quelle touche a été pressée (ou relâchée), puis il va éventuellement afficher le caractère correspondant et/ou le stocker dans un buffer. Une fois la routine de traitement de caractère terminée, la tâche interrompue peut reprendre.

VII-B - Un chipset pour gérer les interruptions matérielles : le 8259A

VII-B-1 - À quoi sert le 8259A ?

Les interruptions matérielles peuvent être vues comme des sonnettes 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 controller**) est un chipset qui permet d'éviter cela en multiplexant les requêtes d'interruption envoyées par les périphériques. Le 8259A est l'un des premiers PIC développés par Intel.

Chaque 8259A peut gérer les interruptions de 8 périphériques, mais la plupart des PC ont deux contrôleurs cascades entre eux, ce qui leur permet de gérer jusqu'à 14 périphériques. Le second contrôleur (esclave) est chaîné au premier (contrôleur maître). Un contrôleur peut réaliser des fonctions plus complexes que la simple transmission d'interruptions : il peut masquer des interruptions et définir des priorités de traitement.

Sur les machines modernes, les PIC 8259 sont remplacés par les **APIC** rétrocompatibles. https://en.wikipedia.org/wiki/Advanced_Programmable_Interrupt_Controller

VII-B-2 - Comment le 8259A traite les interruptions

Quand un périphérique transmet au 8259A une requête pour une interruption, celle-ci subit un traitement complexe :

- 1 Le registre **IMR (Interrupt Mask Register)** interne au contrôleur permet de masquer certaines interruptions. Si l'interruption est masquée, on ne la traite pas ;
- 2 Le registre **IRR (Interrupt Request Register)** est chargé de gérer les priorités entre interruptions au cas où plusieurs d'entre elles surviennent en même temps. La requête avec la plus haute priorité est évaluée ;
- 3 Le 8259A signale au processeur la demande d'interruption. La requête faite par le contrôleur au processeur est appelée **IRQ (Interrupt Request)** ;
- 4 Le processeur acquitte ce signal puis demande au contrôleur, par un autre signal, le numéro de l'interruption demandée afin de déclencher la bonne routine **ISR** ;
- 5 Le contrôleur dépose sur le bus de données un octet, le **vecteur d'interruption** ;
- 6 Une fois la routine terminée, le processeur doit envoyer au contrôleur un signal pour l'avertir que le traitement de l'interruption est terminé.

VII-B-3 - Comment programmer le 8259A ?

Il est possible de paramétrer les deux PIC d'un PC en écrivant dans leurs registres internes via les ports d'entrée/sortie du processeur. On accède au contrôleur maître via les ports 0x20 et 0x21 et au contrôleur esclave via les ports 0xA0 et 0xA1. En écrivant sur ces ports, on envoie au contrôleur des données qui seront inscrites dans l'un de ses registres.

Il y a deux types de registres :

- les registres **ICW (Initialization Command Word)**, qui réinitialisent le contrôleur ;
- les registres **OCW (Operation Control Word)**, qui permettent de paramétrer certaines fonctions du contrôleur une fois que celui-ci a été réinitialisé.

Pour réinitialiser un contrôleur, il faut remplir les registres d'initialisation ICW1, ICW2, ICW3 et ICW4 en respectant cet ordre. Après avoir transmis les données pour le registre ICW1, le contrôleur attend une valeur à mettre dans le registre ICW2 et ainsi de suite. S'il n'y a qu'un seul contrôleur, ce n'est pas la peine de renseigner les registres ICW3 et ICW4.

Les registres OCW peuvent être remplis à n'importe quel moment après l'initialisation des PIC. Le registre OCW1 permet notamment de masquer et démasquer les interruptions.

VII-B-3-a - Registres ICW

ICW1 (port 0x20 / port 0xA0)

```
|0|0|0|1|x|0|x|x|
| | | +--- avec ICW4 (1) ou sans (0)
| +----- un seul contrôleur (1), ou cascades (0)
+----- déclenchement par niveau (level) (1) ou par front (edge) (0)
```

Par exemple, pour mettre à jour le registre ICW1 du contrôleur maître et du contrôleur esclave :

```
mov al, 0x11 ; Initialisation de ICW1
out 0x20, al ; maître
out 0xA0, al ; esclave
```

ICW2 (port 0x21 / port 0xA1) :

```
|x|x|x|x|x|0|0|0|
| | | | |
+----- adresse de base des vecteurs d'interruption
```

Les bits de poids fort servent à calculer l'adresse de base du vecteur d'interruption. Cette valeur correspond à un offset dans la table IDT (voir plus loin). Par exemple, pour initialiser les adresses de base des vecteurs d'interruption des deux contrôleurs :

```
mov al, 0x20
out 0x21, al ; maître, vecteur de départ = 32
mov al, 0x70
out 0xA1, al ; esclave, vecteur de départ = 96
```

Dans l'exemple ci-dessus, les IRQ 0-7 sont mappées pour utiliser les interruptions 0x20-0x27 et les IRQ 8-15 sont mappées sur les interruptions 0x70-0x77. Note : sur les architectures de type x86, les 32 premiers vecteurs (0x0-0x19) sont réservés à la gestion des exceptions. Cela explique pourquoi les IRQ ne sont pas mappées à partir du début de la table.

ICW3 (port 0x21 / port 0xA1) :

Ce registre informe les contrôleurs de la façon dont ils sont connectés entre eux :

```
|x|x|x|x|x|x|x|x| pour le maître
| | | | | | |
+----- contrôleur esclave rattaché à la broche d'interruption (1), ou non (0)
```

Remarque : un contrôleur maître peut être rattaché à plusieurs contrôleurs esclaves.

```
|0|0|0|0|0|x|x|x| pour l'esclave
| | |
+----- Identifiant de l'esclave, qui correspond au numéro de broche IR sur le
maître
```

On détermine ici par quelles broches communiquent les deux contrôleurs. Dans l'exemple suivant, le contrôleur esclave est rattaché à la broche 2 du maître :

```
mov al, 0x04 ; maître
out 0x21, al
mov al, 0x02 ; esclave
out 0xA1, al
```

ICW4 (port 0x21 / port 0xA1) :

ce registre spécifie dans quel mode doit fonctionner le contrôleur. Ceci vient du fait que le 8259A a été conçu pour être générique et supporter différents systèmes :

```
|0|0|0|x|x|x|x|1|
| | | +----- mode "automatic end of interrupt" AEOI (1)
| | +----- mode bufferisé esclave (0) ou maître (1)
| +----- mode bufferisé (1)
+----- mode "fully nested" (1)
```

Dans le cas présent, nous avons seulement besoin du mode par défaut :

```
mov al, 0x01
out 0x21, al
out 0xA1, al
```

Si on initialise les registres d'un contrôleur les uns à la suite des autres, une petite temporisation est nécessaire :

```
mov al, 0x11 ; initialisation de ICW1
out 0x20, al
jmp .1      ; temporisation
.1:
mov al, 0x20 ; initialisation de ICW2
out 0x21, al ; vecteur de départ = 32
jmp .2      ; temporisation
.2:
mov al, 0x04 ; initialisation de ICW3
out 0x21, al
jmp .3
.3:
mov al, 0x01 ; initialisation de ICW4
out 0x21, al
```

VII-B-3-b - Registres OCW

OCW1 (port 0x21 / port 0xA1) :

```
|x|x|x|x|x|x|x|x|
| | | | | | | |
+----- pour chaque IRQ : masque d'interruption établi (1) ou non (0)
```

Pour modifier le registre IMR permettant de masquer les interruptions :

```
in al, 0x21 ; lecture de l'Interrupt Mask Register (IMR)
and al, 0xEF ; 0xEF => 11101111b. débloque l'IRQ 4
```

```
out 0x21, al ; recharge l'IMR
```

OCW2 (port 0x20 / port 0xA0) :

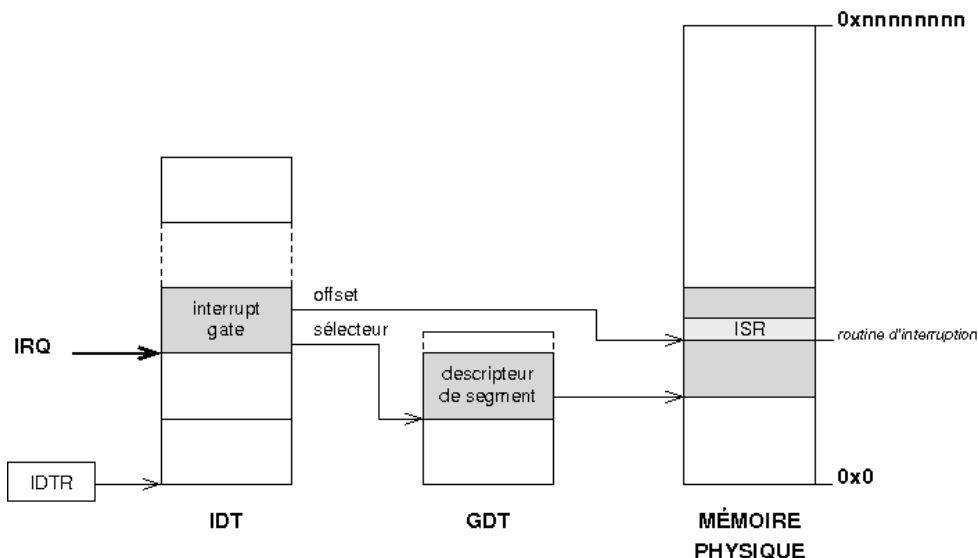
Ce registre est utilisé essentiellement pour avertir le contrôleur que l'interruption en cours a été traitée et qu'il peut rétablir les interruptions matérielles :

```
mov al, 0x20
out 0x20, al ; "End Of Interrupt" (EOI) envoyé au PIC
```

Pour une description des registres OCW2 et OCW3, non utilisés dans le noyau, consultez la documentation de référence.

VII-C - Exécuter la bonne routine de service grâce à la table des descripteurs d'interruptions

La table **IDT (Interrupt Descriptor Table)** permet d'associer à chaque interruption une routine de service. Chaque entrée de l'IDT est un descripteur particulier qui pointe sur une routine. Lorsqu'une interruption survient, le contrôleur transmet au processeur un **vecteur d'interruption**. Le processeur l'utilise pour calculer un offset dans l'IDT, charge un descripteur système, puis active la routine de service correspondant :

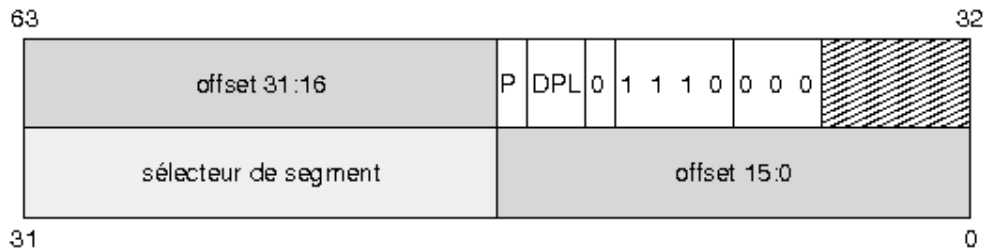


Les descripteurs dans l'IDT utilisés pour gérer les interruptions sont des **descripteurs système** d'un type particulier : les **interrupt gate**.



Il est aussi possible d'utiliser des descripteurs du type trap gate. La seule différence est que ces derniers ne désactivent pas les interruptions (via le bit IF du registre EFLAGS) lorsqu'une interruption est reçue.

VII-C-1 - Descripteur système de type Interrupt Gate



- Le **bit P** est utilisé pour déterminer si le segment est présent en mémoire physique. Il est à 1 si c'est le cas.
- Le **DPL** indique le niveau de privilège du segment. Le niveau 0 correspond au mode super-utilisateur.

VIII - Gérer les interruptions - la mise en œuvre

- **Préalable : étendre le code C du noyau à l'aide de directives en assembleur**
- **Initialiser la table IDT**
- **Le programme principal du noyau**

Sources

Le package contenant les sources est téléchargeable ici : [kernel_ManageINT.tgz](#)

Notez que vous pouvez utiliser la commande `diff` pour visualiser les parties modifiées ou ajoutées par rapport aux sources du chapitre précédent :

```
diff -u -r -N ReloadGDT/ ManageINT/
```

VIII-A - Préalable : étendre le code C du noyau à l'aide de directives en assembleur

Un certain nombre d'instructions existent en assembleur, et sont nécessaires à l'écriture de notre noyau, mais n'ont pas d'équivalent en C. C'est notamment le cas des instructions `cli`, `sti`, `in` et `out`. Les macros définies dans le fichier `io.h` permettent de contourner cette difficulté en étendant le jeu d'instructions.

```
/* désactive les interruptions */
#define cli asm("cli::")

/* réactive les interruptions */
#define sti asm("sti::")

/* écrit un octet sur un port */
#define outb(port,value) \
    asm volatile ("outb %al, %dx" :: "d" (port), "a" (value));

/* écrit un octet sur un port et marque une temporisation */
#define outbp(port,value) \
    asm volatile ("outb %al, %dx; jmp 1f; 1:" :: "d" (port), "a" (value));

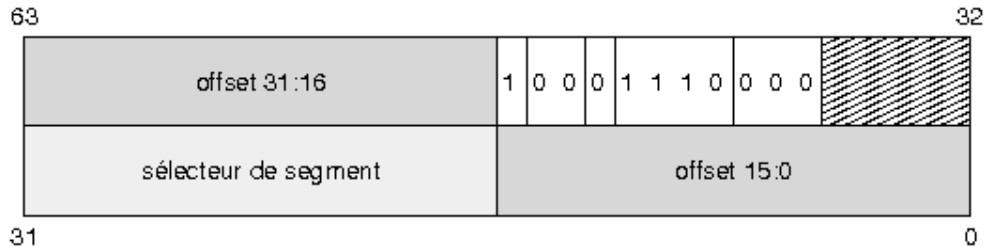
/* lit un octet sur un port */
#define inb(port) ({ \
    unsigned char _v; \
    asm volatile ("inb %dx, %al" : "=a" (_v) : "d" (port)); \
    _v; \
})
```

Nous avons déjà vu dans les chapitres précédents comment inclure de l'assembleur dans le code C de gcc avec la directive `asm()`. Il faut noter la particularité suivante : gcc se base sur gas qui utilise la syntaxe **AT&T**.

VIII-B - Initialiser la table IDT

VIII-B-1 - Créer les descripteurs système de type Interrupt Gate

Les descripteurs système de l'IDT gérant les interruptions, dont le fonctionnement est expliqué au [chapitre précédent](#), ont ce modèle :



La structure ci-dessous, définie dans le fichier `idt.h`, sert à créer les descripteurs d'interruption. Notez là encore la présence de la directive `__attribute__((packed))` pour empêcher gcc d'insérer des octets de padding au sein de la structure :

```
/* descripteur de segment */
struct idtdesc {
    u16 offset0_15;
    u16 select;
    u16 type;
    u16 offset16_31;
} __attribute__((packed));
```

La fonction `init_idt_desc()` sert à initialiser les descripteurs système.

```
void init_idt_desc(u16 select, u32 offset, u16 type, struct idtdesc* desc) {
    desc->offset0_15 = (offset & 0xffff);
    desc->select = select;
    desc->type = type;
    desc->offset16_31 = (offset & 0xffff0000) >> 16;
    return;
}
```

Par exemple, pour initialiser le descripteur associé à l'IRQ 1 (interruptions clavier), on utilise le code suivant :

```
init_idt_desc(0x08, (u32) _asm_irq_1, INTGATE, &kidt[33]); /* clavier */
```

- le premier argument pointe sur le descripteur de segment de code qui contient la routine de gestion de l'interruption (dans notre cas le descripteur est à l'offset 0x08 dans la GDT) ;
- le deuxième argument est l'offset par rapport au début du segment de code pour trouver le début de la routine. Cet offset correspond au nom de la fonction à exécuter ;
- le troisième argument comprend le type de descripteur dans l'IDT (ici [interrupt gate](#)) et le niveau de privilège du segment ;
- le dernier argument correspond au descripteur à initialiser.

VIII-B-2 - Créer une routine d'interruption (ISR)

Lors d'une interruption, un vecteur est transmis par le PIC au processeur pour exécuter la bonne ISR. Une ISR obéit à deux contraintes particulières :

- une fois l'interruption traitée, il faut avertir le contrôleur de la fin du traitement de l'interruption en lui envoyant un message **End of Interrupt** (EOI) ;

- une routine d'interruption doit retourner en utilisant la directive assembleur `iret` (au lieu de `ret`, utilisé habituellement lors de l'appel à une fonction).

Pour envoyer au contrôleur un message EOI, indiquant que l'interruption en cours a été traitée, on utilise le code assembleur suivant :

```
; envoyer EOI au PIC
mov al, 0x20
out 0x20, al
```

En revanche, pour retourner de la routine d'interruption, rien en C ne nous permet d'utiliser `iret`. La routine d'interruption appelée doit donc être écrite en assembleur. Par exemple, pour gérer l'interruption de l'IRQ 1 :

```
_asm_irq_1:
call isr_kbd_int
mov al, 0x20 ; EOI
out 0x20, al
iret
```

Ce code appelle la fonction `isr_kbd_int()` qui effectue réellement la gestion de l'interruption du clavier, puis elle envoie un EOI au contrôleur avant de retourner. La partie de code en assembleur ne fait donc qu'encapsuler le code écrit en C dans le fichier `interrupt.c`.

```
#include "types.h"
#include "screen.h"

void isr_default_int(void)
{
    print("interrupt\n");
}

void isr_clock_int(void)
{
    static int tic = 0;
    static int sec = 0;
    tic++;
    if (tic % 100 == 0) {
        sec++;
        tic = 0;
        print("clock\n");
    }
}

void isr_kbd_int(void)
{
    print("keyboard\n");
}
```

Les routines en assembleur sont dans le fichier `int.asm`.

```
extern isr_default_int, isr_clock_int, isr_kbd_int
global _asm_default_int, _asm_irq_0, _asm_irq_1

_asm_default_int:
call isr_default_int
mov al, 0x20
out 0x20, al
iret

_asm_irq_0:
call isr_clock_int
mov al, 0x20
out 0x20, al
iret

_asm_irq_1:
```

```
call isr_kbd_int
mov al,0x20
out 0x20,al
iret
```



Est-on vraiment obligés d'utiliser, pour chaque routine d'interruption, ces enveloppes en assembleur et n'est-il pas plus rapide d'inclure dans le code en C une directive du type `asm("iret")` ? Le problème est qu'en sortant d'une fonction de cette façon, on oublie de restaurer convenablement les registres, dont le pointeur de pile `esp`.

VIII-B-3 - Initialiser l'IDT avec la fonction `init_idt()`

La fonction `init_idt()` effectue l'initialisation des descripteurs système et le chargement de l'IDT. Étant donné leur similarité, le code nécessaire pour initialiser et charger l'IDT ressemble beaucoup à celui utilisé pour l'initialisation de la GDT. À l'exception des descripteurs associés aux IRQ 0 et 1 (respectivement l'horloge et le clavier), l'ensemble des descripteurs pointent vers un handler d'interruption par défaut :

```
for (i = 0; i < IDTSIZE; i++) /* default */
    init_idt_desc(0x08, (u32) _asm_default_int, INTGATE, &kidt[i]);

init_idt_desc(0x08, (u32) _asm_irq_0, INTGATE, &kidt[32]); /* horloge */
init_idt_desc(0x08, (u32) _asm_irq_1, INTGATE, &kidt[33]); /* clavier */
```

VIII-C - Le programme principal du noyau

```
#include "types.h"
#include "gdt.h"
#include "screen.h"
#include "io.h"
#include "idt.h"

void init_pic(void);

int main(void);

void _start(void)
{
    kY = 16;
    kattr = 0x0E;

    init_idt();
    print("kernel : idt loaded\n");

    init_pic();
    print("kernel : pic configured\n");

    /* initialisation de la GDT et des segments */
    init_gdt();

    /* Initialisation du pointeur de pile %esp */
    asm("    movw $0x18, %ax \n \
        movw %ax, %ss \n \
        movl $0x20000, %esp");

    main();
}

int main(void)
{
    print("kernel : gdt loaded\n");

    sti;
```

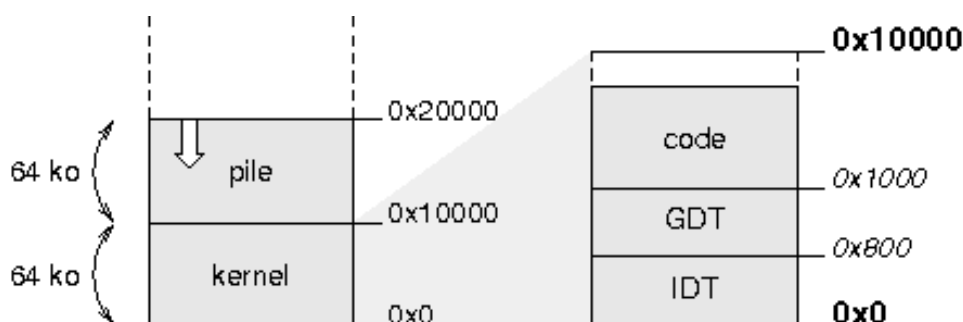


```
katrr = 0x47;
print("kernel : allowing interrupt\n");
katrr = 0x07;

while (1);
}
```

- La fonction `init_gdt()` réinitialise la GDT puis la fonction `init_idt()` configure et charge la table IDT des descripteurs d'interruptions.
- Les chipsets 8259A (maître et esclave) sont reprogrammés par la fonction `init_pic()`. Cette fonction, définie dans le fichier `pic.c`, initialise les contrôleurs d'interruption maître et esclave de la façon décrite au [chapitre précédent](#).
- Une fois les PIC configurés et la table IDT en mémoire, on réactive les interruptions avec la macro `sti`.

Le schéma ci-dessous résume l'organisation des données en mémoire physique après les initialisations :



VIII-C-1 - Compiler et exécuter le noyau

```
tar xzf kernel_ManageINT.tgz
cd ManageINT
make
```

À l'exécution du noyau, un **tic** est affiché toutes les 100 interruptions de l'horloge. En revanche, pour le moment, l'appui d'une touche du clavier ne fonctionne qu'une seule fois. Nous verrons plus tard comment gérer correctement le clavier :



IX - Gérer les interruptions du clavier

- **Le chipset 8042**
- **La gestion du curseur**

Sources

Le package contenant les sources : **kernel_ManageKBD.tgz**

IX-A - Le chipset 8042

Le clavier contient un microcontrôleur chargé de gérer les différents événements (pression, relâchement des touches et affichages lumineux) : le 8042. Les ports d'entrée/sortie utilisés pour communiquer avec ce chipset sont les suivants :

- 60h pour la lecture ou la transmission de données ;
- 64h pour connaître le statut ou émettre des commandes.

Quand une touche est pressée ou relâchée, le contrôleur écrit dans son registre de données un code appelé **scan code**. Le contenu de ce registre est accessible via le port 60h :

```
i = inb(0x60);
```

La routine suivante attend que le buffer de sortie du 8042 soit plein puis stocke le contenu de celui-ci :

```
do {  
    i = inb(0x64);  
} while((i & 0x01) == 0);  
  
i = inb(0x60);
```

Il existe deux types de scan code :

- la pression d'une touche génère un **make code** ;
- le relâchement d'une touche génère un **break code** (break_code = make_code + 0x80).

La routine d'interruption complète pour la gestion des interruptions du clavier : **isr_kbd_int()**

```
#include "types.h"  
#include "screen.h"  
#include "io.h"  
#include "kbd.h"  
  
void isr_default_int(void)  
{  
    print("interrupt\n");  
}  
  
void isr_clock_int(void)  
{  
    static int tic = 0;  
    static int sec = 0;  
    tic++;  
    if (tic % 100 == 0) {  
        sec++;  
        tic = 0;  
    }  
}
```

```
void isr_kbd_int(void)
{
    uchar i;
    static int lshift_enable;
    static int rshift_enable;
    static int alt_enable;
    static int ctrl_enable;

    do {
        i = inb(0x64);
    } while ((i & 0x01) == 0);

    i = inb(0x60);
    i--;

    /// putcar('\n'); dump(&i, 1); putcar(' ');

    if (i < 0x80) { /* touche enfoncée */
        switch (i) {
            case 0x29:
                lshift_enable = 1;
                break;
            case 0x35:
                rshift_enable = 1;
                break;
            case 0x1C:
                ctrl_enable = 1;
                break;
            case 0x37:
                alt_enable = 1;
                break;
            default:
                putcar(kbdmap
                    [i * 4 + (lshift_enable || rshift_enable)]);
        }
    } else { /* touche relâchée */
        i -= 0x80;
        switch (i) {
            case 0x29:
                lshift_enable = 0;
                break;
            case 0x35:
                rshift_enable = 0;
                break;
            case 0x1C:
                ctrl_enable = 0;
                break;
            case 0x37:
                alt_enable = 0;
                break;
        }
    }

    show_cursor();
}
```

IX-B - La gestion du curseur

Le curseur est géré en s'adressant directement au contrôleur VGA via le port 0x3d4 (contrôle) et le port 0x3d5 (données). La routine qui affiche le curseur à l'écran est très simple :

```
void move_cursor (u8 x, u8 y)
{
    u16 c_pos;

    c_pos = y * 80 + x;
    outb(0x3d4, 0x0f);
    outb(0x3d5, (u8) c_pos);
}
```

```
    outb(0x3d4, 0x0e);
    outb(0x3d5, (u8) (c_pos >> 8));
}

void show_cursor (void)
{
    move_cursor(kX, kY);
}
```



X - Créer une tâche : un noyau qui exécute une tâche utilisateur

- **Mode noyau et mode utilisateur**
- **Définir des segments de mémoire distincts**
- **Copier le code exécutable en RAM**
- **Créer et initialiser un TSS**
- **Exécuter une tâche**
- **Sauvegarder le contexte d'une tâche**
- **Compiler et exécuter le noyau**

Sources

Le package contenant les sources est téléchargeable ici : [kernel_MonoTask.tgz](#)

X-A - Mode noyau et mode utilisateur

X-A-1 - Pourquoi ?

Nous avons vu comment créer un programme qui peut à tout moment accéder à n'importe quel endroit de la mémoire et exécuter n'importe quelle instruction. Cette liberté a un prix : le risque de corrompre des données ou d'exécuter une instruction indésirable. Si ce risque est un mal nécessaire pour le noyau qui doit nécessairement pouvoir tout faire, il n'est pas souhaitable que le code d'un utilisateur bénéficie des mêmes permissions. Ce dernier ne doit pas pouvoir accéder aux données des autres utilisateurs ou corrompre l'ensemble du système (intentionnellement ou non).

On distingue deux types de modes d'exécution d'un programme : le **mode noyau** (ou **privilegié** ou encore **superviseur**) et le **mode utilisateur** :

- le code en **mode noyau** a un accès total à la machine (mémoire, instructions, périphériques) ;
- le code en **mode utilisateur** n'a qu'un accès limité à la mémoire et il ne peut pas exécuter certaines instructions dangereuses.

Par exemple, sur un système de type Unix, un programme utilisateur peut seulement accéder à ses propres données et il ne peut pas, par exemple, rebooter la machine.

X-A-2 - Comment ?

La famille de processeur i386 offre plusieurs méthodes pour exécuter du code en mode utilisateur. La méthode décrite ici est celle du **software task switching**. Pour commuter une tâche par cette méthode, le noyau doit :

- 1 Définir les segments de mémoire utilisables par la tâche ;
- 2 Copier la tâche (le code exécutable et les données) en mémoire ;
- 3 Créer et initialiser la structure TSS ;
- 4 Empiler certaines valeurs clefs permettant la commutation sur la pile ;
- 5 Commuter grâce à l'instruction `iret`.

Cela peut sembler un peu compliqué, mais pourtant il n'en est rien. Ces différents points sont expliqués en détail ci-dessous.

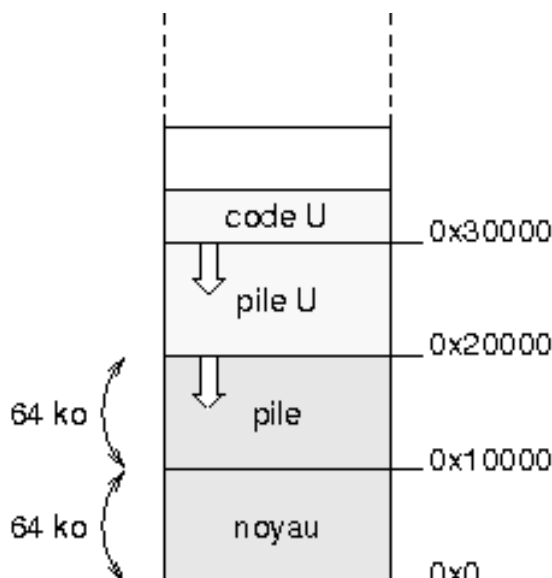
X-B - Définir des segments de mémoire distincts

Dans les chapitres précédents, nous avons vu comment utiliser le mécanisme de segmentation pour adresser l'ensemble de la mémoire en mode protégé. Le mécanisme de segmentation permet de restreindre la mémoire accessible par une tâche utilisateur en lui associant des descripteurs de segment dont les champs **base** et **limite** définissent une plage de mémoire restreinte. La sécurisation est également étendue par l'utilisation du champ **DPL** des descripteurs qui permet d'empêcher l'utilisation de certaines instructions critiques.

Dans notre implémentation, nous avons une seule tâche en mode utilisateur dont l'espace accessible est restreint à :

- un segment de code de 4K ;
- un segment de données de 4K confondu avec le segment de code ;
- un segment de pile de 4K.

Le noyau, quant à lui, peut adresser l'ensemble de la mémoire (cela n'est pas clairement représenté dans le schéma ci-dessous) :



Les descripteurs de segment associés à la tâche utilisateur sont créés et initialisés de la façon suivante :

```
/* descripteur de segments en mode utilisateur */
init_gdt_desc(0x30000, 0x0, 0xFF, 0x0D, &kgdt[4]); /* ucode */
init_gdt_desc(0x30000, 0x0, 0xF3, 0x0D, &kgdt[5]); /* udata */
init_gdt_desc(0x0, 0x20, 0xF7, 0x0D, &kgdt[6]); /* ustack */
```

X-C - Copier le code exécutable en RAM

Pour cette première implémentation, la tâche utilisateur est une fonction très simple qui boucle sur elle-même :

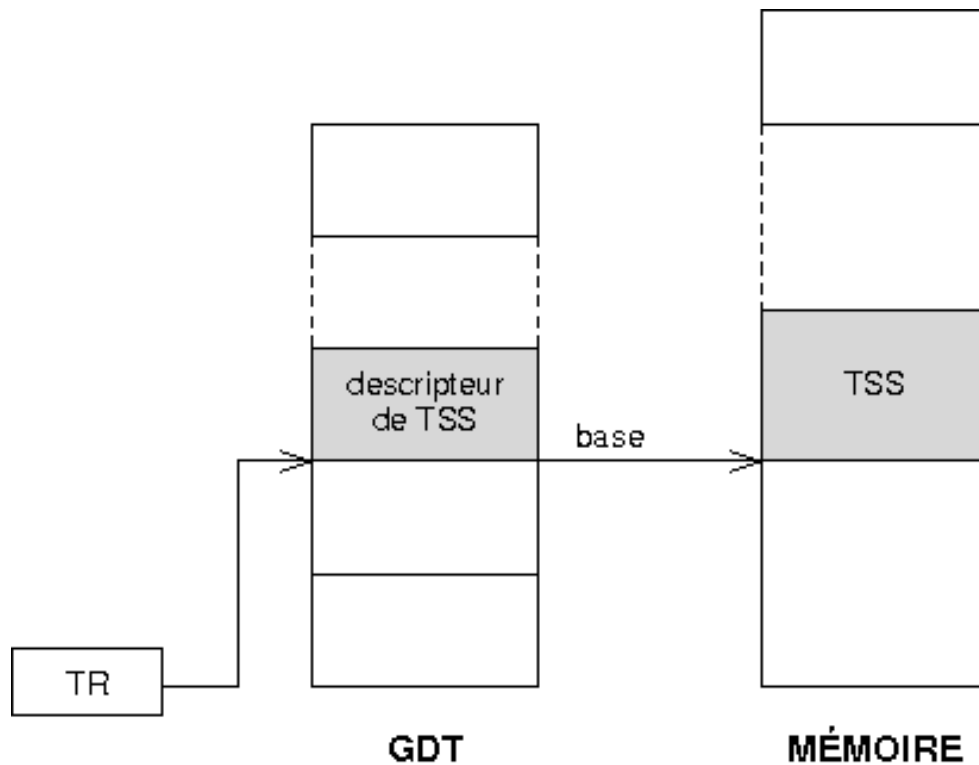
```
void task1(void)
{
    while(1);
    return; /* never go there */
}
```

De façon arbitraire, il a été décidé que le code de la tâche doit résider en 0x30000. Pour le moment, nous n'avons pas vraiment de moyen de créer un exécutable et de le charger en mémoire via un chargeur ELF ou quelque chose de similaire. Nous allons donc tout simplement créer une fonction dans le noyau et copier le code de cette fonction à l'endroit voulu. Comme c'est une toute petite fonction qui occupe au maximum 100 octets, elle est chargée en mémoire par le code suivant :

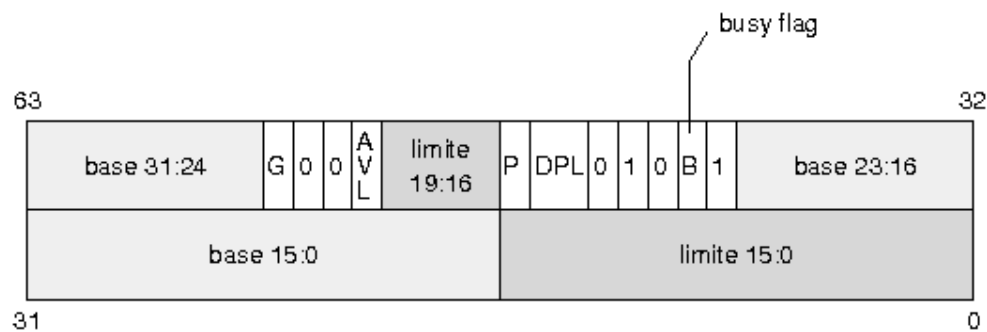
```
/* Copie de la fonction à son adresse */
memcpy((char*) 0x30000, &task1, 100); /* copie de 100 instructions */
```

X-D - Créer et initialiser un TSS

Le **TSS** (Task State Segment) est une structure qui permet de garder en mémoire l'état des registres lors d'une commutation de tâche ou d'une interruption. Dans notre cas, le TSS va uniquement servir à garder en mémoire les indications relatives à la pile du noyau (à savoir les valeurs des registres SS et ESP). Cette structure, utilisée de façon assez diverse selon l'implémentation de la commutation de tâche, peut résider n'importe où en mémoire. Elle est localisable grâce à un descripteur particulier dans la GDT : le **descripteur de TSS**. Un sélecteur de segment spécifique, le **Task Register (TR)**, permet de pointer sur ce descripteur :



X-D-1 - Descripteur de TSS



X-D-2 - Initialiser le TSS

L'initialisation du **TSS** se fait en plusieurs étapes :

1 Création et initialisation de la structure du TSS

```
struct tss {
    u16    previous_task, __previous_task_unused;
    u32    esp0;
    u16    ss0, __ss0_unused;
    u32    esp1;
    u16    ss1, __ss1_unused;
    u32    esp2;
    u16    ss2, __ss2_unused;
    u32    cr3;
    u32    eip, eflags, eax, ecx, edx, ebx, esp, ebp, esi, edi;
    u16    es, __es_unused;
    u16    cs, __cs_unused;
    u16    ss, __ss_unused;
    u16    ds, __ds_unused;
```

```
u16    fs, __fs_unused;
u16    gs, __gs_unused;
u16    ldt_selector, __ldt_sel_unused;
u16    debug_flag, io_map;
} __attribute__((packed));
```

```
struct tss default_tss;

default_tss.debug_flag = 0x00;
default_tss.io_map = 0x00;
default_tss.esp0 = 0x20000;
default_tss.ss0 = 0x18;
```

2 Création du descripteur de TSS

```
init_gdt_desc((u32) &default_tss, 0x67, 0xE9, 0x00, &kgdt[7]); /* descripteur de tss */
```

3 Chargement du registre TR à l'aide de l'instruction ltr

```
asm(" movw $0x38, %ax \n \
      ltr %ax");
```

4 La mise à jour du segment et du pointeur de pile noyau du TSS (champs ss0 et esp0 du TSS) avec les valeurs courantes se fait très simplement :

```
asm(" movw %%ss, %0 \n \
      movl %%esp, %1" : "=m" (default_tss.ss0), "=m" (default_tss.esp0) : );
```

X-E - Exécuter une tâche

Intel(c) propose plusieurs mécanismes pour effectuer une commutation de tâche. La commutation **hardware** est « presque » entièrement gérée par le processeur. Elle est cependant rarement utilisée, car :

- sa mise en œuvre est paradoxalement plutôt difficile ;
- le nombre de processus est limité par la taille de la GDT.

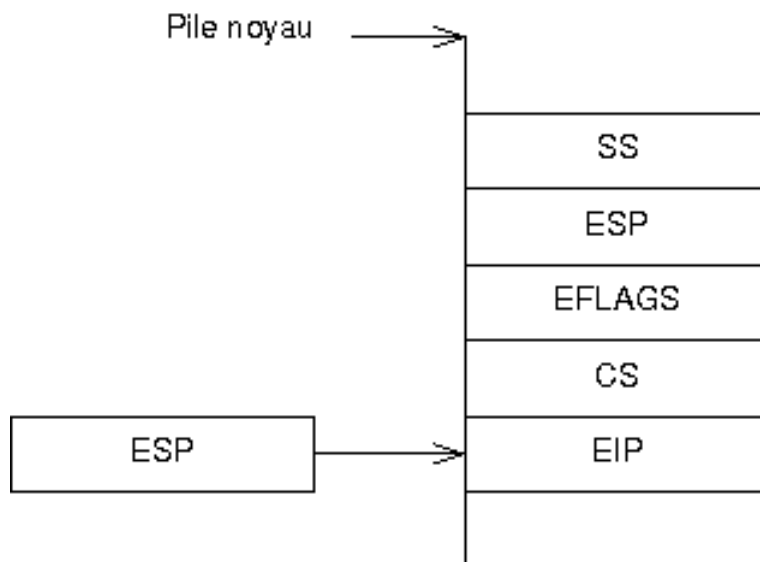
La commutation logicielle, à l'inverse, est comparativement simple à mettre en œuvre et n'a pas de limitation concernant le nombre de processus possible. Pour commuter une tâche de façon logicielle, il suffit de :

- 1 Empiler certaines valeurs clefs sur la pile ;
- 2 Commuter grâce à l'instruction `iret`.

X-E-1 - Simuler un retour d'interruption pour changer de tâche

L'instruction `iret` sert normalement à retourner d'une interruption. Dans le cas présent, une petite astuce à la base de notre implémentation va se servir de `iret` pour donner la main à une tâche non privilégiée.

Quand une tâche utilisateur s'exécute et qu'une interruption survient, le processeur va automatiquement sauvegarder sur la pile noyau différents registres lui permettant de retourner dans le contexte utilisateur une fois le traitement de l'interruption terminé :



Une fois l'interruption traitée, l'instruction `iret` fait les choses suivantes :

- 1 Le processeur dépile les registres EIP et CS ;
- 2 Il vérifie s'il y a un changement de privilège ;
- 3 Il met à jour les registres EIP et CS ;
- 4 Il dépile et met à jour le registre EFLAGS ;
- 5 En cas de changement de privilège, il dépile et met à jour les registres SS et ESP.

Pour commuter en mode non privilégié, il suffit donc de :

- 1 Empiler des registres SS et ESP qui pointent sur la pile utilisateur ;
- 2 Empiler un registre d'état EFLAGS valide ;
- 3 Empiler des registres CS et EIP qui pointent vers le code utilisateur ;
- 4 Exécuter `iret`.

Le code ci-dessous effectue un saut vers la fonction `task1()` en mode utilisateur :

```
asm("    cli \n \
        push $0x33 \n \
        push $0x30000 \n \
        pushfl \n \
        popl %?x \n \
        orl $0x200, %?x \n \
        and $0xffffbfff, %?x \n \
        push %?x \n \
        push $0x23 \n \
        push $0x0 \n \
        movl $0x20000, %0 \n \
        movw $0x2B, %%ax \n \
        movw %%ax, %%ds \n \
        iret" : "=m" (default_tss.esp0) : );
```

X-E-1-a - Le code vu pas à pas

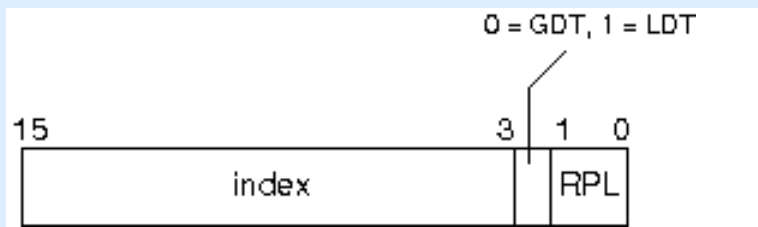
```
asm("    cli \n \
```

On commence par désactiver les interruptions.

```
push $0x33 \n \
push $0x30000 \n \
```

Nous avons vu plus haut que le descripteur de segment de pile est à l'offset 0x30 dans la GDT et qu'il définit un espace mémoire de 4 K débutant à l'adresse 0x20000. Le sommet de la pile est donc en 0x30000. On commence par placer le futur sélecteur et le futur pointeur de pile utilisateur sur la pile noyau.

Lors de l'appel à `iret`, nous voulons que le processeur mette à jour le segment de pile SS avec celui qui est à l'index 0x30 de la GDT (la pile utilisateur). Mais nous voulons aussi que le processeur passe à un niveau d'exécution différent (ici, moins privilégié). On indique ce niveau d'exécution au processeur en l'ajoutant dans le champ RPL (Requested Privilege Level) du sélecteur :



Dans notre exemple, nous voulons passer en mode utilisateur (valeur de privilège 3, on parle aussi de passer en mode ring 3). Il faut donc utiliser le sélecteur $0x30 + 3 = 0x33$ pour utiliser le segment de pile utilisateur en tant qu'utilisateur. Sans cela, le processeur va déclencher une exception de type General Protection fault.

```
pushfl          \n \
popl   %?x      \n \
orl   $0x200,   %?x \n \
and   $0xfffffbfff, %?x \n \
push  %?x       \n \
```

On place le registre EFLAGS sur la pile en ayant pris soin de désactiver le bit **Nested Task (NT)** et en ayant activé le bit **Interrupt Flag (IF)**.

```
push $0x23 \n \
push $0x0  \n \
```

On place sur la pile le sélecteur et le pointeur de code utilisateur. Comme avec le segment de pile, nous voulons que le processeur passe à un niveau d'exécution différent, ici moins privilégié. On indique ce niveau de privilège, ici **3**, au processeur en l'ajoutant à la valeur du sélecteur. Dans notre exemple, il faut donc utiliser le sélecteur $0x20 + 3 = 0x23$ pour utiliser le segment de code utilisateur avec un niveau de privilège de **3**.

À ce stade-là, la pile est prête pour basculer vers la tâche utilisateur.

```
movl $0x20000, %0 \n \
```

Cette instruction met à jour la valeur de `default_tss.esp0`, qui contient la valeur du pointeur de pile noyau utilisé par les interruptions pendant que la tâche utilisateur s'exécute.

```
movw $0x2B, %%ax \n \
movw %%ax, %%ds \n \
```

On initialise le sélecteur de segment de données en le faisant pointer sur le descripteur de la partie utilisateur. On applique la même remarque que vu précédemment avec les segments de code et de pile et on utilise donc le sélecteur $0x28 + 3 = 0x2B$.

```
iret" : "=m" (default_tss.esp0) : );
```

L'instruction `iret` est normalement utilisée pour retourner d'une interruption. Elle dépile l'adresse de retour (CS et EIP), le registre EFLAGS et les informations de pile (SS et ESP). On note que :

- le bit **NT** du registre EFLAGS doit être désactivé pour éviter que le processeur fasse une commutation de tâche **hardware** ;
- le bit **IF** du registre EFLAGS doit être activé pour autoriser les interruptions une fois en mode utilisateur.

X-F - Sauvegarder le contexte d'une tâche

X-F-1 - Des interruptions qui sauvegardent la tâche en cours

Quand le processeur est en mode utilisateur et qu'une interruption est reçue, il doit passer en mode privilégié pour exécuter l'**ISR** appropriée. En cas de changement de privilège, le processeur fait automatiquement les choses suivantes :

- 1 Il récupère dans le TSS les valeurs des champs `ss0` et `esp0` correspondant aux registres `SS` et `ESP` de la pile noyau ;
- 2 Il bascule sur la pile noyau et il empile dessus les registres `SS`, `ESP`, `EFLAGS`, `CS` et `EIP` liés à la tâche utilisateur ;
- 3 Dans certains cas, il empile un code d'erreur ;
- 4 Il exécute l'ISR.

Mais cela n'est pas suffisant. Pour sauvegarder entièrement le contexte d'exécution de la tâche utilisateur, il faut stocker quelque part, par exemple en les empilant sur la pile noyau, les autres registres du processeur susceptibles d'être modifiés par la routine d'interruption.

Pour illustrer ce point, à l'origine notre handler pour l'interruption `IRQ0` était :

```
_asm_irq_0:
    call isr_clock_int
    mov al,0x20
    out 0x20,al
    iret
```

Nous devons le modifier afin qu'il sauvegarde les registres (on remarque aussi qu'il initialise le sélecteur de segment de données `DS` afin que le noyau puisse accéder au segment de données) :

```
%macro SAVE_REGS 0
    pushad
    push ds
    push es
    push fs
    push gs
    push ebx
    mov bx,0x10
    mov ds,bx
    pop ebx
%endmacro

%macro RESTORE_REGS 0
    pop gs
    pop fs
    pop es
    pop ds
    popad
%endmacro

_asm_irq_0:
    SAVE_REGS
    call isr_clock_int
    mov al,0x20
    out 0x20,al
```

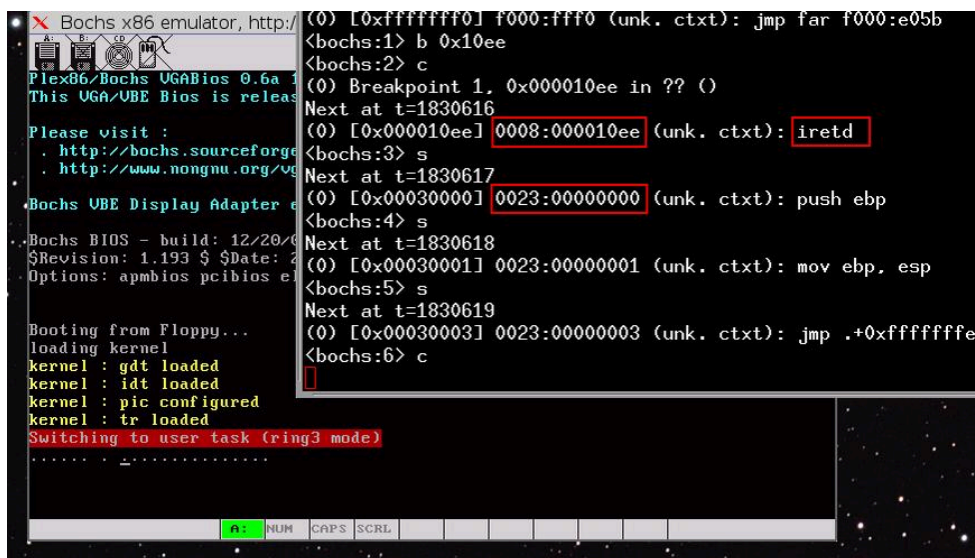
```
RESTORE_REGS
iret
```

X-G - Compiler et exécuter le noyau

La compilation n'apporte aucune surprise :

```
tar xzf kernel_MonoTask.tgz
cd MonoTask
make
```

En revanche, pour tester le nouveau noyau, il faut utiliser Bochs en mode debug. La fenêtre ci-dessous montre ce qui se passe après l'exécution de l'instruction `iret`. L'instruction d'après est en mode utilisateur et adresse le bon segment de code. Un point est affiché toutes les 100 interruptions d'horloge, ce qui prouve que les interruptions fonctionnent correctement malgré le passage en mode utilisateur :



XI - Les appels système : un noyau monotâche qui implémente des appels système

- Des appels système pour accéder aux services du noyau
- Charger une tâche
- Exécuter le noyau

Sources

Le package contenant les sources est téléchargeable ici : [kernel_MonoTask_Syscall.tgz](#)

*En 64 bits, l'appel aux services système diffère. On ne passe pas par des interruptions logicielles, mais par l'instruction **SYSENTER**. Les registres utilisés pour le passage de paramètres (eax pour le numéro d'appel, puis ebx, ecx, edx, esi, edi, ebp en 32 bits, rax pour le numéro d'appel, puis rdi, rsi, rdx, rcx, r8, et r9 en 64 bits).*

XI-A - Des appels système pour accéder aux services du noyau

XI-A-1 - Pourquoi ?

Le mécanisme de segmentation empêche le code utilisateur d'accéder librement aux périphériques ou au noyau. Pour accéder à ces ressources, par exemple pour écrire dans la mémoire vidéo ou sur disque, le code utilisateur utilise des services implémentés au niveau du noyau : les **appels système**. Le noyau illustré ici implémente un appel système permettant d'afficher un message à l'écran.

XI-A-2 - Appeler une routine privilégiée à l'aide d'une interruption logicielle

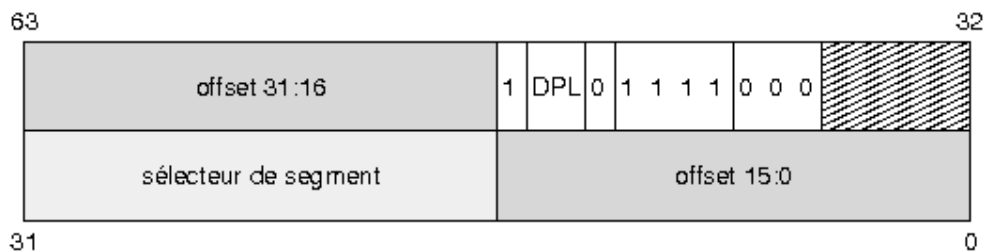
Nous avons vu aux chapitres précédents que des interruptions peuvent être déclenchées par les périphériques ou directement par le processeur en cas d'exception. Un troisième type d'interruption existe, ce sont les **interruptions logicielles**, déclenchées volontairement par le code noyau ou utilisateur à l'aide de l'instruction `int`. Ce sont ces interruptions qui vont nous servir pour implémenter les appels système.

XI-A-3 - Une interruption logicielle qui utilise un Trap Gate

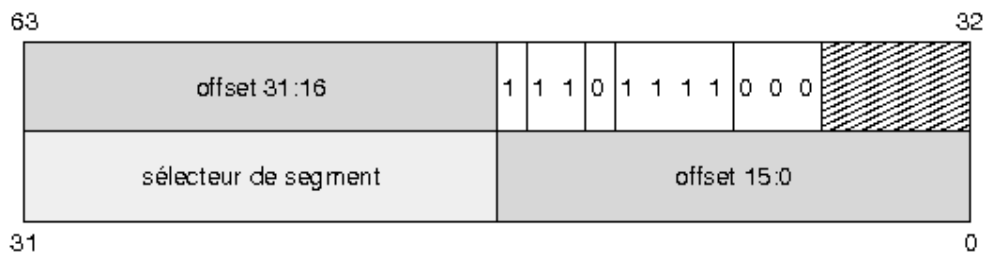
Nous avons déjà détaillé la façon dont fonctionnent les interruptions matérielles. Les interruptions logicielles fonctionnent exactement de la même façon à deux différences près :

- nous allons utiliser un descripteur de **Trap Gate** au lieu d'un descripteur de type Interrupt Gate. La seule différence entre les deux est qu'un **Trap Gate** ne désactive pas les interruptions ;
- nous allons initialiser différemment le champ **DPL** du descripteur.

XI-A-3-a - Descripteur système de type Trap Gate



Le **DPL** (Descriptor Privilege Level) est utilisé pour contrôler l'accès au segment selon le niveau de privilège du code appelant. La valeur 3 indique que toutes les applications, même les moins privilégiées, peuvent utiliser le **trap gate** tandis que la valeur 0 restreint son utilisation au seul code noyau. Comme nous voulons que le **trap gate** soit utilisable par les applications utilisateur, il faut un **DPL** de 3 :



Il est initialisé à l'aide du code suivant :

```
init_idt_desc(0x08, (u32) _asm_syscalls, 0xEF00, &idt[48]); /* appels systeme - int 0x30 */
```

Grâce à ce descripteur, suite à une interruption de type int 0x30, le processeur va exécuter la routine de service `_asm_syscalls`.

XI-A-4 - Comment passer des paramètres à l'appel système ?

Nous avons vu ci-dessus qu'un appel système se fait très simplement grâce aux interruptions logicielles. Le noyau peut ainsi fournir des services aux applications utilisateur, mais comment passer des paramètres à ces services ? Deux solutions sont très courantes :

- passer les paramètres via la pile utilisateur ;
- passer les paramètres en les plaçant dans les registres, la plus simple à implémenter.

Nous allons voir comment implémenter la deuxième solution. Elle a le mérite d'être simple, mais le nombre réduit de registres sur les architectures i386 limite le nombre de paramètres que l'on peut passer par cette méthode (`eax`, `ebx`, `ecx`, `edx`, `edi` et `esi`).

Dans notre implémentation d'un appel système qui affiche une chaîne de caractères à l'écran, les registres utilisés sont :

- `eax`, qui contient le numéro d'appel système ;
- `ebx`, qui contient l'adresse de la chaîne de caractères à afficher.

Le code qui réalise l'appel :

```
asm("mov %0, %%x; mov $0x01, %%x; int $0x30" :: "m" (msg));
```

XI-A-5 - La routine d'interruption (1)

La routine de traitement de l'interruption est semblable à celle déjà vue pour les interruptions matérielles. Elle commence par sauvegarder les registres utilisateur et par initialiser le registre DS pour qu'il pointe sur le segment de données du noyau. Ensuite, elle pousse sur la pile le registre `eax`, qui contient le numéro de l'appel système, pour le transmettre à la fonction `do_syscalls()` appelée juste après :

```
_asm_syscalls:
    SAVE_REGS
    push eax                ; transmission du numéro d'appel
    call do_syscalls
    pop eax
    RESTORE_REGS
    iret
```

XI-A-6 - La routine d'interruption (2)

La fonction `do_syscalls()` effectue le vrai travail de gestion de l'interruption. Le principe de cette fonction est simple :

- déterminer l'appel système grâce au numéro passé en argument à la fonction ;
- récupérer les paramètres de l'appel, placés dans les registres et poussés sur la pile

```
#include "types.h"
#include "gdt.h"
#include "screen.h"

void do_syscalls(int sys_num)
{
    u16 ds_select;
    u32 ds_base;
    struct gdt_desc *ds;
```

```

uchar *message;

if (sys_num == 1) {
    asm("    mov 44(%?p), %?x    \n \
        mov %?x, %0            \n \
        mov 24(%?p), %%ax      \n \
        mov %%ax, %1" : "=m"(message), "=m"(ds_select) : );

    ds = (struct gdt_desc *) (GDTBASE + (ds_select & 0xF8));
    ds_base = ds->base0_15 + (ds->base16_23 << 16) + (ds->base24_31 << 24);

    print((char*) (ds_base + message));
} else {
    print("syscall\n");
}

return;
}

```

XI-A-6-a - Que fait exactement cette fonction ?

```
void do_syscalls(int sys_num)
```

La ligne ci-dessus permet de récupérer le numéro d'appel système passé en paramètre à la fonction (par le biais de l'instruction `push eax`). Dans notre implémentation, il n'y a pour le moment qu'un appel système, mais un noyau en a généralement plusieurs dizaines ou centaines.

```
if (sys_num == 1) {
```

Le bloc traite l'appel système numéro 1.

```

asm("    mov 44(%?p), %?x    \n \
    mov %?x, %0            \n \
    mov 24(%?p), %%ax      \n \
    mov %%ax, %1" : "=m" (message), "=m" (ds_select) : );

```

Dans notre implémentation, les paramètres sont passés à l'appel système à l'aide des registres. Nous savons que l'adresse de la chaîne à afficher a été placée par le programme utilisateur dans le registre `EBX`. Mais à cet endroit de la fonction `do_syscalls`, rien ne nous garantit que ce registre n'a pas été modifié. Heureusement, ce registre a été sauvegardé sur la pile noyau, ce qui permet de récupérer sa valeur.

Une fois récupérée la valeur de `EBX`, on devrait pouvoir afficher la chaîne sans problème ? Eh bien non !

La chaîne de caractères contenant le message à afficher n'est toujours pas accessible, car `EBX` contient seulement un déplacement et il faut récupérer l'adresse de la base du segment de données utilisateur pour connaître l'adresse physique exacte où se situe la chaîne. On obtient cette adresse à partir de la valeur sauvegardée du registre `DS` qui pointe sur le bon descripteur de segment dans la GDT.

Note : l'annexe sur les **Stack Frame** explique dans le détail comment récupérer des paramètres sur la pile.

```

ds = (struct gdt_desc*) (GDTBASE + (ds_select & 0xF8));
ds_base = ds->base0_15 + (ds->base16_23 << 16) + (ds->base24_31 << 24);

```

À partir de la valeur `ds_select`, on retrouve l'emplacement du descripteur de segment de données dans la **GDT**. La valeur du registre `DS` n'est pas utilisée telle quelle, car le champ **RPL** a été positionné. Pour obtenir l'offset du descripteur dans la GDT, il faut appliquer le masque `0xF8`. L'adresse de base est ensuite reconstituée à partir de ses différents champs.

```
print(ds_base + message);
```

Enfin, la fonction `print()` est appelée avec l'adresse physique de la chaîne à afficher.

```

} else {
    print("syscall\n");
}
return;

```

Pour les autres appels système, on affiche seulement un petit message à l'écran et la fonction se termine.

XI-B - Charger une tâche

Le code applicatif est assez simple et affiche simplement un message avant de boucler indéfiniment.

Le modèle d'organisation de la mémoire utilisé est décrit au chapitre précédent. Il spécifie que le code applicatif, défini par la fonction `task1()`, doit être en `0x30000`. Comme ce code fait au maximum 100 octets, le noyau le recopie directement à l'adresse voulue :

```

/* copie de la fonction a son adresse */
memcpy((char*) 0x30000, (char*) &task1, 100); /* copie de 100 instructions */

```

Les données utilisateur, ici le message à afficher, doivent être stockées dans le segment entre `0x30000` et `0x31000`. En principe, nous aimerions avoir une fonction utilisateur ressemblant à ceci :

```

void task1(void)
{
    char *msg = "hello world !\n";
}

```

Hélas, à ce stade du développement de notre noyau, ça n'est pas possible, car la fonction `task1()` fait partie du noyau et n'a pas été compilée pour tenir compte du fait qu'elle serait relogée ailleurs. Nous sommes donc obligés de ruser un peu pour stocker la chaîne de caractères quelque part entre `0x30000` et `0x31000`. Arbitrairement (enfin pas tout à fait, nous avons pris soin de prendre une valeur qui n'écrase pas le code de la fonction), la chaîne est placée en `0x30100` :

```

void task1(void)
{
    char *msg = (char*) 0x100; /* le message sera stocké en 0x30100 */
    msg[0] = 't';
    msg[1] = 'a';
    msg[2] = 's';
    msg[3] = 'k';
    msg[4] = 'l';
    msg[5] = '\n';
    msg[6] = 0;

    asm("mov %0, %%x; mov $0x01, %%x; int $0x30" :: "m" (msg));
    while(1);
    return; /* never goes there */
}

```

Une fois la chaîne placée en `0x30100`, le code peut exécuter l'appel système permettant de l'afficher à l'écran :

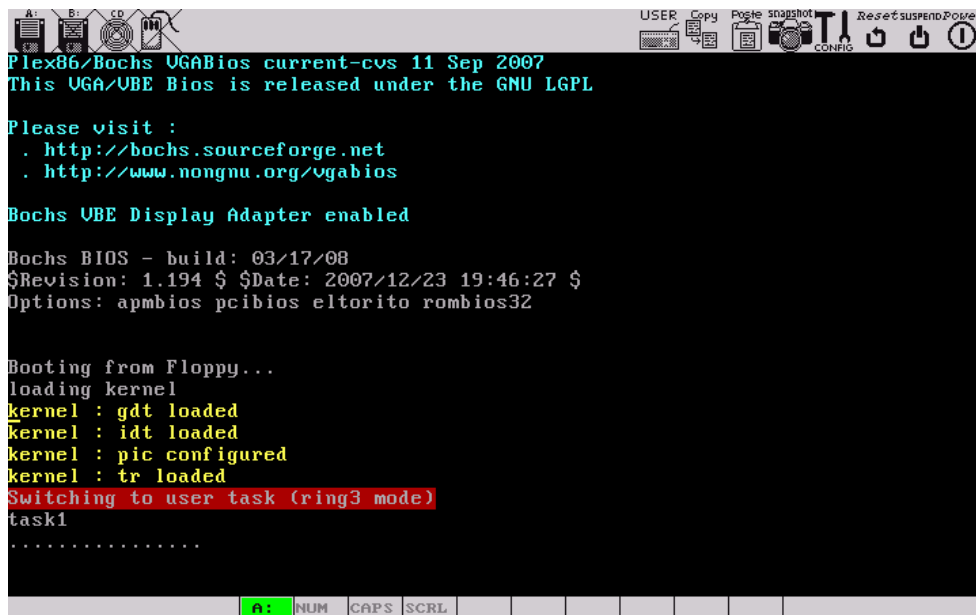
```

asm("mov %0, %%x; mov $0x01, %%x; int $0x30" :: "m" (msg));

```

XI-C - Exécuter le noyau

La tâche en mode utilisateur fait un appel système au service d'affichage, en mode noyau, qui écrit dans la mémoire vidéo :



XII - Gérer la mémoire - un noyau qui implémente la pagination

- **Mémoire virtuelle et mémoire physique**
- **Une première implémentation très simple**
- **Exécuter le noyau**

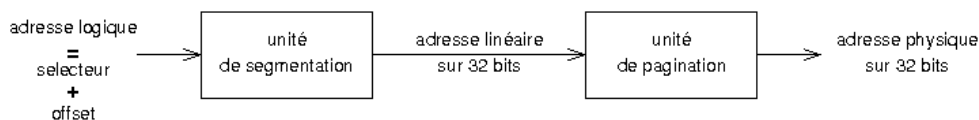
Sources

Le package contenant les sources est téléchargeable ici : [kernel_PagingEnable.tgz](#)

XII-A - Mémoire virtuelle et mémoire physique

XII-A-1 - 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** :



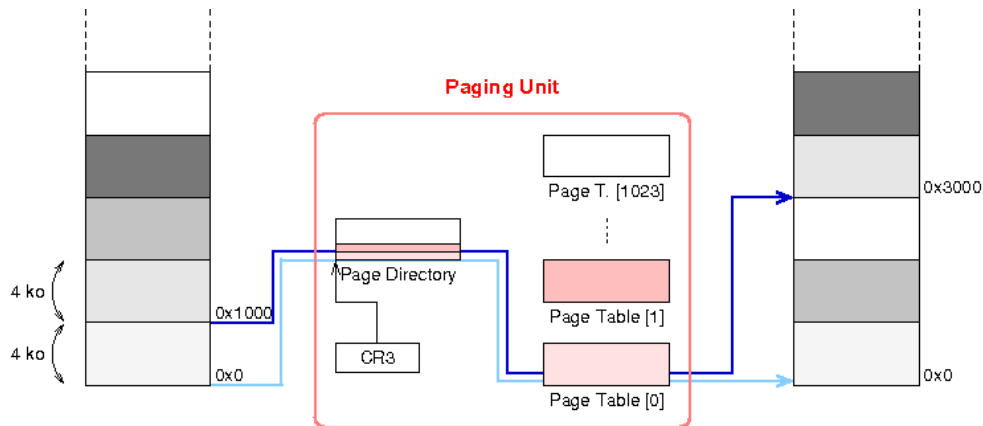
La pagination est un mécanisme d'adressage de la mémoire qui permet :

- 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.

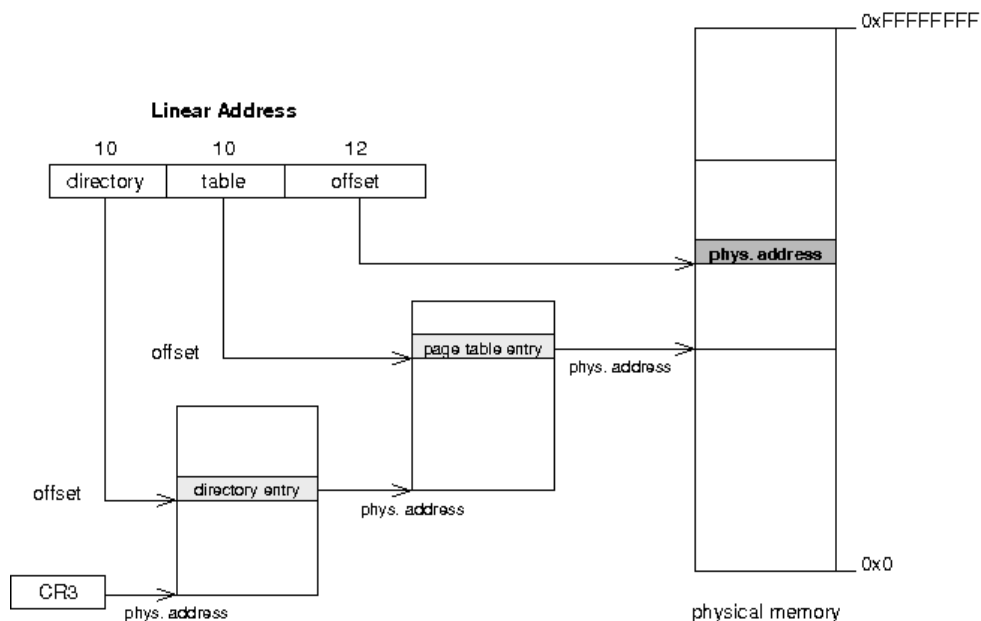
Le schéma ci-dessous illustre l'association entre des pages en mémoire linéaire (ou virtuelle) et en mémoire physique par l'unité de pagination de la **MMU (Memory Management Unit)** :



XII-A-2 - Le fonctionnement

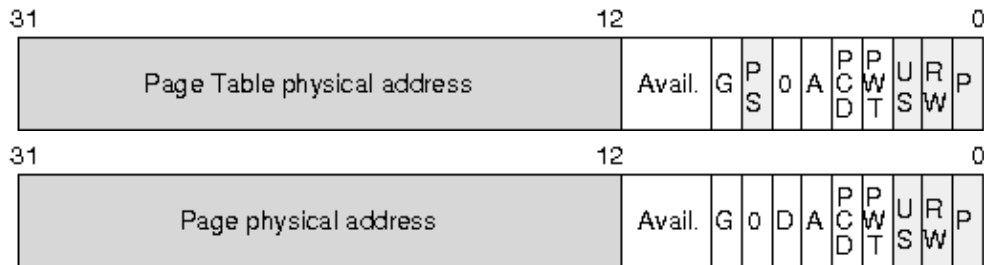
La traduction d'une adresse linéaire en une adresse physique se fait en plusieurs étapes :

- 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 pages pointe sur une page de 4 ko ;
- 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.



XII-A-3 - 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 4 ko et à 1 pour 4 Mo) ;
- 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 4 ko, 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 avons déjà vu ce type de mécanisme avec les sélecteurs de segment de la GDT.

XII-A-3-a - Un peu d'arithmétique

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



En 64 bits, un niveau supplémentaire est ajouté, nommé **PLM4**

XII-A-4 - 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 sont modifiés, 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.

XII-A-5 - Activer la pagination

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

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

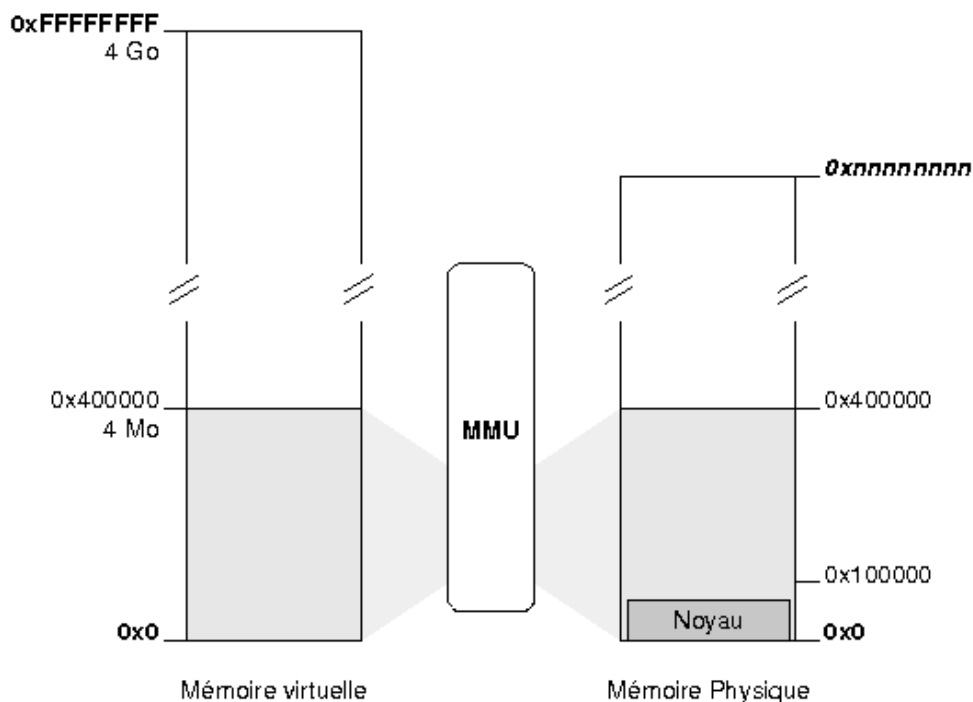
Bien entendu, pour que cela fonctionne, il faut au préalable avoir initialisé un répertoire et au moins une table des pages !

XII-B - Une première implémentation très simple

Pour une première implémentation, nous n'allons pas créer de tâche utilisateur. La pagination s'appliquera donc uniquement au noyau.

XII-B-1 - Une organisation simple de la mémoire avec l'identity mapping

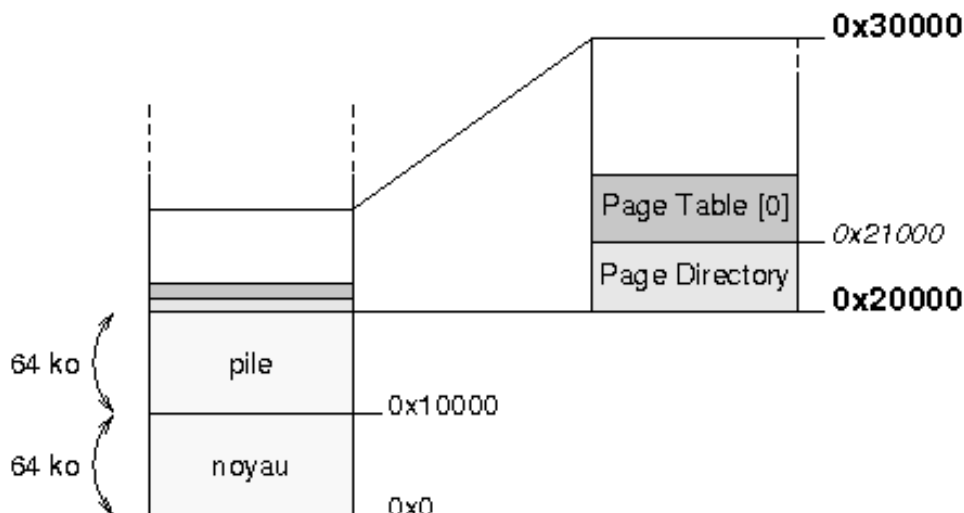
Dans cette première implémentation, nous allons activer la pagination pour le noyau tel que les quatre premiers Mo de mémoire virtuelle coïncident avec les 4 premiers Mo de mémoire physique :



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 précédents que notre noyau est tout petit et qu'il loge dans moins de 64 ko, nous allons adopter l'organisation suivante :

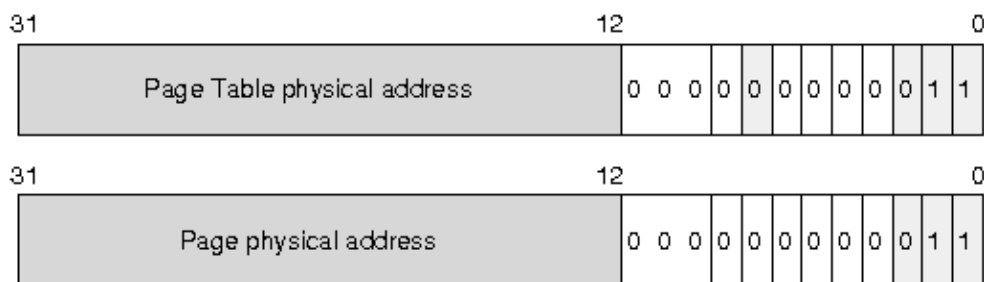
- le code et les données du noyau occupent l'espace entre l'adresse 0x0 et 0x10000 ;
- la pile du noyau utilise l'espace entre l'adresse 0x10000 et 0x20000 ;
- le répertoire de pages sera chargé à l'adresse 0x20000 ;
- pour adresser 4 Mo, une seule table de pages suffit. Elle sera en 0x21000.



XII-B-2 - Initialiser le répertoire et les tables de pages

Les entrées du répertoire et de la table seront initialisées 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 4 ko.



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 */

/* créé 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;

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

```

/* Création 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, %%x    \n \
      mov %%x, %%cr3 \n \
      mov %%cr0, %%x \n \
      or %1, %%x    \n \
      mov %%x, %%cr0" :: "i"(PT0_ADDR), "i"(PAGING_FLAG));
}

```

XII-B-2-a - Que fait exactement cette fonction ?

```
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 autres ? Tout simplement parce que l'un des points clefs 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 cela aurait généré un décalage entre les adresses en mémoire virtuelle et celles en 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 pages, ce qui suffit pour adresser 4 Mo.

```

/* Création 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 pages 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 autres pages se succèdent.

XII-B-3 - Une nouvelle exception pour gérer les Page Fault

```
init_idt_desc(0x08, (u32) _asm_exc_PF, INTGATE, &idt[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**. À 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.

Dans notre cas, le noyau va simplement afficher un message d'erreur et stopper.

XII-C - Exécuter le noyau

L'affichage du message et des petits points montre que malgré le passage en mode paginé, le noyau fonctionne normalement. L'annexe sur le **mode debug de Bochs** indique quelques commandes utiles pour vérifier la bonne prise en compte du répertoire et des tables de pages.

```

Flex86/Bochs UGABios 0.6a 19 Aug 2006
This UGA/UBE Bios is released under the GNU LGPL

Please visit :
. http://bochs.sourceforge.net
. http://www.nongnu.org/ugabios

Bochs UBE Display Adapter enabled

Bochs BIOS - build: 12/20/07
$Revision: 1.193 $ $Date: 2007/12/20 18:12:11 $
Options: apmbios pcibios eltorito rombios32

Booting from Floppy...
loading kernel
kernel : gdt loaded
kernel : idt loaded
kernel : pic configured
kernel : tr loaded
kernel : paging enable
kernel : interrupts enable
.....
A: NUM CAPS SCRL

```

XIII - Gérer la mémoire - utiliser la pagination pour une tâche utilisateur

- **Préalable : gérer la mémoire physique**
- **Comment organiser l'espace d'adressage d'une tâche utilisateur**
- **Comment organiser l'espace noyau, pour gérer les appels système**
- **Créer une tâche**
- **Exécuter la tâche utilisateur**

Sources

Le package contenant les sources est téléchargeable ici : **kernel_PagingUserEnable.tgz**

XIII-A - Préalable : gérer la mémoire physique

Pour créer une tâche utilisateur, nous avons la vague idée que le noyau doit mettre à jour le répertoire et les tables de pages, qu'il doit créer une pile, et qu'il doit aussi réserver de la mémoire pour y copier le code et les données. Cela suffit-il ? Non ! Car avant toute chose, le noyau doit savoir quelles pages en mémoire physique sont libres et lesquelles sont utilisées. Il faut donc un système de gestion des pages de la mémoire physique.

Le système de gestion des pages mis en œuvre ici repose sur l'utilisation d'un **bitmap**. Un bitmap est un tableau dont chaque élément est un bit. Dans notre cas, chaque bit correspond à une page en mémoire physique : le premier bit correspond à la première page, le deuxième bit correspond à la deuxième page, etc. Chaque bit renseigne sur le statut libre ou occupé de la page associée :

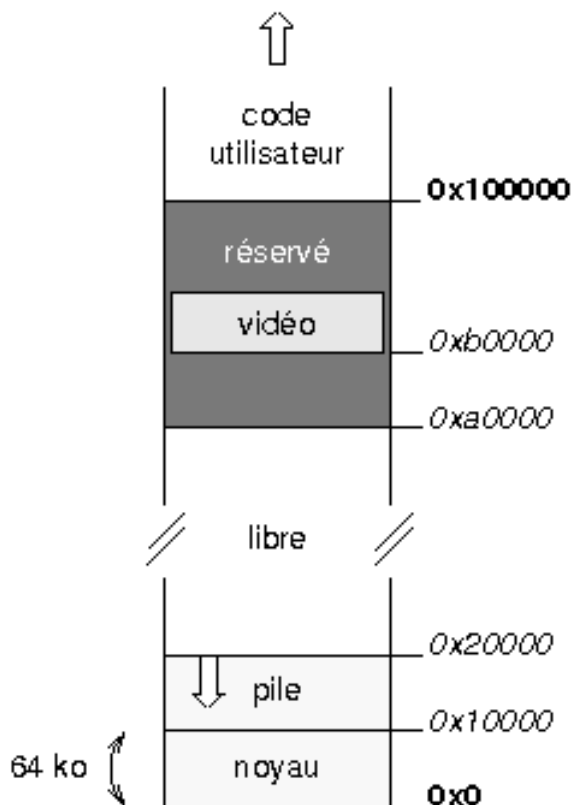
```
u8 mem_bitmap[RAM_MAXPAGE / 8]; /* bitmap allocation de pages */
```



Nous utilisons ici un bitmap pour gérer l'utilisation des pages physiques (page frame), mais d'autres solutions sont possibles comme l'utilisation d'une pile d'adresses libres ou bien d'une liste chaînée de structures décrivant les pages allouables.

XIII-A-1 - Initialisation du bitmap

La première étape du gestionnaire est d'initialiser le bitmap de façon à marquer les pages déjà occupées/réservées. L'espace physique sera organisé de la façon suivante :



On remarque que la tâche utilisateur sera chargée physiquement à l'adresse 0x100000. C'est un choix arbitraire et très simplifié qui nous servira à mettre au point la pagination dans le contexte d'une tâche utilisateur. Nous verrons dans les chapitres suivants comment mettre au point un vrai gestionnaire de mémoire.

Les pages utilisées par le noyau ainsi que celles utilisées par le hardware doivent être marquées comme étant prises :

```
/* Initialisation du bitmap de pages physiques */
for (pg = 0; pg < RAM_MAXPAGE / 8; pg++)
    mem_bitmap[pg] = 0;

/* Pages réservées pour le noyau */
for (pg = PAGE(0x0); pg < PAGE(0x20000); pg++)
    set_page_frame_used(pg);
```



```
/* Pages réservées pour le hardware */
for (pg = PAGE(0xA0000); pg < PAGE(0x100000); pg++)
    set_page_frame_used(pg);
```

Ce bout de code fait appel à deux macros :

- `PAGE(addr)` calcule, pour une adresse donnée, le numéro de page physique à laquelle elle appartient ;
- `set_page_frame_used(page)` met à jour le bitmap pour indiquer qu'une page est utilisée.

Mais il existe également d'autres macros et fonctions indispensables à la gestion des pages physiques :

- réserver une page libre : `get_page_frame()` ;
- désallouer une page occupée pour la rendre libre : `release_page_frame()` ;

Toutes ces fonctions sont dans le fichier `mm.c` et dans le fichier `mm.h`

`mm.c` :

```
#include "types.h"

#define __MM
#include "mm.h"

/*
 * Parcourt le bitmap à la recherche d'une page libre et la marque
 * comme utilisée avant de retourner son adresse physique.
 */
char* get_page_frame(void)
{
    int byte, bit;
    int page = -1;

    for (byte = 0; byte < RAM_MAXPAGE / 8; byte++)
        if (mem_bitmap[byte] != 0xFF)
            for (bit = 0; bit < 8; bit++)
                if (!(mem_bitmap[byte] & (1 << bit))) {
                    page = 8 * byte + bit;
                    set_page_frame_used(page);
                    return (char *) (page * PAGESIZE);
                }

    return (char *) -1;
}

/* Créé un mapping tel que vaddr = paddr sur 4Mo */
void init_mm(void)
{
    u32 page_addr;
    int i, pg;

    /* Initialisation du bitmap de pages physiques */
    for (pg = 0; pg < RAM_MAXPAGE / 8; pg++)
        mem_bitmap[pg] = 0;

    /* Pages réservées pour le noyau */
    for (pg = PAGE(0x0); pg < PAGE(0x20000); pg++)
        set_page_frame_used(pg);

    /* Pages réservées pour le hardware */
    for (pg = PAGE(0xA0000); pg < PAGE(0x100000); pg++)
        set_page_frame_used(pg);

    /* Prend une page pour le Page Directory et une pour la Page Table[0] */
    pd0 = (u32*) get_page_frame();
```

```

    pt0 = (u32*) get_page_frame();

    /* Initialisation du Page Directory */
    pd0[0] = (u32) pt0;
    pd0[0] |= 3;
    for (i = 1; i < 1024; i++)
        pd0[i] = 0;

    /* Initialisation de la Page Table[0] */
    page_addr = 0;
    for (pg = 0; pg < 1024; pg++) {
        pt0[pg] = page_addr;
        pt0[pg] |= 3;
        page_addr += 4096;
    }

    asm("    mov %0, %%x \n \
           mov %%x, %%cr3 \n \
           mov %%cr0, %%x \n \
           or %1, %%x \n \
           mov %%x, %%cr0"::"m"(pd0), "i"(PAGING_FLAG));
}

/* Crée un répertoire de pages pour une tâche */
u32 *pd_create_task1(void)
{
    u32 *pd, *pt;
    u32 i;

    /* Prend et initialise une page pour le Page Directory */
    pd = (u32*) get_page_frame();
    for (i = 0; i < 1024; i++)
        pd[i] = 0;

    /* Prend et initialise une page pour la Page Table[0] */
    pt = (u32*) get_page_frame();
    for (i = 0; i < 1024; i++)
        pt[i] = 0;

    /* Espace kernel */
    pd[0] = pd0[0];
    pd[0] |= 3;

    /* Espace u */
    pd[USER_OFFSET >> 22] = (u32) pt;
    pd[USER_OFFSET >> 22] |= 7;

    pt[0] = 0x100000;
    pt[0] |= 7;

    return pd;
}

```

mm.h :

```

#include "types.h"

#define PAGESIZE      4096
#define RAM_MAXPAGE   0x10000

#define VADDR_PD_OFFSET(addr) ((addr) & 0xFFC00000) >> 22
#define VADDR_PT_OFFSET(addr) ((addr) & 0x003FF000) >> 12
#define VADDR_PG_OFFSET(addr) (addr) & 0x00000FFF
#define PAGE(addr)         (addr) >> 12

#define PAGING_FLAG 0x80000000 /* CR0 - bit 31 */
#define USER_OFFSET 0x40000000
#define USER_STACK  0xE0000000

#ifdef __MM__
u32 *pd0; /* kernel page directory */

```

```

u32 *pt0; /* kernel page table */
u8 mem_bitmap[RAM_MAXPAGE / 8]; /* bitmap allocation de pages (1 Go) */
#endif

struct pd_entry {
    u32 present:1;
    u32 writable:1;
    u32 user:1;
    u32 pwt:1;
    u32 pcd:1;
    u32 accessed:1;
    u32 _unused:1;
    u32 page_size:1;
    u32 global:1;
    u32 avail:3;

    u32 page_table_base:20;
} __attribute__((packed));

struct pt_entry {
    u32 present:1;
    u32 writable:1;
    u32 user:1;
    u32 pwt:1;
    u32 pcd:1;
    u32 accessed:1;
    u32 dirty:1;
    u32 pat:1;
    u32 global:1;
    u32 avail:3;

    u32 page_base:20;
} __attribute__((packed));

/* Marque une page comme utilisée/libre dans le bitmap */
#define set_page_frame_used(page) mem_bitmap[((u32) page)/8] |= (1 << (((u32) page)%8))
#define release_page_frame(p_addr) mem_bitmap[((u32) p_addr/PAGESIZE)/8] &= ~(1 << (((u32) p_addr/PAGESIZE)%8))

/* Sélectionne une page libre dans le bitmap */
char *get_page_frame(void);

/* Initialise les structures de données de gestion de la mémoire */
void init_mm(void);

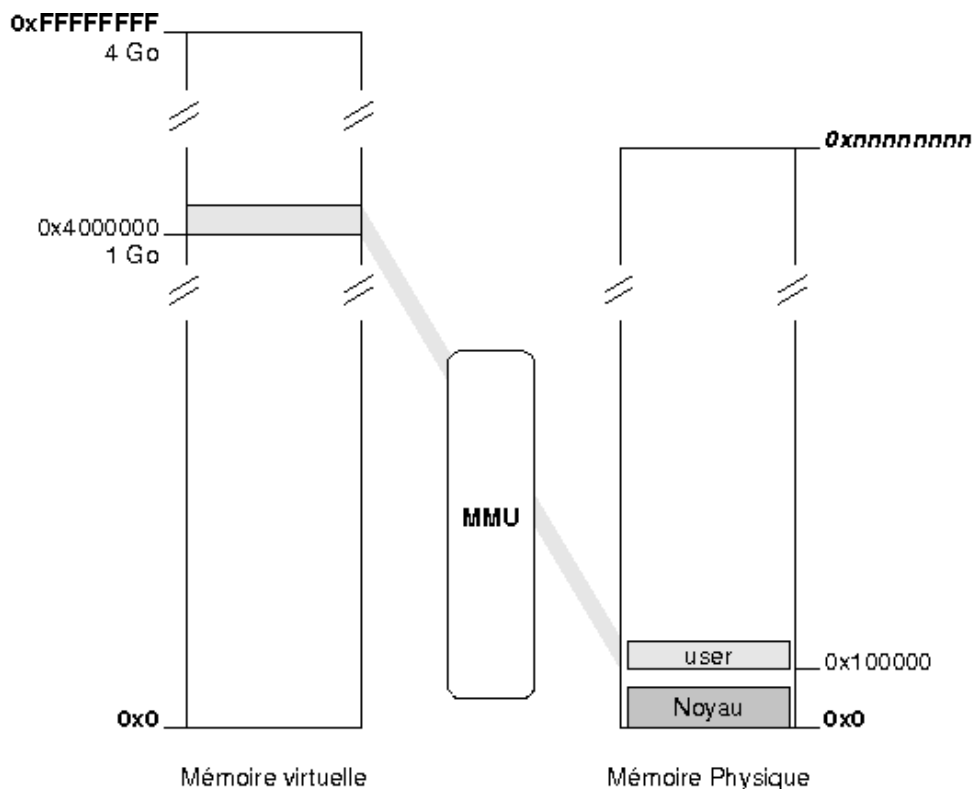
/* Crée un répertoire de pages pour une tâche */
u32 *pd_create_task1(void);

```

XIII-B - Comment organiser l'espace d'adressage d'une tâche utilisateur

Lors de la compilation d'une tâche utilisateur, il n'est pas possible de savoir à l'avance où elle résidera en mémoire physique. Or, à partir du moment où la tâche va manipuler des adresses, cette information est essentielle. Comment faire ? Une solution qui se base sur la pagination va être exposée en détail ici. Son principe est très simple...

Tout d'abord, la tâche est liée de façon à toujours s'exécuter à la même adresse. Le choix de cette adresse est arbitraire et dans notre cas, les tâches doivent être liées en tenant compte qu'elles seront chargées à l'adresse 0x40000000. L'adresse 0x40000000 est une adresse virtuelle, mais ça, le compilateur n'en sait rien ! La tâche pourra être chargée à n'importe quel endroit de la mémoire physique. C'est le noyau qui, grâce au répertoire et aux tables de pages propres à cette tâche, se chargera de faire la correspondance entre l'espace d'adressage virtuel et la mémoire physique. Par exemple, si la tâche est chargée physiquement en 0x100000, le noyau (via le MMU) se chargera de faire la correspondance avec l'espace virtuel débutant en 0x40000000. La tâche aura donc toujours l'impression de s'exécuter réellement à cette adresse :



XIII-C - Comment organiser l'espace noyau, pour gérer les appels système

L'utilisation de la pagination n'introduit pas de changements sur la façon dont sont implémentés les **appels système**. Ceux-ci sont toujours implémentés en utilisant une interruption :

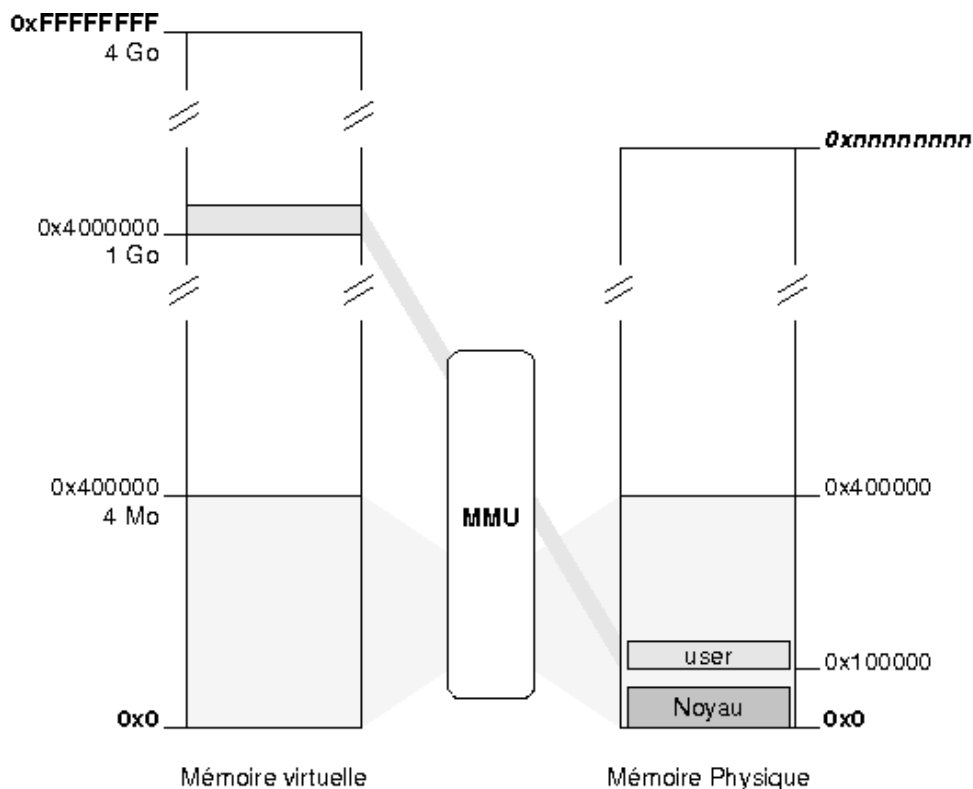
```
init_idt_desc(0x08, (u32) _asm_syscalls, TRAPGATE, &kidt[48]); /* appels systeme - int 0x30 */
```

La seule différence par rapport au modèle segmenté concerne la façon d'adresser les données du noyau et de l'utilisateur. Mais nous verrons ci-dessous que cela va dans le sens d'une simplification !

Supposons par exemple que l'appel système doive écrire à l'écran une chaîne de caractères située en 0x40000100. Au moment de l'appel système, la tâche fournit en paramètre l'adresse de la chaîne en la plaçant dans le registre ebx. Au moment de l'appel système, le noyau doit pouvoir :

- accéder aux données de cette tâche (dans notre exemple, code et données débutent en 0x40000000) ;
- accéder aux données et aux fonctions du noyau, par exemple la fonction printk et la mémoire vidéo en 0xB8000.

Pour pouvoir accéder aux données utilisateur, c'est très simple : il suffit que le noyau utilise le répertoire de pages de la tâche en question. Mais comment faire pour accéder au code et aux données du noyau ? La solution est très simple : pour pouvoir accéder aux données du noyau, il faut que ce répertoire de pages ait aussi les entrées adéquates pour accéder à l'espace virtuel du noyau. Autrement dit, le répertoire de pages de la tâche utilisateur comprend une partie qui lui permet d'accéder à son propre espace d'adressage, et une autre partie qui lui permet d'accéder à l'espace d'adressage du noyau ! :



Le traitement de l'appel système servant à afficher une chaîne est maintenant très simple, car l'adresse passée en paramètre est directement accessible :

```
#include "types.h"
#include "lib.h"

void do_syscalls(int sys_num)
{
    char *u_str;

    if (sys_num == 1) {
        asm("mov %?x, %0": "=m"(u_str) :);
        printk(u_str);
    } else {
        printk("unknown syscall %d\n", sys_num);
    }

    return;
}
```

XIII-C-1 - Note concernant la sécurité

Bien que présent dans son espace d'adressage virtuel, la tâche ne doit pas pouvoir accéder à l'espace du noyau à tout moment : cela constituerait un gros problème de sécurité. Pour empêcher la tâche en mode utilisateur d'accéder à l'espace noyau hors d'un appel système, il suffit de positionner le bit **US** des entrées du répertoire et des tables de pages à :

- 0 (espace privilégié) pour les pages du noyau ;
- 1 (espace non privilégié) pour les pages de l'espace utilisateur.

Nous voulons que la tâche accède au noyau uniquement via des services précis, en l'occurrence des appels système. Le processeur, jusque là en mode utilisateur (ring 3), passe alors en mode noyau (ring 0). Pour distinguer ces deux états, on dit que la tâche s'exécute en mode utilisateur ou en mode noyau.

XIII-D - Créer une tâche

XIII-D-1 - Une tâche très simple

Pour servir d'exemple, la tâche à exécuter est une fonction très simple, presque identique à celle utilisée au chapitre traitant des **appels système**. La seule différence concerne l'adressage :

- selon notre organisation de la mémoire, la tâche va s'exécuter à l'adresse virtuelle 0x40000000 ;
- dans notre exemple, la tâche sera chargée physiquement à l'adresse 0x100000 ;
- cela signifie que le bloc de mémoire virtuelle commençant en 0x40000000 sera mappé sur le bloc de mémoire physique en 0x100000 grâce au répertoire et aux tables de pages.

Le code ci-dessus recopie les caractères à cet emplacement puis fait un appel système pour afficher la chaîne :

```
void task1(void)
{
    char *msg = (char*) 0x40000100; /* le message sera stocké en 0x100100 */
    msg[0] = 't';
    msg[1] = 'a';
    msg[2] = 's';
    msg[3] = 'k';
    msg[4] = 'l';
    msg[5] = '\n';
    msg[6] = 0;

    asm("mov %0, %%x; mov $0x01, %%x; int $0x30" :: "m" (msg));
    while(1);
    return; /* never go there */
}
```

XIII-D-2 - Charger la tâche

Dans le modèle d'organisation de la mémoire, nous avons décidé que la tâche serait en 0x100000 en mémoire physique. Cette fonction occupe au maximum 100 octets, la fonction ci-dessous effectue la copie du code à la bonne adresse :

```
memcpy((u32*) 0x100000, &task1, 100); /* copie de 100 instructions */
```

XIII-D-3 - Créer et initialiser le répertoire et les tables de pages de la tâche utilisateur

L'utilisation de la pagination repose sur le mode protégé qui implique l'utilisation de la segmentation. Il faut donc créer des descripteurs de segments pour le code, les données et la pile utilisateur. Pour simplifier le modèle, ces segments parcourent toute la mémoire :

```
init_gdt_desc(0x0, 0xFFFFF, 0xFF, 0x0D, &kgdt[4]); /* ucode */
init_gdt_desc(0x0, 0xFFFFF, 0xF3, 0x0D, &kgdt[5]); /* udata */
init_gdt_desc(0x0, 0x0, 0xF7, 0x0D, &kgdt[6]); /* ustack */
```

Il faut également créer et initialiser le répertoire et les tables de pages propres à la tâche utilisateur. C'est le rôle de la fonction `pd_create_task1()`. Elle utilise les fonctions d'allocation et de désallocation de pages de la mémoire physique pour y stocker les répertoires et les tables de pages :

```
u32 *pd_create_task1(void)
{
    u32 *pd, *pt;
    u32 i;

    /* Prend et initialise une page pour le Page Directory */
```

```

pd = (u32*) get_page_frame();
for (i = 0; i < 1024; i++)
    pd[i] = 0;

/* Prend et initialise une page pour la Page Table[0] */
pt = (u32*) get_page_frame();
for (i = 0; i < 1024; i++)
    pt[i] = 0;

/* Espace kernel */
pd[0] = pd0[0];
pd[0] |= 3;

/* Espace u */
pd[USER_OFFSET >> 22] = (u32) pt;
pd[USER_OFFSET >> 22] |= 7;

pt[0] = 0x100000;
pt[0] |= 7;

return pd;
}

```



Cette implémentation est une simplification qui vise à illustrer la pagination, mais il faut garder à l'esprit que le modèle mémoire utilisé ici est très rudimentaire. Nous verrons plus tard que dans un OS plus complet, qui offre une gestion plus fine de la mémoire, la gestion des répertoires et des tables de pages obéit à des mécanismes plus complexes.

XIII-E - Exécuter la tâche utilisateur

Le code principal du noyau dans le fichier `kernel.c`.

```

#include "types.h"
#include "lib.h"
#include "gdt.h"
#include "screen.h"
#include "io.h"
#include "idt.h"
#include "mm.h"

void init_pic(void);
int main(void);

void _start(void)
{
    kY = 16;
    kattr = 0x0E;

    /* Initialisation de la GDT et des segments */
    init_gdt();

    /* Initialisation du pointeur de pile %esp */
    asm("    movw $0x18, %ax \n \
        movw %ax, %ss \n \
        movl $0x20000, %esp");

    main();
}

void task1(void)
{
    char *msg = (char *) 0x40000100; /* le message sera stocké en 0x100100 */
    msg[0] = 't';
    msg[1] = 'a';
    msg[2] = 's';
    msg[3] = 'k';
}

```

```

msg[4] = '1';
msg[5] = '\n';
msg[6] = 0;

asm("mov %0, %%x; mov $0x01, %%x; int $0x30::\"m\"(msg)");

while (1);
return;          /* never go there */
}

int main(void)
{
    u32 *pd;

    printk("kernel : gdt loaded\n");

    init_idt();
    printk("kernel : idt loaded\n");

    init_pic();
    printk("kernel : pic configured\n");

    hide_cursor();

    /* Initialisation du TSS */
    asm("    movw $0x38, %ax \n \
        ltr %ax");
    printk("kernel : tr loaded\n");

    init_mm();
    printk("kernel : paging enable\n");

    pd = pd_create_task1();
    memcpy((char *) 0x100000, (char *) &task1, 100);          /* copie de 100 instructions */
    printk("kernel : task created\n");

    kattr = 0x47;
    printk("kernel : trying switch to user task...\n");
    kattr = 0x07;
    asm ("    cli \n \
        movl $0x20000, %0 \n \
        movl %1, %%x \n \
        movl %%x, %%cr3 \n \
        push $0x33 \n \
        push $0x40000F00 \n \
        pushfl \n \
        popl %%x \n \
        orl $0x200, %%x \n \
        and $0xFFFFBFFF, %%x \n \
        push %%x \n \
        push $0x23 \n \
        push $0x40000000 \n \
        movw $0x2B, %%ax \n \
        movw %%ax, %%ds \n \
        iret" : "=m"(default_tss.esp0) : "m"(pd));

    while (1);
}

```

Pour vérifier avec Bochs l'utilisation de la pagination pour la tâche utilisateur, il est possible de poser un point d'arrêt sur une adresse virtuelle. L'adresse ci-dessous correspond au point d'entrée de la tâche utilisateur. Ensuite, la directive `info tab` indique comment la MMU traduit les adresses :

```

<bochs:1> lb 0x40000000
<bochs:2> c
...
<bochs:3> info tab
cr3: 0x00022000
0x00000000-0x003fffff -> 0x00000000-0x003fffff
0x40000000-0x40000fff -> 0x00100000-0x00100fff

```



```

This UGA/UBE Bios is released under the GNU LGPL

Please visit :
. http://bochs.sourceforge.net
. http://www.nongnu.org/vgabios

Bochs UBE Display Adapter enabled

Bochs BIOS - build: 03/17/08
$Revision: 1.194 $ $Date: 2007/12/23 19:46:27 $
Options: apmbios pcibios eltorito rombios32

Booting from Floppy...
loading kernel
kernel : gdt loaded
kernel : idt loaded
kernel : pic configured
kernel : tr loaded
kernel : paging enable
kernel : interrupts enable
kernel : create task
kernel : trying switch to user task...
task1
.....
A: NUM CAPS SCRL

```

XIV - Un système multitâche simple

- Exécuter plusieurs tâches simultanément grâce à l'ordonnanceur
- Un ordonnanceur appelé à chaque interruption d'horloge|outline
- Charger une tâche en mémoire
- Le contexte d'une tâche
- Exécuter plusieurs tâches

Sources

Le package contenant les sources est téléchargeable ici : [kernel_MultiTask.tgz](#)

XIV-A - Exécuter plusieurs tâches simultanément grâce à l'ordonnanceur

Nous avons détaillé dans une [partie précédente](#) comment le noyau charge et exécute une tâche utilisateur. Dans ce chapitre, nous allons voir comment implémenter un ordonnanceur (scheduler) simple qui permet au noyau d'exécuter plusieurs tâches en quasi simultanéité.

L'implémentation actuelle est très simple :

- le noyau charge plusieurs tâches ;
- les interruptions sont ensuite rétablies ;
- lors de chaque interruption d'horloge, le processeur appelle une fonction particulière dont le rôle est de partager le temps CPU entre les différentes tâches. Si une tâche est éligible, le noyau va :
 - 1 Sauvegarder la tâche en cours,
 - 2 Choisir la nouvelle tâche,
 - 3 Effectuer la commutation ;
- la nouvelle tâche s'exécute jusqu'à la nouvelle interruption d'horloge (...).

XIV-B - Un ordonnanceur appelé à chaque interruption d'horloge

Lors de chaque interruption d'horloge, le processeur fait appel à la fonction `schedule()` :

```
void isr_clock_int(void)
{
    static int tic = 0;
    static int sec = 0;
    tic++;
    if (tic%100 == 0) {
        sec++;
        tic = 0;
    }
    schedule();
}
```

La fonction `schedule()` peut être confrontée à différents cas :

- si aucune tâche n'est prête, le scheduler ne fait rien ;
- si aucune tâche ne tourne, mais qu'une tâche est prête, nous sommes dans une situation déjà vue précédemment ! Le noyau passe la main à la tâche utilisateur à la différence que celle-ci n'est pas chargée par le programme principal, mais suite à une interruption d'horloge ;
- si une tâche tourne et qu'au moins une autre tâche est prête : le scheduler effectue un changement de tâche. Cette partie est la plus délicate et nous place dans le vif du sujet... Pour passer la main à une nouvelle tâche, le noyau doit :

- 1 sauvegarder la tâche en cours,
- 2 charger la nouvelle tâche,
- 3 effectuer la commutation.

Le code de l'ordonnanceur est présent dans le fichier `schedule.c`

```
#include "types.h"
#include "gdt.h"
#include "process.h"

void schedule(void)
{
    u32* stack_ptr;
    u32 esp0, eflags;
    u16 ss, cs;

    /* Stocke dans stack_ptr le pointeur vers les registres sauvegardés */
    asm("mov (%?p), %%x; mov %%x, %0" : "=m" (stack_ptr) : );

    /* s'il n'y a pas de processus chargé et qu'au moins un est prêt, on le charge */
    if (current == 0 && n_proc) {
        current = &p_list[0];
    }
    /*
     * S'il y a un seul processus (qu'on laisse tourner) ou aucun
     * processus, on retourne directement.
     */
    else if (n_proc <= 1) {
        return;
    }
    /* s'il y a au moins deux processus, on commute vers le suivant */
    else if (n_proc > 1) {
        /* Sauver les registres du processus courant */
        current->regs.eflags = stack_ptr[16];
        current->regs.cs = stack_ptr[15];
        current->regs.eip = stack_ptr[14];
        current->regs.eax = stack_ptr[13];
        current->regs.ecx = stack_ptr[12];
        current->regs.edx = stack_ptr[11];
        current->regs.ebx = stack_ptr[10];
        current->regs.ebp = stack_ptr[8];
        current->regs.esi = stack_ptr[7];
        current->regs.edi = stack_ptr[6];
    }
}
```

```

current->regs.ds = stack_ptr[5];
current->regs.es = stack_ptr[4];
current->regs.fs = stack_ptr[3];
current->regs.gs = stack_ptr[2];

current->regs.esp = stack_ptr[17];
current->regs.ss = stack_ptr[18];

/*
 * La fonction l'interruption et la fonction schedule()
 * empilent un grand nombre de registres sur la pile.
 * L'instruction ci-dessous permet de repartir sur une pile
 * noyau propre une fois la commutation effectuée.
 */
default_tss.esp0 = (u32) (stack_ptr + 19);

/* Choix du nouveau processus (un simple roundrobin) */
if (n_proc > current->pid+1)
    current = &p_list[current->pid+1];
else
    current = &p_list[0];
}

/*
 * Empile les registres ss, esp, eflags, cs et eip nécessaires à la
 * commutation. Ensuite, la fonction do_switch() restaure les
 * registres, la table de pages du nouveau processus courant et commute
 * avec l'instruction iret.
 */
ss = current->regs.ss;
cs = current->regs.cs;
eflags = (current->regs.eflags | 0x200) & 0xFFFFBFFF;
esp0 = default_tss.esp0;

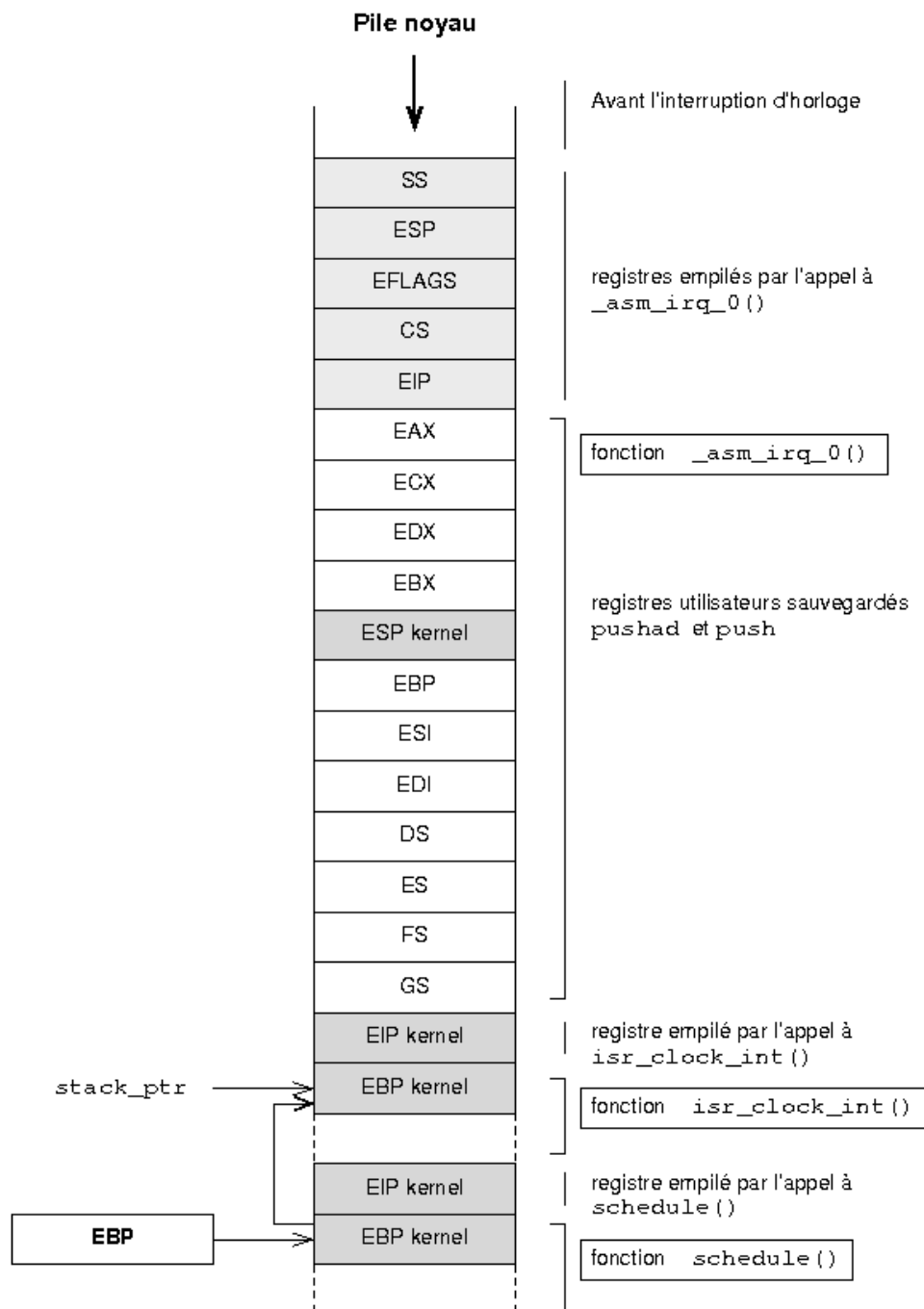
asm("    mov %0, %%esp; \
    push %1; \
    push %2; \
    push %3; \
    push %4; \
    push %5; \
    push %6; \
    ljmp $0x08, $do_switch" \
    :: \
    "m" (esp0), \
    "m" (ss), \
    "m" (current->regs.esp), \
    "m" (eflags), \
    "m" (cs), \
    "m" (current->regs.eip), \
    "m" (current)
    );
}

```

XIV-B-1 - Sauvegarder la tâche en cours

Nous allons tout d'abord détailler comment la fonction `schedule()` sauvegarde les registres de la tâche en cours (celle qui a été interrompue) dans la structure qui lui est dédiée et pointée par `current`.

Les registres à sauvegarder ont déjà été empilés sur la pile noyau par l'interruption d'horloge (ss, esp, eflags, cs et eip) et par l'appel explicite aux instructions `pushad` et `push`.



La sauvegarde des registres se fait dans la structure `struct process` associée à la tâche encore en cours et pointée par `current` :

```
struct process {
    unsigned int pid;

    struct {
        u32 eax, ecx, edx, ebx;
        u32 esp, ebp, esi, edi;
        u32 eip, eflags;
        u32 cs:16, ss:16, ds:16, es:16, fs:16, gs:16;
        u32 cr3;
    } regs __attribute__((packed));
} __attribute__((packed));
```

Le procédé pour récupérer ces valeurs sur la pile repose sur l'utilisation du registre `ebp` (voir pour plus de détails la partie sur les **Stack Frame**). La variable `stack_ptr` adresse la pile de façon à pouvoir ensuite récupérer facilement le contenu des registres sauvegardés lors de l'interruption :

```
/* stocke dans stack_ptr le pointeur vers les registres sauvegardés */
asm("mov (%?p), %?x; mov %?x, %0" : "=m" (stack_ptr) : );

/* sauve l'ensemble des registres */
current->regs.ss = stack_ptr[18];
current->regs.esp = stack_ptr[17];
current->regs.eflags = stack_ptr[16];
/* etc... */
```

XIV-B-2 - Choisir la nouvelle tâche

L'ordonnanceur choisit une nouvelle tâche et fait pointer la variable `current` vers la structure de la nouvelle tâche à activer :

```
/* Choix du nouveau processus (un simple round robin) */
if (n_proc > current->pid+1)
    current = &p_list[current->pid+1];
else
    current = &p_list[0];
```

XIV-B-3 - Commuter

Une fois que l'ancienne tâche est sauvegardée et que la nouvelle tâche est choisie, le scheduler doit charger les registres du processeur avec les valeurs sauvegardées précédemment avant de commuter.

En premier lieu, le pointeur de pile noyau du TSS est initialisé de façon à ce que :

- la pile noyau, sur laquelle des registres ont été sauvegardés, soit nettoyée une fois la commutation effectuée ;
- la pile noyau soit préparée pour la nouvelle tâche.

```
default_tss.esp0 = (u32) (stack_ptr + 19);
```



L'utilisation d'une seule pile noyau pour l'ensemble des tâches a l'avantage de la simplicité, mais a également un inconvénient important : il est impossible de préempter une tâche alors qu'elle effectue un appel système. La raison est exposée en détail au chapitre suivant qui traite de la réalisation d'un système multitâche avec des appels système interruptibles.

Ensuite, les valeurs sauvegardées correspondant aux registres `ss`, `esp`, `eflags`, `cs` et `eip` sont empilées afin de préparer la commutation. La fonction `do_switch()` est appelée pour effectuer la commutation proprement dite :

```
asm("    mov %0, %%esp; \
    push %1; \
    push %2; \
    push %3; \
    push %4; \
    push %5; \
    push %6; \
    ljmp $0x08, $do_switch" \
    :: \
    "m" (esp0), \
    "m" (ss), \
    "m" (current->regs.esp), \
    "m" (eflags), \
    "m" (cs), \
```

```
"m" (current->regs.eip), \  
"m" (current)  
);
```

XIV-B-3-a - La fonction do_switch()

La fonction `do_switch()`, implémentée dans le fichier `sched.asm`, n'est pas très compliquée :

```
global do_switch  
  
do_switch:  
    ; recuper l'adresse de *current  
    mov esi, [esp]  
    pop eax                ; depile @current  
  
    ; prépare les registres  
    push dword [esi+4]     ; eax  
    push dword [esi+8]     ; ecx  
    push dword [esi+12]    ; edx  
    push dword [esi+16]    ; ebx  
    push dword [esi+24]    ; ebp  
    push dword [esi+28]    ; esi  
    push dword [esi+32]    ; edi  
    push dword [esi+48]    ; ds  
    push dword [esi+50]    ; es  
    push dword [esi+52]    ; fs  
    push dword [esi+54]    ; gs  
  
    ; enlève le mask du PIC  
    mov al, 0x20  
    out 0x20, al  
  
    ; charge table des pages  
    mov eax, [esi+56]  
    mov cr3, eax  
  
    ; charge les registres  
    pop gs  
    pop fs  
    pop es  
    pop ds  
    pop edi  
    pop esi  
    pop ebp  
    pop ebx  
    pop edx  
    pop ecx  
    pop eax  
  
    ; retourne  
    iret
```

- 1 L'adresse de la structure de la nouvelle tâche est récupérée dans le registre `esi` ;
- 2 Un signal est envoyé au PIC pour rétablir les interruptions hardware ;
- 3 Le répertoire de pages de la nouvelle tâche est chargé (note : à ce point-là, nous sommes toujours dans l'espace d'adressage du noyau, partagé par toutes les tâches) ;
- 4 Les registres sont directement chargés à partir de la structure de la nouvelle tâche ;
- 5 L'instruction `iret` retourne de l'interruption et charge les registres `ss`, `esp`, `eflags`, `cs` et `eip`.

À ce moment, la commutation est achevée et la nouvelle tâche utilisateur s'exécute !

XIV-B-4 - Note concernant les appels système

Dans cette première implémentation, les tâches ne peuvent être préemptées alors qu'elles réalisent un appel système, car le partage de la pile noyau conduirait à des incohérences. Il faut donc désactiver les interruptions lors des appels. Cela se fait par le choix d'un INTGATE à la place d'un TRAPGATE :

```
init_idt_desc(0x08, (u32) _asm_syscalls, INTGATE|0x6000, &idt[48]); /* appels système - int 0x30
*/
```

XIV-C - Charger une tâche en mémoire

Le chargement d'une tâche, un peu sur le même modèle que dans les chapitres précédent, se fait grâce à la fonction `load_task()`. Dans l'exemple ci-dessous, le noyau charge la tâche `task1()` à l'adresse physique `0x100000` :

```
load_task((u32*) 0x100000, (u32*) &task1, 0x2000);
```

La fonction `load_task()` est définie dans le fichier `process.c`

```
#include "lib.h"
#include "mm.h"

#define __PLIST__
#include "process.h"

void load_task(u32 * code_phys_addr, u32 * fn, unsigned int code_size)
{
    u32 page_base, pages;
    u32 *pd;
    int i;

    /* Copie du code à l'adresse spécifiée */
    memcpy((char *) code_phys_addr, (char *) fn, code_size);

    /* Mise à jour du bitmap */
    page_base = (u32) PAGE(code_phys_addr);

    if (code_size % PAGE_SIZE)
        pages = code_size / PAGE_SIZE + 1;
    else
        pages = code_size / PAGE_SIZE;

    for (i = 0; i < pages; i++)
        set_page_frame_used(page_base + i);

    /* Création du répertoire et des tables de pages */
    pd = pd_create(code_phys_addr, code_size);

    /* Initialisation des registres */
    p_list[n_proc].pid = n_proc;
    p_list[n_proc].regs.ss = 0x33;
    p_list[n_proc].regs.esp = 0x40001000;
    p_list[n_proc].regs.cs = 0x23;
    p_list[n_proc].regs.eip = 0x40000000;
    p_list[n_proc].regs.ds = 0x2B;
    p_list[n_proc].regs.cr3 = (u32) pd;

    n_proc++;
}
```

- 1 Elle commence par copier la fonction à l'adresse physique voulue et met à jour le bitmap de pages ;
- 2 Elle crée le répertoire et les tables de pages avec la fonction `pd_create()`. Les pages physiques occupées par le code de la tâche sont mappées sur les pages virtuelles en `0x40000000` ;
- 3 La pile noyau est pour le moment sur la même page que celle contenant le code et le pointeur de pile est en `0x40000FF0` ;

- 4 Elle demande une page pour la pile noyau. L'adresse linéaire de cette page correspond à son adresse physique (elle se situe dans l'espace d'adressage du noyau) ;
- 5 Elle initialise la structure process de cette tâche.

Ce code n'est pas très compliqué, car l'objet de ce chapitre est seulement de montrer comment implémenter de façon simple un système multitâche. J'ai donc délibérément supprimé tout ce qui peut « alourdir » le code. Il est cependant nécessaire d'en pointer deux importantes limitations :

- le choix de pages libres pour le chargement de la tâche n'est pas géré par l'allocateur ;
- les pages choisies sont successives (il n'y a donc pas de gestion de la fragmentation).

XIV-D - Le contexte d'une tâche

Lorsqu'une tâche est interrompue pour laisser la main à une autre, son contexte d'exécution doit être intégralement conservé afin qu'elle puisse plus tard reprendre là où elle avait été interrompue. La structure `struct process` définit pour chaque tâche :

- un identifiant, le **pid** ;
- l'ensemble des registres du processeur susceptibles d'être utilisés par la tâche.

Un tableau de `struct process` est défini pour conserver le contexte de toutes nos tâches. Le nombre maximum de tâches est pour le moment très restreint, juste 32 :

```
struct process p_list[32];
```

La variable `*current` pointe sur la tâche en cours d'exécution et `n_proc` nous renseigne sur le nombre total de tâches :

```
struct process *current = 0;  
int n_proc = 0;
```

XIV-E - Exécuter plusieurs tâches

La fonction principale `main()` du noyau est très simple : elle initialise les structures habituelles puis elle charge les tâches utilisateur en mémoire et rétablit les interruptions. À chaque tic d'horloge, l'ordonnanceur est activé et donne la main à une nouvelle tâche.



XV - Un noyau multitâche avec des appels système préemptibles

- **Un scheduler qui tient compte du contexte d'exécution**
- **Utiliser les TrapGate pour rendre les appels système interruptibles**

Sources

Le package contenant les sources est téléchargeable ici : [kernel_MultiTask_Preemptible.tgz](#)

XV-A - Un scheduler qui tient compte du contexte d'exécution

XV-A-1 - Une pile noyau par tâche : pourquoi ?

Quand une tâche fait un appel système, la pile noyau est automatiquement utilisée par le processeur. Les routines de traitement de cet appel utilisent alors cette pile pour stocker des données, passer des paramètres à des fonctions, etc. Une fois que l'appel système est terminé, les données temporaires sont dépilées et la pile noyau retourne à son état initial, tel qu'elle était avant l'appel.

Si l'on autorise les interruptions et la commutation de tâches pendant un appel système, voilà ce qui risque de se passer :

- 1 Par exemple, la tâche A fait un appel système et utilise la pile noyau pour traiter cet appel ;
- 2 L'ordonnanceur est activé alors que l'appel n'est pas terminé et il donne la main à la tâche B ;
- 3 La tâche B fait elle aussi un appel système, et elle utilise donc elle aussi la pile du noyau qui contenait déjà les données en cours pour le traitement de l'appel de la tâche A. À ce moment, la pile noyau contient des données de A et des données de B ;
- 4 L'ordonnanceur active de nouveau la tâche A et celle-ci reprend sans se douter que la pile a été modifiée. Comme elle va utiliser la pile en supposant qu'elle est dans l'état où elle l'avait laissé, elle va corrompre les données de la tâche B ;
- 5 La tâche B reprend, mais la pile est corrompue... et c'est le drame !

Pour éviter ce scénario catastrophe, il existe une solution très simple : il suffit que chaque tâche ait sa propre pile noyau !

XV-A-2 - Une pile noyau par tâche : comment

La fonction `load_task()` est modifiée afin d'associer à chaque tâche sa propre pile noyau : `process.c`.

```
#include "lib.h"
#include "mm.h"

#define __PLIST__
#include "process.h"

void load_task(u32 * code_phys_addr, u32 * fn, unsigned int code_size)
{
    u32 page_base, pages, kstack_base;
    u32 *pd;
    int i;

    /* Copie du code a l'adresse spécifiée */
    memcpy((char *) code_phys_addr, (char *) fn, code_size);

    /* Mise a jour du bitmap */
    page_base = (u32) PAGE(code_phys_addr);

    if (code_size % PAGE_SIZE)
        pages = code_size / PAGE_SIZE + 1;
    else
        pages = code_size / PAGE_SIZE;

    for (i = 0; i < pages; i++)
        set_page_frame_used(page_base + i);

    /* Création du répertoire et des tables de pages */
    pd = pd_create(code_phys_addr, code_size);

    kstack_base = (u32) get_page_frame();
    if (kstack_base > 0x400000) {
        printk("not enough memory to create a kernel stack\n");
        return;
    }

    /* Initialisation des registres */
    p_list[n_proc].pid = n_proc;
    p_list[n_proc].regs.ss = 0x33;
    p_list[n_proc].regs.esp = 0x40001000;
    p_list[n_proc].regs.eflags = 0x0;
    p_list[n_proc].regs.cs = 0x23;
    p_list[n_proc].regs.eip = 0x40000000;
    p_list[n_proc].regs.ds = 0x2B;
    p_list[n_proc].regs.es = 0x2B;
    p_list[n_proc].regs.fs = 0x2B;
    p_list[n_proc].regs.gs = 0x2B;
    p_list[n_proc].regs.cr3 = (u32) pd;

    p_list[n_proc].kstack.ss0 = 0x18;
    p_list[n_proc].kstack.esp0 = kstack_base + PAGE_SIZE;

    p_list[n_proc].regs.eax = 0;
    p_list[n_proc].regs.ecx = 0;
    p_list[n_proc].regs.edx = 0;
    p_list[n_proc].regs.ebx = 0;

    p_list[n_proc].regs.ebp = 0;
    p_list[n_proc].regs.esi = 0;
    p_list[n_proc].regs.edi = 0;

    n_proc++;
}
```

Le code ci-dessous alloue à une tâche sa propre pile noyau. Par simplicité, une pile occupe une page :

```
kstack_base = (u32) get_page_frame();

/* ... */

p_list[n_proc].kstack.ss0 = 0x18;
p_list[n_proc].kstack.esp0 = kstack_base + PAGE_SIZE;
```

Suite à une interruption, le processeur empile automatiquement des données sur la pile noyau de la tâche interrompue, sans compter celles ajoutées par les différentes routines du noyau. Il est crucial qu'à l'issue d'une interruption, la pile noyau utilisée soit remise dans le même état qu'avant l'interruption.

XV-A-3 - L'ordonnanceur

La complexité de l'ordonnanceur provient essentiellement des difficultés induites par la gestion des différentes piles noyau et utilisateur.

Le code de l'ordonnanceur est réparti dans deux fichiers :

- `schedule.c` contient la partie préparatoire au changement de contexte

```
#include "types.h"
#include "gdt.h"
#include "process.h"

void switch_to_task(int n, int mode)
{
    u32 kesp, eflags;
    u16 kss, ss, cs;

    current = &p_list[n];

    /* charger tss */
    default_tss.ss0 = current->kstack.ss0;
    default_tss.esp0 = current->kstack.esp0;

    /*
     * Empile les registres ss, esp, eflags, cs et eip nécessaires à la
     * commutation. Ensuite, la fonction do_switch() restaure les
     * registres, la table de pages du nouveau processus courant et commute
     * avec l'instruction iret.
     */
    ss = current->regs.ss;
    cs = current->regs.cs;
    eflags = (current->regs.eflags | 0x200) & 0xFFFFBFFF;

    if (mode == USERMODE) {
        kss = current->kstack.ss0;
        kesp = current->kstack.esp0;
    } else { /* KERNELMODE */
        kss = current->regs.ss;
        kesp = current->regs.esp;
    }

    asm("    mov %0, %%ss; \
        mov %1, %%esp; \
        cmp %[KMODE], %[mode]; \
        je next; \
        push %2; \
        push %3; \
        next: \
        push %4; \
        push %5; \
        push %6; \
        push %7; \
        ljmp $0x08, $do_switch"
        :: \
```

```

        "m"(kss), \
        "m"(kesp), \
        "m"(ss), \
        "m"(current->regs.esp), \
        "m"(eflags), \
        "m"(cs), \
        "m"(current->regs.eip), \
        "m"(current), \
        [KMODE] "i"(KERNELMODE), \
        [mode] "g"(mode)
    );
}

void schedule(void)
{
    struct process *p;
    u32 *stack_ptr;

    /* Stocke dans stack_ptr le pointeur vers les registres sauvegardés */
    asm("mov (%?p), %?x; mov %?x, %0": "=m"(stack_ptr) : );

    /* S'il n'y a pas de processus chargé et qu'au moins un est prêt, on le charge */
    if (current == 0 && n_proc) {
        switch_to_task(0, USERMODE);
    }
    /*
     * S'il y a un seul processus (qu'on laisse tourner) ou aucun
     * processus, on retourne directement.
     */
    else if (n_proc <= 1) {
        return;
    }
    /* S'il y a au moins deux processus, on commute vers le suivant */
    else if (n_proc > 1) {
        /* Sauver les registres du processus courant */
        current->regs.eflags = stack_ptr[16];
        current->regs.cs = stack_ptr[15];
        current->regs.eip = stack_ptr[14];
        current->regs.eax = stack_ptr[13];
        current->regs.ecx = stack_ptr[12];
        current->regs.edx = stack_ptr[11];
        current->regs.ebx = stack_ptr[10];
        current->regs.ebp = stack_ptr[8];
        current->regs.esi = stack_ptr[7];
        current->regs.edi = stack_ptr[6];
        current->regs.ds = stack_ptr[5];
        current->regs.es = stack_ptr[4];
        current->regs.fs = stack_ptr[3];
        current->regs.gs = stack_ptr[2];

        if (current->regs.cs != 0x08) {
            current->regs.esp = stack_ptr[17];
            current->regs.ss = stack_ptr[18];
        } else {
            /* Interruption pendant un appel système */
            current->regs.esp = stack_ptr[9] + 12;
            current->regs.ss = default_tss.ss0;
        }

        /* Sauver le tss */
        current->kstack.ss0 = default_tss.ss0;
        current->kstack.esp0 = default_tss.esp0;

        /* Choix du nouveau processus (un simple roundrobin) */
        if (n_proc > current->pid + 1)
            p = &p_list[current->pid + 1];
        else
            p = &p_list[0];

        /* Commutation */
        if (p->regs.cs != 0x08)
            switch_to_task(p->pid, USERMODE);
        else

```

```

        }
        switch_to_task(p->pid, KERNELMODE);
    }
}

```

- sched.asm contient le code qui effectue effectivement le changement de contexte

```

global do_switch

do_switch:
    ; récupérer l'adresse de *current
    mov esi, [esp]
    pop eax                ; depile @current

    ; prépare les registres
    push dword [esi+4]     ; eax
    push dword [esi+8]     ; ecx
    push dword [esi+12]    ; edx
    push dword [esi+16]    ; ebx
    push dword [esi+24]    ; ebp
    push dword [esi+28]    ; esi
    push dword [esi+32]    ; edi
    push dword [esi+48]    ; ds
    push dword [esi+50]    ; es
    push dword [esi+52]    ; fs
    push dword [esi+54]    ; gs

    ; enlève le mask du PIC
    mov al, 0x20
    out 0x20, al

    ; charge table des pages
    mov eax, [esi+56]
    mov cr3, eax

    ; charge les registres
    pop gs
    pop fs
    pop es
    pop ds
    pop edi
    pop esi
    pop ebp
    pop ebx
    pop edx
    pop ecx
    pop eax

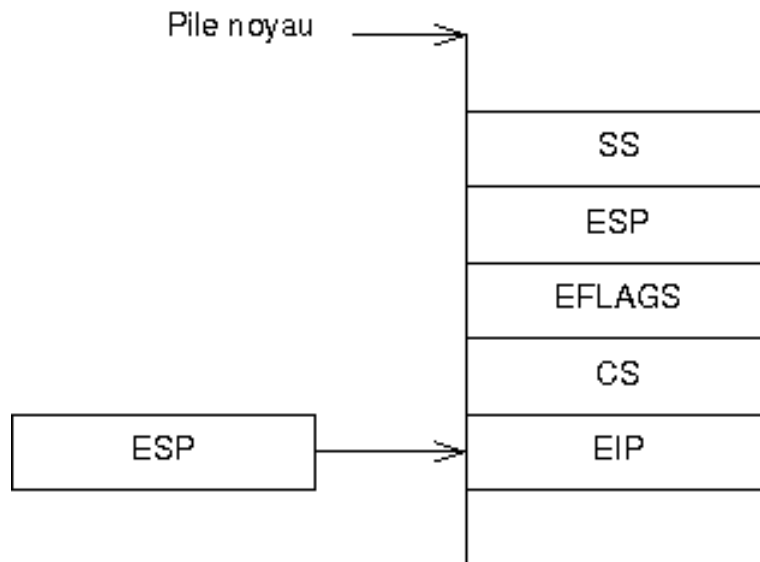
    ; retourne
    iret

```

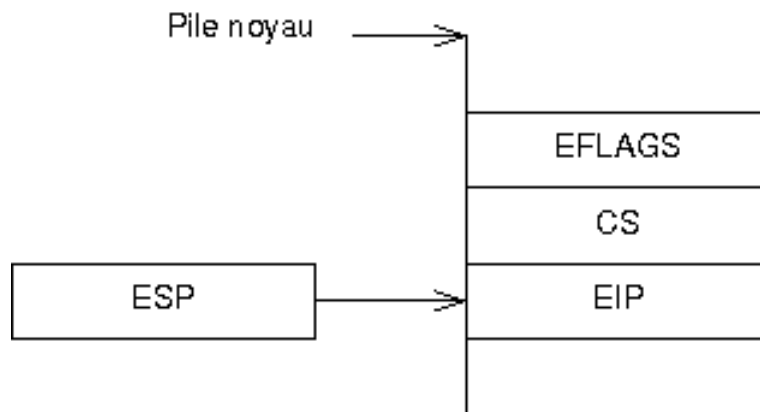
Le design de l'ordonnanceur est plus compliqué, car :

- il doit prendre en compte le contexte de la tâche lors de son interruption ;
- chaque tâche a sa propre pile noyau ;
- l'ordonnanceur change de pile noyau pendant la commutation ;
- la pile noyau ne contient pas le même nombre de registres empilés selon que la tâche interrompue était en mode utilisateur ou en mode noyau. L'interruption d'une tâche en mode utilisateur provoque automatiquement l'empilement des registres ss, esp, eflags, cs et eip. Quand l'interruption survient alors que la tâche est en mode noyau, les seuls registres empilés sont : eflags, cs et eip.

Schémas de la pile noyau après l'interruption d'une tâche en mode utilisateur :



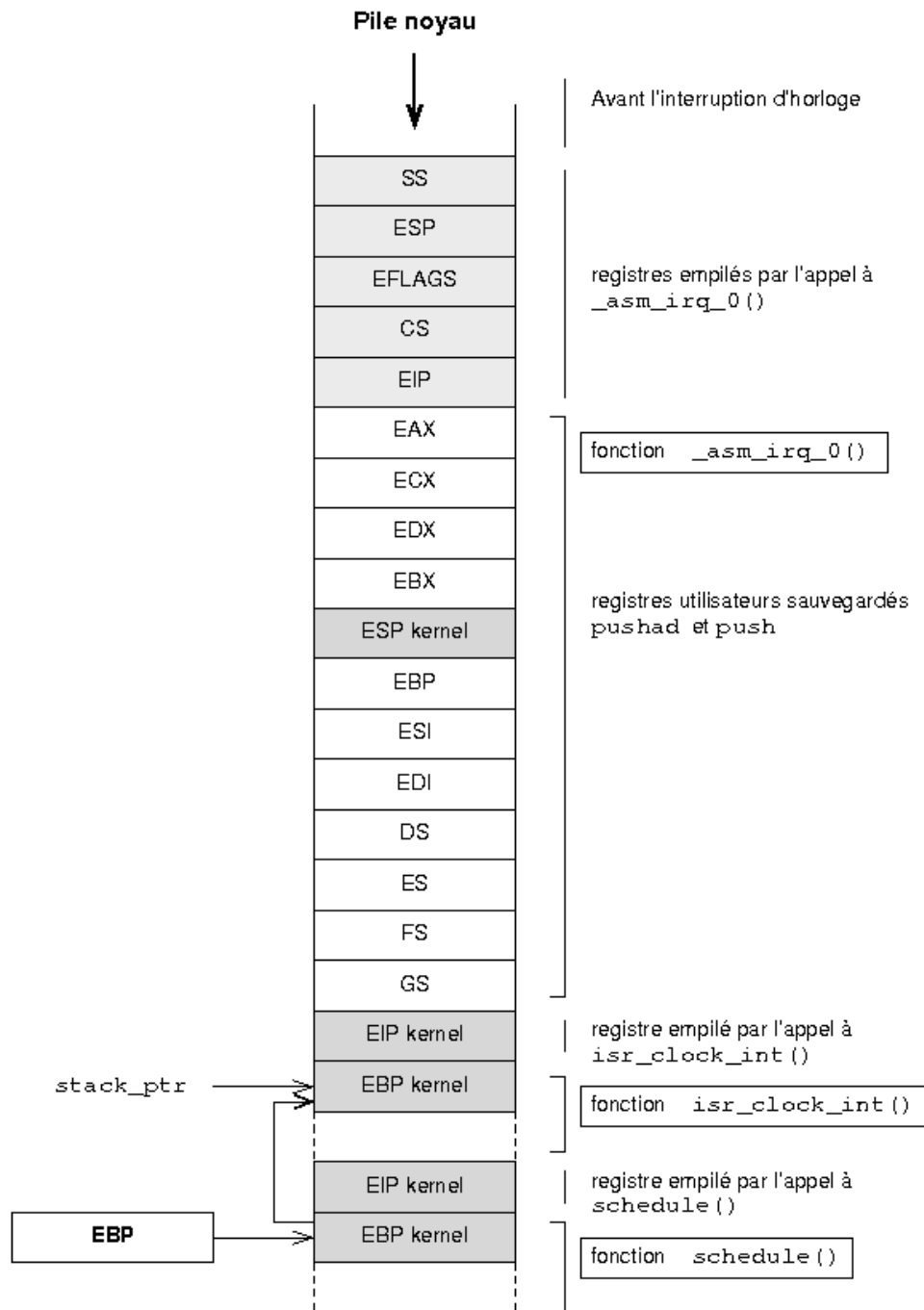
Schémas de la pile noyau après l'interruption d'une tâche en mode noyau :



Toutes les contraintes supportées par l'ordonnanceur peuvent se résumer à :

- sauvegarder les registres de la tâche interrompue ;
- remettre la pile noyau dans l'état précédent l'interruption avant de quitter celle-ci.

Le schéma suivant représente la pile noyau lors de l'appel à la fonction `schedule()` dans le cas où le processeur était en mode utilisateur lors de l'interruption :



Dans le cas où le processeur était en mode noyau au moment de l'interruption, le schéma est très sensiblement différent, car les registres `ss` et `esp` n'ont pas été empiés.

Notre but est de sauvegarder l'état de l'ensemble des registres de la tâche interrompue, ce qui ne pose aucun problème pour l'ensemble d'entre eux (`eax`, `ebx`, etc.). La seule difficulté est de retrouver les valeurs de `ss` et de `esp` juste avant l'interruption d'horloge, car si l'interruption est survenue pendant un appel système, on ne peut procéder de la même façon pour retrouver ces valeurs, car elles n'ont pas été empiées :

```
if (current->regs.cs != 0x08) {
    current->regs.esp = stack_ptr[17];
    current->regs.ss = stack_ptr[18];
}
else { /* interruption pendant un appel systeme */
    current->regs.esp = stack_ptr[9] + 12; /* equivalent a : &stack_ptr[17] */
    current->regs.ss = default_tss.ss0;
```

```
}

```

- Le premier cas est celui de l'interruption d'une tâche en mode utilisateur. L'appel système a sauvegardé tous les registres, dont ss et esp, sur la pile noyau de la tâche. C'est un cas assez simple déjà vu au chapitre précédent.
- Le deuxième cas concerne l'interruption d'une tâche en mode noyau. Le processeur utilisait la même pile avant l'interruption et esp pointait juste au sommet des registres empilés par l'interruption. Pour retrouver cette valeur, plusieurs calculs sont possibles :
 - `stack_ptr[9]` contient la valeur du pointeur de pile juste **après** l'interruption (et **avant** l'appel à `push` et à `pushad`). Avant l'interruption, `esp` pointait sur la même pile, mais juste avant les trois registres `eflags`, `cs` et `eip` empilés par l'interruption. Nous avons donc : 3 registres * 4 octets par registre = 12. `stack_ptr[9] + 12` pointe donc au sommet de la pile,
 - le sommet de la pile est situé 17 registres au-dessus du pointeur `stack_ptr`, on pourrait donc utiliser l'un des deux calculs suivants : `stack_ptr + 17 * 4` ou `&stack_ptr[17]`.

Après avoir sélectionné une nouvelle tâche, pointée par `*p`, l'ordonnanceur teste si cette nouvelle tâche à activer avait été interrompue en mode utilisateur ou en mode noyau. La fonction `switch_to_task()` est appelée avec le pid de la tâche à activer et le mode dans lequel il faut l'activer :

```
/* commutation */
if (p->regs.cs != 0x08)
    switch_to_task(p->pid, USERMODE);
else
    switch_to_task(p->pid, KERNELMODE);

```

XV-A-3-a - La fonction `switch_to_task()`

La fonction `switch_to_task()` prépare la commutation :

- elle change de pile noyau et passe sur la pile de la nouvelle tâche ;
- elle empile les registres nécessaires à l'instruction `iret` ;
- elle passe la main à la fonction `do_switch()` qui charge les registres et effectue la commutation.

Les variables `current->regs.ss`, `current->regs.cs` et `current->regs.eflags` ne peuvent être directement utilisées dans la fonction `asm()`. Ces trois affectations initialisent des variables qui pourront elles être passées en paramètre à la fonction assembleur :

```
ss = current->regs.ss;
cs = current->regs.cs;
eflags = (current->regs.eflags | 0x200) & 0xFFFFBFFF;

```

Ce code récupère les paramètres de la pile noyau pour la nouvelle tâche. Si celle-ci est en mode utilisateur, on va utiliser une pile noyau « propre ». Si en revanche elle est en mode noyau, il suffit de reprendre les valeurs `ss` et `esp` sauvegardées lors du changement de contexte :

```
if (mode == USERMODE) {
    kss = current->kstack.ss0;
    kesp = current->kstack.esp0;
}
else { /*KERNELMODE */
    kss = current->regs.ss;
    kesp = current->regs.esp;
}

```

Cette fonction effectue un gros travail :

- 1 Les deux premières instructions basculent sur la nouvelle pile noyau ;

- 2 Si la tâche est en mode utilisateur, elle empile les valeurs sauvegardées des registres `ss` et `esp` ;
- 3 Elle empile les valeurs sauvegardées des registres `eflags`, `cs` et `eip` ;
- 4 Elle passe la main à la fonction `do_switch()` qui charge les registres avec les valeurs sauvegardées et effectue la commutation avec `iret`.

```
asm("  mov %0, %%ss; \
      mov %1, %%esp; \
      cmp %[KMODE], %[mode]; \
      je next; \
      push %2; \
      push %3; \
      next: \
      push %4; \
      push %5; \
      push %6; \
      push %7; \
      ljmp $0x08, $do_switch" \
      :: \
      "m" (kss), \
      "m" (kesp), \
      "m" (ss), \
      "m" (current->regs.esp), \
      "m" (eflags), \
      "m" (cs), \
      "m" (current->regs.eip), \
      "m" (current), \
      [KMODE] "i" (KERNELMODE), \
      [mode] "g" (mode)
);
```

XV-B - Utiliser les TrapGate pour rendre les appels système interruptibles

Les appels système sont rendus interruptibles :

```
init_idt_desc(0x08, (u32) _asm_syscalls, TRAPGATE, &kidt[48]); /* appels systeme - int 0x30 */
```

Pour tester le fait qu'ils sont bien interruptibles, une temporisation est ajoutée pour les rendre plus lents : `syscalls.c`.

```
#include "types.h"
#include "lib.h"
#include "io.h"

void do_syscalls(int sys_num)
{
    char *u_str;
    int i;

    if (sys_num == 1) {
        asm("mov %?x, %0": "=m"(u_str) :);
        for (i = 0; i < 100000; i++); /* temporisation */
        cli;
        printk(u_str);
        sti;
    } else {
        printk("unknown syscall %d\n", sys_num);
    }

    return;
}
```

L'utilisation de la fonction `cli` peut cependant être requise, car il n'est pas souhaitable qu'un appel système soit interrompu à certains moments critiques de son exécution. Notamment ici, pour avoir un affichage cohérent, j'ai préféré désactiver les interruptions le temps de l'affichage de la chaîne.

XVI - Un noyau qui boote avec Grub

- [Le standard multiboot](#)
- [Un premier noyau minimaliste démarré par Grub](#)
- [Compiler et linker le noyau](#)
- [Créer et mettre à jour une image de disquette avec Grub](#)
- [Booter Pépin avec Grub](#)

Les sources

Le package contenant les sources est téléchargeable ici : [kernel_GrubEnable.tgz](#)

XVI-A - Le standard multiboot

Le document [GNU/Multiboot spec](#). définit un standard qui uniformise la séquence de boot des différents systèmes d'exploitation. **Grub** est un boot loader qui permet de démarrer un grand nombre de systèmes différents pour peu qu'ils répondent à ce standard. Pour répondre à ce standard, un noyau doit :

- être un fichier exécutable 32 bits standard ;
- l'image du noyau doit inclure un en-tête spécial, le [multiboot header](#), dans ses 8192 premiers octets.

XVI-B - Un premier noyau minimaliste démarré par Grub



L'utilisation de Grub2 est détaillée en [annexe](#)

Pour illustrer le boot avec Grub, le package suivant contient un noyau minimaliste répondant au standard multiboot : [kernel_SimpleGrub.tgz](#)

XVI-B-1 - L'en-tête

```
global _start, start
extern kmain

?fine MULTIBOOT_HEADER_MAGIC 0x1BADB002
?fine MULTIBOOT_HEADER_FLAGS 0x00000003
?fine CHECKSUM -(MULTIBOOT_HEADER_MAGIC + MULTIBOOT_HEADER_FLAGS)

_start:
    jmp start

; The Multiboot header
align 4
multiboot_header:
dd MULTIBOOT_HEADER_MAGIC
dd MULTIBOOT_HEADER_FLAGS
dd CHECKSUM
; ----- Multiboot Header Ends Here -----

start:
    push ebx
    call kmain

    cli ; stop interrupts
    hlt ; halt the CPU
```

XVI-B-1-a - Le programme principal

Grub permet de fournir au noyau plusieurs informations dont la taille de la mémoire physique disponible. Cette information est transmise au noyau via la structure de données `struct mb_partial_info` :

```
void printk(char*, ...);

struct mb_partial_info {
    unsigned long flags;
    unsigned long low_mem;
    unsigned long high_mem;
    unsigned long boot_device;
    unsigned long cmdline;
};

void kmain(struct mb_partial_info *mbi)
{
    printk("Grub example kernel is loaded...\n");
    printk("RAM detected : %uk (lower), %uk (upper)\n", mbi->low_mem, mbi->high_mem);
    printk("Done.\n");
}
```

XVI-C - Compiler et linker le noyau

La compilation et le linkage pour obtenir le nouveau noyau changent, car il faut que l'en-tête du Multiboot se trouve au début de la zone texte. Plusieurs méthodes sont possibles :

- 1 Décrire une section pour l'en-tête et utiliser un script pour le linkage avec `ld` ;
- 2 Décrire une section pour l'en-tête et utiliser `ld` en ligne de commande avec des paramètres approximatifs ;
- 3 Lors du linkage, mettre le fichier objet `boot.o` en premier afin qu'il soit relogé au début du noyau :

```
ld -Ttext=100000 --entry=_start boot.o kernel.o screen.o printk.o -o kernel
```

La commande shell `mbchk` permet de vérifier la compatibilité du noyau généré avec le standard Multiboot :

```
mbchk kernel
kernel: The Multiboot header is found at the offset 4104.
kernel: Page alignment is turned on.
kernel: Memory information is turned on.
kernel: Address fields is turned off.
kernel: All checks passed.
```

On peut aussi vérifier où sont relogés les différents objets avec la commande shell `nm` :

```
nm -n kernel
00100000 T _start
00100008 t multiboot_header
00100014 T start
0010001c T kmain
0010005c T scrollup
001000ec T putcar
00100218 T strlen
00100243 T itoa
00100309 T printk
001016e0 D kY
001016e1 D kattr
001016e4 A __bss_start
001016e4 A _edata
001016e4 B kX
001016e8 A _end
```

XVI-D - Créer et mettre à jour une image de disquette avec Grub

La création d'une telle image nécessite la permission de root.



*Le script ci-dessous est potentiellement dangereux et chaque commande exécutée en tant que root sur un système UNIX doit être bien comprise. Si une commande du script ci-dessous ne vous semble pas claire, je vous conseille de prendre le temps qu'il faut pour étudier la documentation accessible via la commande shell `man`. Même si cet effort vous semble futile et n'a en apparence pas de rapport avec le développement d'un noyau, je vous garantis que les connaissances acquises vous seront très utiles. Pour ceux qui débutent en administration système, je recommande l'excellentissime **Guide du Rootard**.*

```
# création d'une image vide avec ext2fs
dd if=/dev/zero of=floppyA bs=1024 count=1440
mkfs floppyA

sudo -s

# montage de l'image
mkdir /mnt/loop
mount -o loop -t ext2 floppyA /mnt/loop

# création de l'arborescence initiale
mkdir /mnt/loop/grub

cp /boot/grub/stage* /mnt/loop/grub

cat > /mnt/loop/grub/menu.lst << EOF
title=Pépin
root (fd0)
kernel /kernel
boot
EOF

cp kern/kernel /mnt/loop
umount /mnt/loop

# installation de grub
grub --device-map=/dev/null << EOF
device (fd0) floppyA
root (fd0)
setup (fd0)
quit
EOF
```

Une fois l'image construite, la mise à jour du noyau est plus simple :

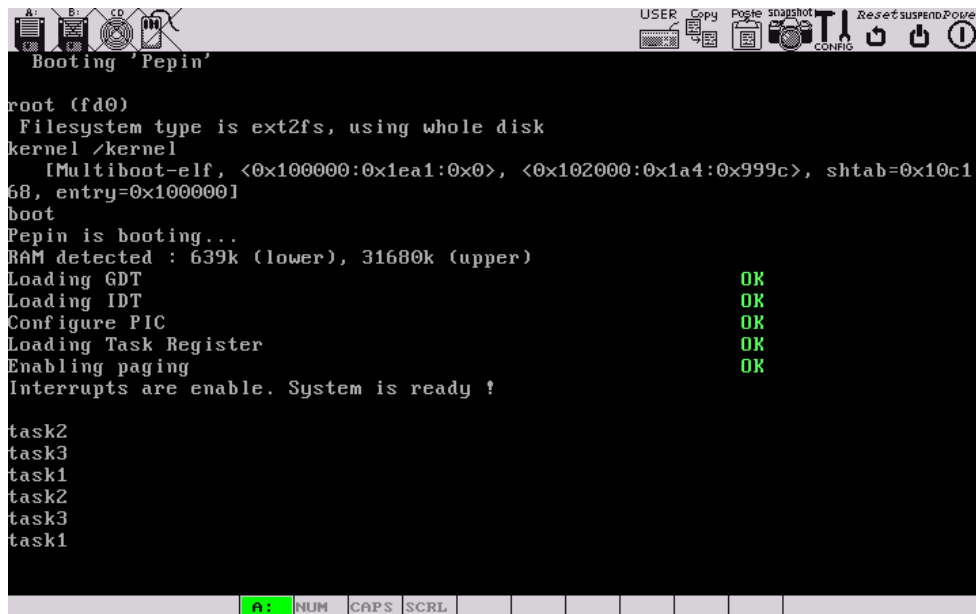
```
mount -o loop -t ext2 floppyA /mnt/loop
cp kern/kernel /mnt/loop
umount /mnt/loop
```

XVI-E - Booter Pépin avec Grub

Le noyau est chargé par Grub en 0x100000.

Certaines plages d'adresses sont réservées par le noyau ou pour le hardware :

0x0	0xFFFF	GDT et IDT
0x1000	0x9FFFF	libre
0xA0000	0xFFFFF	réservé pour le hardware
0x100000	0x1FFFFFF	réservé pour le noyau
0x200000	max	libre



```

Bootimg 'Pepin'

root (fd0)
Filesystem type is ext2fs, using whole disk
kernel /kernel
[Multiboot-elf, <0x100000:0x1ea1:0x0>, <0x102000:0x1a4:0x999c>, shtab=0x10c1
68, entry=0x100000]
boot
Pepin is booting...
RAM detected : 639k (lower), 31680k (upper)
Loading GDT          OK
Loading IDT          OK
Configure PIC        OK
Loading Task Register OK
Enabling paging      OK
Interrupts are enable. System is ready !

task2
task3
task1
task2
task3
task1

```

XVII - Gérer la mémoire physique et la mémoire virtuelle

- Une nouvelle organisation de la mémoire physique et de la mémoire virtuelle
- Allouer dynamiquement de la mémoire pour le noyau
- Mettre à jour le répertoire et les tables de pages
- Compléments sur les répertoires de pages
- Compléments sur la pagination
- Structure d'une tâche

Les sources

Le package contenant les sources est téléchargeable ici : [kernel_AdvMemory.tgz](#).

XVII-A - Une nouvelle organisation de la mémoire physique et de la mémoire virtuelle

XVII-A-1 - Comment gérer « les » mémoires

Nous avons vu que le noyau gère deux types de mémoire : la mémoire physique, réellement disponible et utilisable, et la mémoire virtuelle. À tout moment, le noyau a besoin de savoir quels sont les espaces de mémoire physique et virtuelle qui sont utilisés et libres et cela passe par la mise en place :

- d'une organisation claire de la mémoire physique et de la mémoire virtuelle ;
- de deux systèmes de gestion de la mémoire (un pour chaque type de mémoire) afin de répondre aux besoins du noyau et des tâches utilisateur.

Cette étape du développement du noyau est assez délicate en raison des contraintes liées à la pagination. Le but de cette partie est néanmoins de présenter un système de gestion de la mémoire le plus simple possible. Mais contrairement aux chapitres précédents, peu de code est présenté ici. L'accent est avant tout mis sur les enjeux et la logique sous-tendant la mise en place d'une gestion de la mémoire.

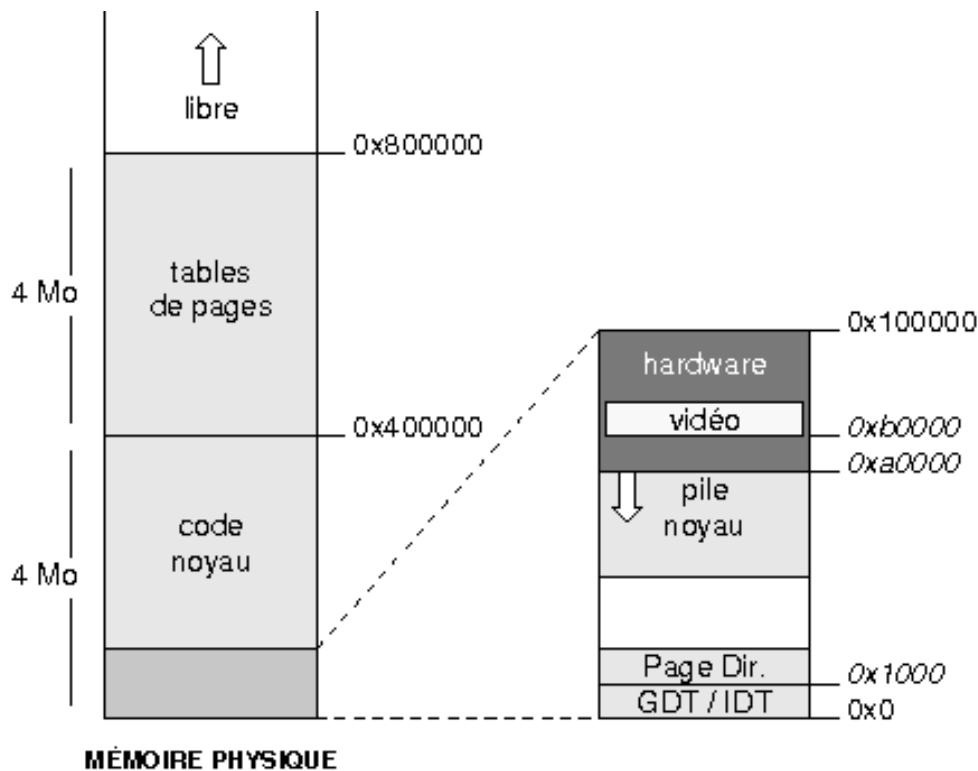
XVII-A-2 - Utilisation de la mémoire physique

Le noyau connaît la taille de la mémoire physique disponible grâce à **Grub**.

Dans l'implémentation de Pépin, les 8 premiers mégaoctets de mémoire physique sont réservés à l'usage du noyau et contiennent :

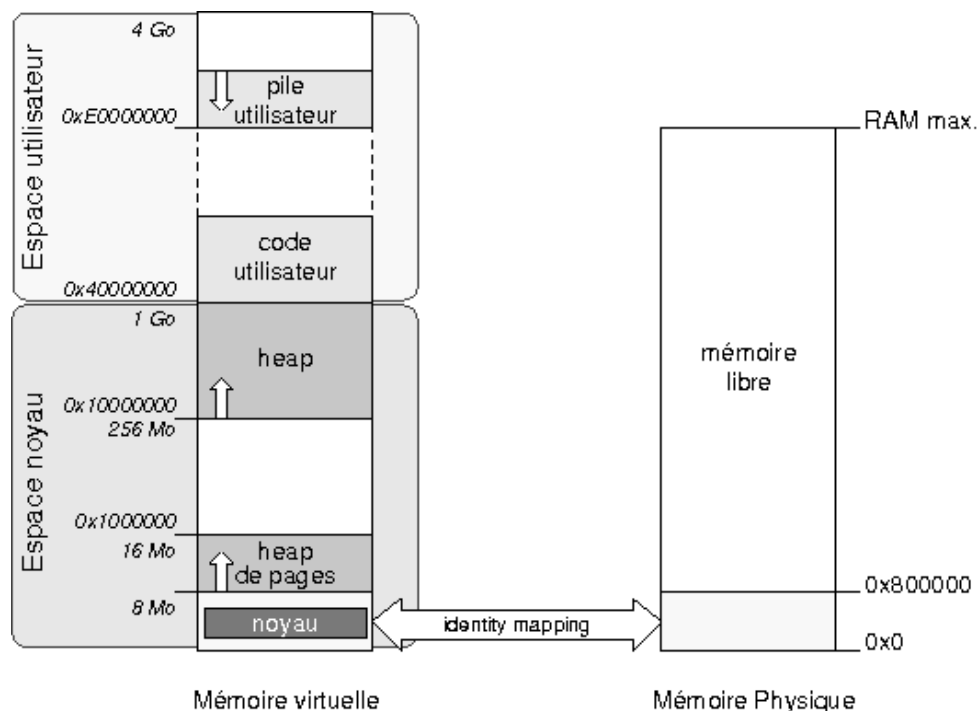
- le noyau ;
- les structures élémentaires (GDT, IDT et TSS) ;
- la pile noyau ;
- une partie réservée au hardware ;
- le répertoire de pages et les tables de pages du noyau.

Le reste de la mémoire physique est librement disponible pour le noyau et les applications :



XVII-A-3 - Utilisation de la mémoire virtuelle

L'espace d'adressage compris entre le début de la mémoire et l'adresse 0x40000000 correspond à l'espace du noyau alors que l'espace compris entre l'adresse 0x40000000 et la fin de la mémoire correspond à l'espace utilisateur :



L'espace d'adressage noyau, qui occupe 1 Gigaoctet de mémoire virtuelle, est commun à toutes les tâches (cette notion est essentielle ! Elle est expliquée en détail dans la partie traitant du déroulement d'une **tâche dans une espace paginé**). Ceci est implémenté très simplement en faisant pointer les 256 premières entrées du répertoire de pages de la tâche sur le répertoire de pages du noyau :

```
/*
 * Espace noyau. Les v_addr < USER_OFFSET sont adressées par la table
 * de pages du noyau.
 */
for (i=0; i<256; i++)
    pdir[i] = pd0[i];
```

Les 8 premiers mégaoctets de mémoire virtuelle et de mémoire physique sont mappés à l'identique. Les raisons ont déjà été expliquées dans les chapitres précédents.

XVII-B - Allouer dynamiquement de la mémoire pour le noyau

XVII-B-1 - Gérer la mémoire physique

La gestion de la mémoire physique est basée sur l'utilisation d'un bitmap qui contient le statut des différentes pages. Les fonctions qui permettent de manipuler ce bitmap sont les suivantes :

- `get_page_frame()` renvoie l'adresse d'une page libre et la marque comme utilisée. Cette fonction est utilisée quand le noyau a besoin d'une page physique libre ;
- `release_page_frame()` libère une page précédemment utilisée ;
- `set_page_frame_used()` marque une page comme étant utilisée.

XVII-B-2 - Gérer la mémoire virtuelle

L'utilisation de la pagination nécessite qu'à chaque adresse physique soit associée une adresse virtuelle, puisque c'est celle-ci qui sera directement manipulée par les instructions du processeur. Pour pouvoir faire cette association,

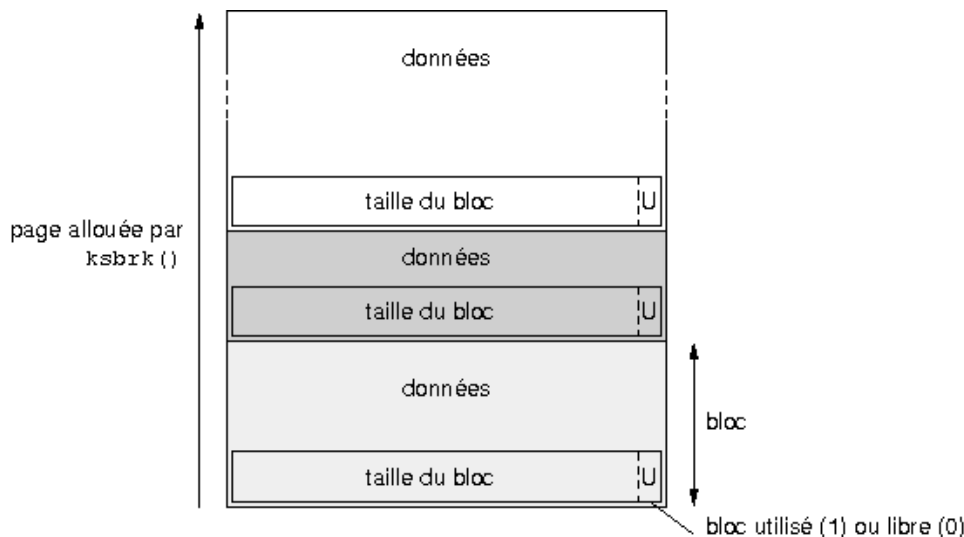
il faut gérer les adresses virtuelles. Deux gestionnaires d'adresses virtuelles, répondant à des besoins différents, sont utilisés par Pépin.

XVII-B-2-a - Le heap

La première fonction d'allocation de mémoire pour le noyau s'apparente à la fonction `malloc()` de la bibliothèque C standard. Il s'agit de la fonction `kmalloc()` qui permet d'allouer au noyau un nombre arbitraire d'octets. La fonction `kfree()` s'apparente à la fonction `free()` de la bibliothèque standard et libère une zone de mémoire précédemment allouée par `kmalloc()`.

Le **Heap** est une zone extensible, segmentée en blocs de données, et dans laquelle vient piocher `kmalloc()`. Le bloc d'octets disponibles pris dans le heap par `kmalloc()` est organisé tel que dans le schéma ci-dessous avec un en-tête et une zone de données. L'en-tête occupe 4 octets :

- 31 bits sont utilisés pour préciser la taille du bloc ;
- 1 bit est utilisé pour indiquer si le bloc est libre ou non.



Quand `kmalloc()` fait une demande de mémoire, le noyau parcourt le heap afin de trouver un bloc libre suffisamment grand. Si un tel bloc existe, la fonction renvoie l'adresse de la zone de données de ce bloc. Si le bloc trouvé est trop grand, celui-ci est scindé en deux et un second bloc libre est créé. Si le heap ne contient pas de bloc suffisamment grand, la fonction `ksbrk()` est appelée par `kmalloc()` pour étendre le heap.

À noter qu'au démarrage, le **Heap** est initialisé et il comprend un seul « bloc ».

Dans le détail, la fonction `ksbrk()` étend le **heap** en ajoutant à son espace d'adressage une page virtuelle. Parallèlement à ça, la fonction demande une page de mémoire physique avec `get_page_frame()` puis elle associe la nouvelle adresse virtuelle à cette page en mémoire physique en mettant à jour le répertoire de pages.

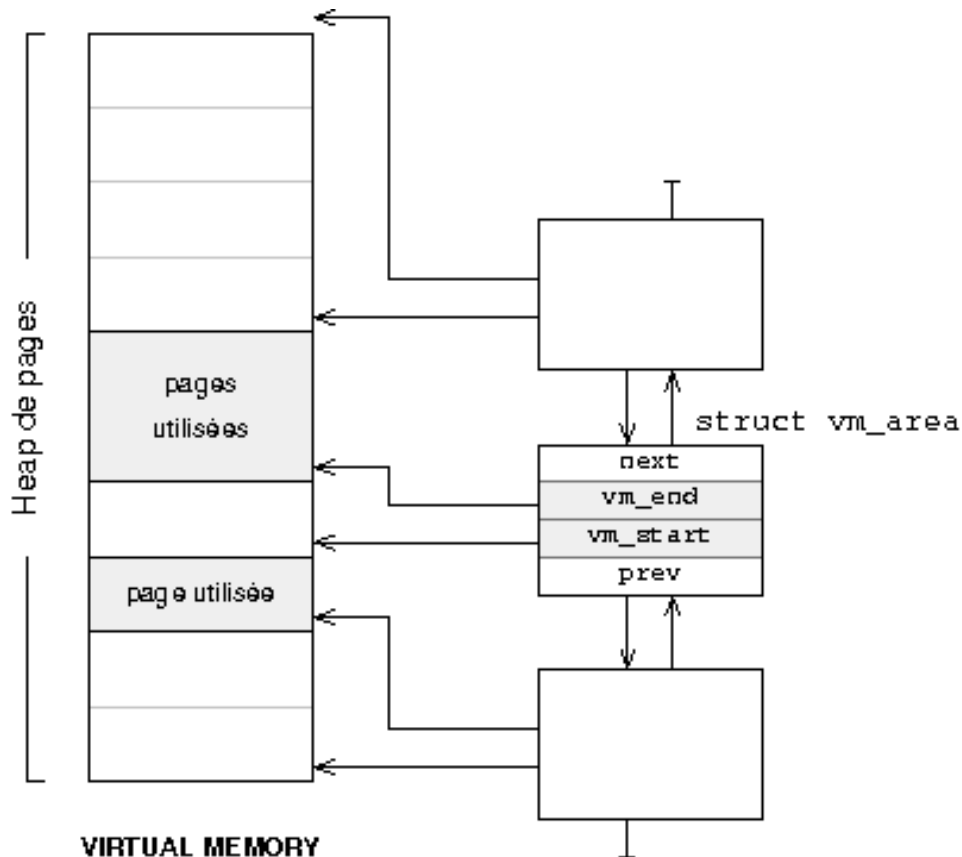


Toutes ces fonctions peuvent être testées dans l'espace utilisateur sous Unix avant d'être implémentées au niveau du noyau. Cela simplifie grandement le développement !

XVII-B-2-b - Le heap de pages

La fonction `kmalloc()` répond à l'essentiel des besoins d'allocation mémoire du noyau. Il y a cependant un cas particulier d'allocation non supportée par cette fonction, c'est quand le noyau a besoin d'une page alignée sur 4096 octets, par exemple pour créer une nouvelle table de pages.

Pour répondre à ce besoin, le noyau utilise un entrepôt de données particulier : le **heap de pages**. Le **Heap de pages** est une zone d'environ 256 Mo au sein de laquelle l'allocation se fait par page (de 4096 octets). L'ensemble des pages disponibles aurait pu être implémenté à l'aide d'une liste chaînée, mais j'ai préféré définir une liste des « zones » libres, une zone représentant ici une simple plage mémoire :



Les fonctions de manipulation du Heap de pages sont :

- `get_page_from_heap()`, qui renvoie une page libre ;
- `release_page_from_heap()`, qui libère une page utilisée.

XVII-B-2-c - Quand le noyau n'a plus assez de mémoire

Nous avons donc vu que le noyau utilise deux fonctions, `kmalloc()` et `get_page_from_heap()`, pour avoir de la mémoire. Mais que se passe-t-il quand il n'y a plus assez de mémoire physique ou virtuelle ? Dans une implémentation propre :

- ces deux fonctions doivent renvoyer une valeur pour signifier un échec ;
- chaque appel de ces deux fonctions doit tester le code de retour pour sortir proprement.

J'ai fait un autre choix pour Pépin : en cas d'échec, ces fonctions affichent un message sur la console et stoppent le système ! C'est assez radical, certes, mais cela permet de conserver au code de Pépin un maximum de simplicité et de concision. Gardez cependant à l'esprit que cela n'est pas une bonne pratique de la programmation !

XVII-C - Mettre à jour le répertoire et les tables de pages

Comment modifier une entrée du répertoire de pages courant ou d'une table de pages de ce dernier ? Cette question peut sembler assez étrange, mais elle cache en réalité un vrai problème que je vais essayer d'exposer brièvement.

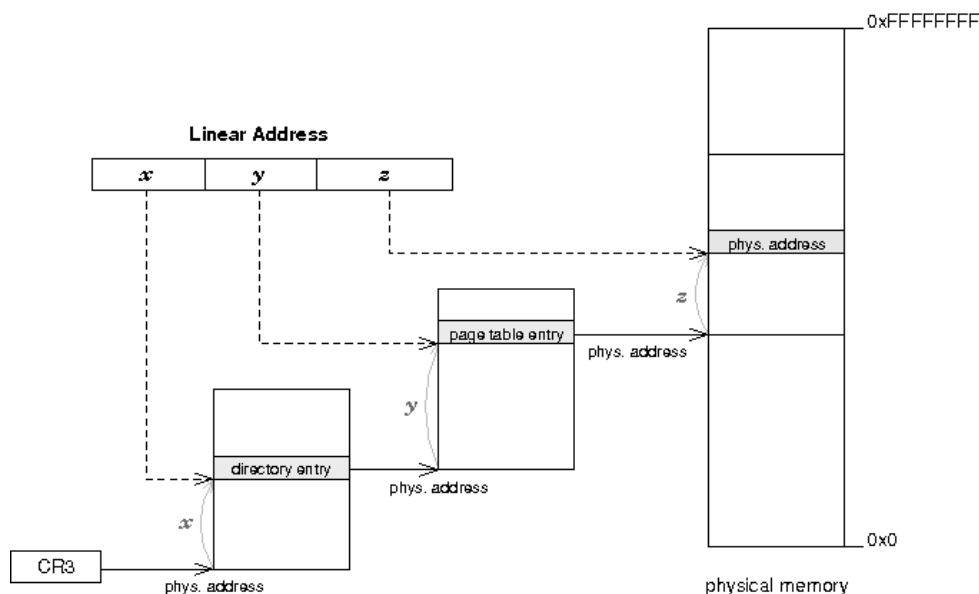
Nous savons qu'avec la pagination, toutes les adresses utilisées sont virtuelles. Cela rend difficile la manipulation directe d'une adresse physique. Supposons par exemple que je veuille modifier une donnée située à l'adresse physique **@p**. Je ne peux directement utiliser cette adresse ! Il faut nécessairement que je passe par une adresse virtuelle **@v** associée à **@p** grâce au répertoire de pages. On ne peut utiliser directement une adresse physique !

Supposons maintenant que je veuille mettre à jour le répertoire de pages courant, ce qui est une opération assez fréquente du noyau. La mise à jour du répertoire en tant que tel ne devrait pas poser trop de problèmes. En revanche, cette mise à jour suppose la plupart du temps celle d'une table de pages. Nous savons que le répertoire de pages contient les adresses des tables de pages. Jusque là, tout à l'air très simple, mais il y a en réalité un énorme problème... le répertoire contient les adresses physiques des tables de pages ! Dans ce cas, comment accéder à une table de pages pour la modifier ?

Plusieurs solutions existent. L'une d'elles est d'associer à chaque répertoire de pages une structure ou un tableau contenant les adresses virtuelles de ses tables de pages. L'inconvénient de cette solution est qu'elle nécessite la mise à jour d'une structure pour chaque répertoire de pages. Une autre solution est de réserver la dernière entrée du répertoire de pages en la faisant pointer sur le répertoire de pages lui-même. Cela semble à première vue assez étrange, mais cette astuce permet en fait de manipuler le répertoire ou les tables de pages très facilement. C'est cette solution qui est exposée ci-dessous.

XVII-C-1 - Rappel sur le mécanisme de pagination

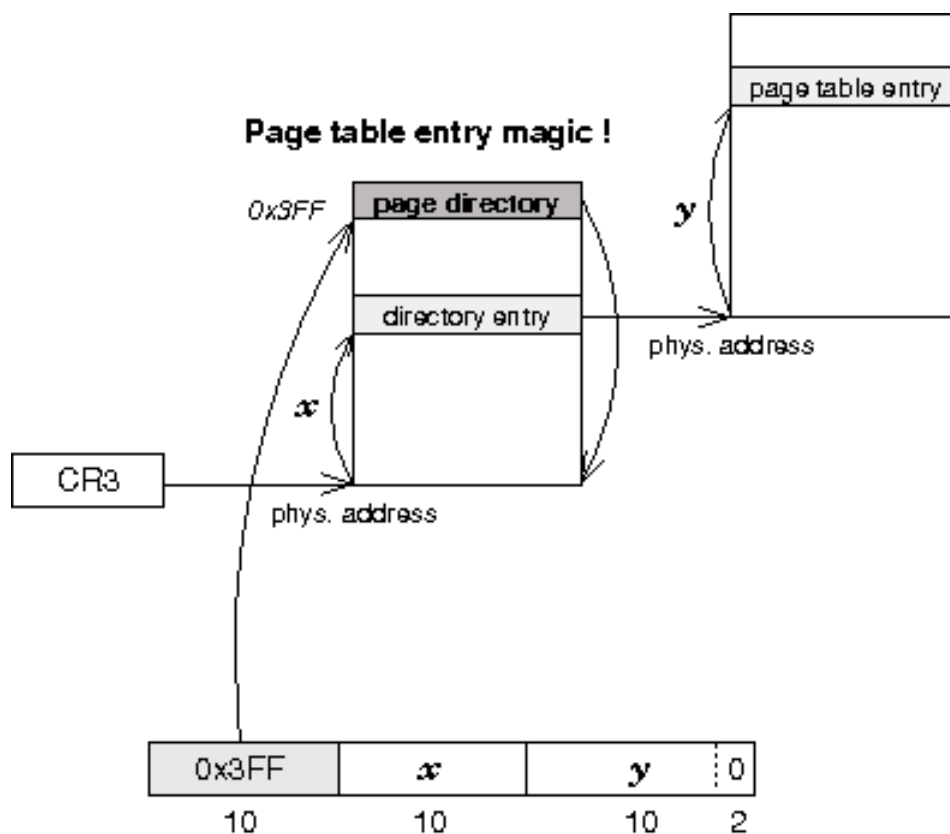
Le mécanisme utilisé par la pagination pour traduire une adresse linéaire en une adresse physique a déjà été vu. Cette adresse se décompose en trois index (ici **x**, **y** et **z**) :



XVII-C-2 - Modifier les tables de pages

Admettons que nous voulions modifier l'entrée de la table de pages qui sert l'adresse virtuelle **x.y.z**. Grâce à la dernière entrée du répertoire de pages qui est initialisée de façon à pointer sur le répertoire de pages lui-même, l'utilisation de l'adresse **0x3FF.x.y** permet de modifier l'entrée proprement dite.

L'utilisation de 0x3FF en début d'adresse fait que le répertoire de pages va être utilisé comme si c'était une table de pages. Ensuite, la **x**e entrée pointe sur la table de pages et la **y**e entrée pointe sur l'endroit à modifier proprement dit :

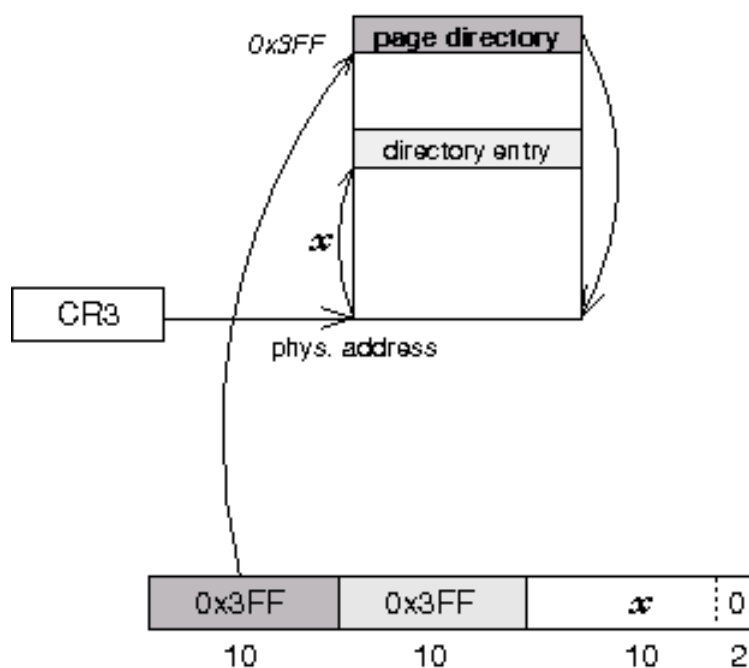


Ce schéma et cette explication ne sont pas forcément très facile à comprendre en raison de la complexité même du système de pagination. Pour que tout devienne plus clair, je vous conseille de faire mentalement plusieurs fois le cheminement conduisant à la traduction d'adresse en regardant bien les schémas précédents.

XVII-C-3 - Modifier le répertoire

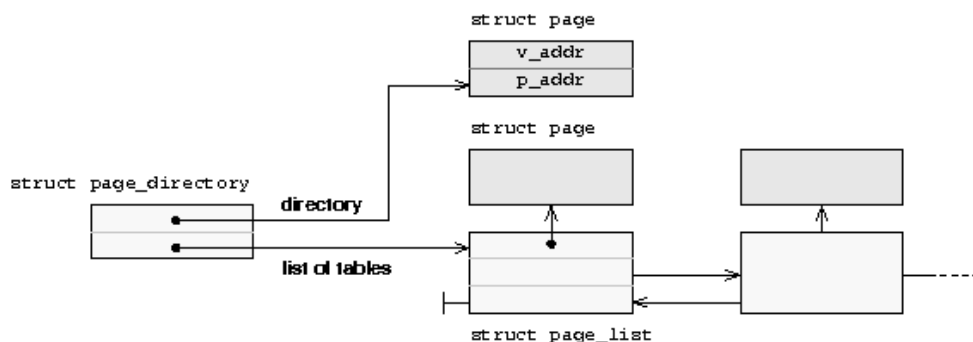
Pour modifier l'entrée du répertoire de pages correspondant à l'adresse virtuelle **x.y.z**, il faut utiliser l'adresse **0x3FF.0x3FF.x**. La logique est la même que précédemment avec juste un niveau supplémentaire de récursion :

Page table entry magic !



XVII-D - Compléments sur les répertoires de pages

Lors de la création d'une tâche utilisateur, un répertoire de pages dédié est créé. Nous avons vu qu'un répertoire peut être accompagné de 1024 tables de pages de 4 Mo chacune. Afin d'optimiser l'espace utilisé en mémoire, les tables sont seulement créées en cas de besoin, lors de la mise à jour d'une entrée du répertoire. La structure `struct page_directory` conserve la trace de ces allocations afin de pouvoir tout supprimer proprement quand une tâche se termine :

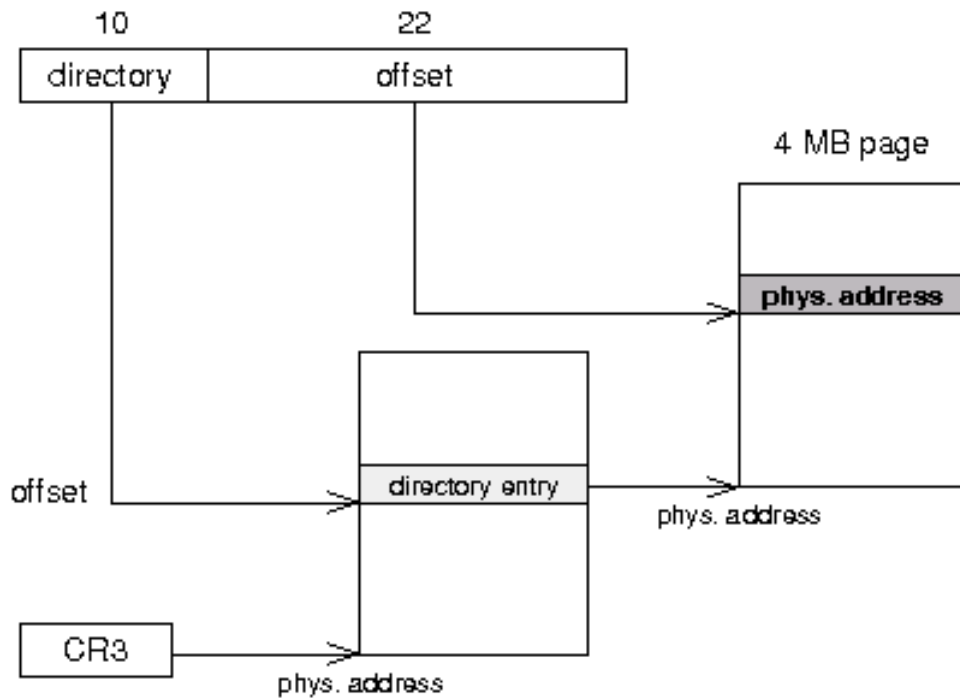


XVII-E - Compléments sur la pagination

XVII-E-1 - Utiliser des pages de 4 Mo

La pagination permet de gérer des pages de 4 Mo. Dans ce cas, l'entrée du répertoire de pages pointe directement sur la page allouée (sans passer par une table de pages). Ce mode d'adressage nécessite :

- l'activation du bit 4 du registre CR4 (Page Size Extension) ;
- le flag PS de l'entrée dans le répertoire de pages à 1.



Les entrées du répertoire de pages sont au format suivant :



- 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** est à 1 pour spécifier une taille de pages de 4 Mo
- le bit **G** est lié à la gestion du cache des pages
- le champ **Avail.** est librement utilisable

XVII-E-2 - Mettre à jour le TLB

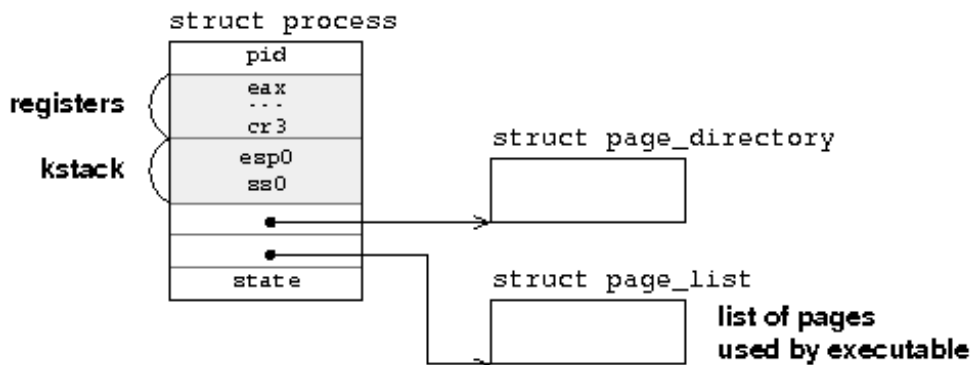
Quand la pagination est activée, la MMU se base sur le répertoire et les tables de pages pour traduire les adresses virtuelles en adresses physiques. Un cache, le Translation lookaside buffer (**TLB**) est utilisé pour augmenter la vitesse de traduction des adresses. Mais quand une entrée de répertoire ou d'une table de pages est modifiée, le cache n'est pas automatiquement mis à jour. Par conséquent, si la MMU continue d'utiliser le cache pour l'entrée en question, le résultat de la traduction d'adresses sera erroné. La mise à jour du cache doit donc être forcée par l'utilisateur et nous disposons pour cela de plusieurs moyens :

- recharger le registre CR3 réinitialise le cache ;
- l'instruction assembleur `invlpg` avec en paramètre une adresse virtuelle réinitialise l'entrée associée du TLB.

```
asm("invlpg %0"::"m"(v_addr));
```

XVII-F - Structure d'une tâche

La structure struct process contient toutes les informations associées à une tâche. Outre les registres du processeur et le pid, cette structure permet de garder une trace des pages allouées pour cette tâche :



XVIII - Lire et écrire sur un disque IDE

- **Écrire et lire sur un disque IDE avec les PIO**
- **Créer et utiliser une image d'un disque dur sous Unix**
- **Tester**

Les sources

Le package contenant les sources est téléchargeable ici : [kernel_Disk.tgz](#)

XVIII-A - Écrire et lire sur un disque IDE avec les PIO

Il existe deux méthodes possibles pour écrire et lire sur une unité de disque IDE : l'utilisation des ports du processeur (**PIO**) et l'utilisation du DMA. L'implémentation décrite ici utilise les ports du processeur. Les accès disque sont assez lents avec cette méthode, mais avec l'avantage d'une implémentation très simple :

```

#include "types.h"
#include "io.h"

int bl_common(int drive, int numblock, int count)
{
    outb(0x1F1, 0x00); /* NULL byte to port 0x1F1 */
    outb(0x1F2, count); /* Sector count */
    outb(0x1F3, (unsigned char) numblock); /* Low 8 bits of the block address */
    outb(0x1F4, (unsigned char) (numblock >> 8)); /* Next 8 bits of the block address */
    outb(0x1F5, (unsigned char) (numblock >> 16)); /* Next 8 bits of the block address */

    /* Drive indicator, magic bits, and highest 4 bits of the block address */
    outb(0x1F6, 0xE0 | (drive << 4) | ((numblock >> 24) & 0x0F));

    return 0;
}

int bl_read(int drive, int numblock, int count, char *buf)
{
    u16 tmpword;
    int idx;

    bl_common(drive, numblock, count);
    outb(0x1F7, 0x20);

    /* Wait for the drive to signal that it's ready: */
  
```

```

    while (!(inb(0x1F7) & 0x08));

    for (idx = 0; idx < 256 * count; idx++) {
        tmpword = inw(0x1F0);
        buf[idx * 2] = (unsigned char) tmpword;
        buf[idx * 2 + 1] = (unsigned char) (tmpword >> 8);
    }

    return count;
}

int bl_write(int drive, int numblock, int count, char *buf)
{
    ul6 tmpword;
    int idx;

    bl_common(drive, numblock, count);
    outb(0x1F7, 0x30);

    /* Wait for the drive to signal that it's ready: */
    while (!(inb(0x1F7) & 0x08));

    for (idx = 0; idx < 256 * count; idx++) {
        tmpword = (buf[idx * 2 + 1] << 8) | buf[idx * 2];
        outw(0x1F0, tmpword);
    }

    return count;
}

```

XVIII-B - Créer et utiliser une image d'un disque dur sous Unix

Il faut créer une image d'un disque dur, par exemple en utilisant la commande `bximage` :

```

bximage
> hd
> flat
> 2
> c.img

```

Ensuite, il faut ajouter dans le fichier de configuration de Bochs une entrée sur ce modèle :

```

ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
ata0-master: type=disk, path="c.img", mode=flat, cylinders=4, heads=16, spt=63

```

XVIII-C - Tester

Le code ci-dessous inscrit une chaîne de caractères sur l'image de disque à l'offset 1024 (bloc numéro 2) avec la fonction `bl_write()` :

```

char *buf, *msg = "Hello world\n";
buf = (char *) kmalloc(512);
memcpy((char *) buf, (char *) msg, strlen(msg) + 1);

bl_write(0, 2, 1, buf);

```

Pour vérifier que l'écriture a bien eu lieu :

```

od -A d -c c.img
0000000  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0001024  H   e   l   l   o           w   o   r   l   d  \n  \0  \0  \0  \0
0001040  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*

```

2064384

Pour lire cette inscription, on procède de façon similaire avec la fonction `bl_read()` :

```
bl_read(0, 2, 1, buf);
printk("buf: %s\n", buf);
```

XIX - Utiliser un système de fichiers Ext2FS

- [Description d'un système de fichiers Ext2FS](#)
- [Lecture dans un système de fichiers Ext2FS](#)
- [Créer et utiliser une image d'un disque dur sous Unix avec un système de fichiers Ext2FS](#)

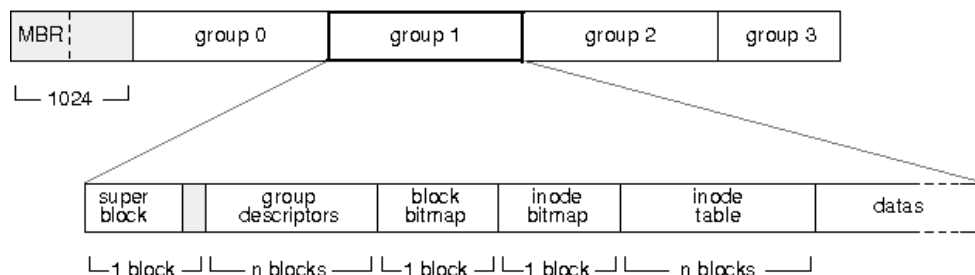
Les sources

Le package contenant les sources est téléchargeable ici : [kernel_Ext2Read.tgz](#)

XIX-A - Description d'un système de fichiers Ext2FS

XIX-A-1 - Un disque subdivisé en groupes

Quand on dépose sur un disque un système de fichiers de type Ext2, on inscrit sur celui-ci une structure particulière afin de stocker et d'organiser des données. La première partie du disque, de 1024 octets, est réservée au secteur de boot. Le reste du disque est divisé en **groupes**. Chaque groupe contient un en-tête et une zone pour les données :



L'en-tête de chaque groupe contient notamment les informations fondamentales décrivant la structure globale du système de fichiers. Chaque groupe possède une copie de ces informations :

- le **superbloc** contient les informations générales décrivant l'organisation logique du disque ;
- l'espace **group descriptors** est un tableau contenant l'ensemble des **descripteurs de groupes**. Pour chaque groupe, un descripteur contient les informations décrivant l'organisation spécifique des données en son sein.

Le reste de l'en-tête contient les structures décrivant l'organisation des données spécifiques à ce groupe :

- le **bloc bitmap** indique pour chaque bloc de ce groupe s'il est libre ou utilisé ;
- le bloc **inode bitmap** indique pour chaque inode de ce groupe si elle est libre ou utilisée ;
- l'espace **inode table** contient la liste des inodes de ce groupe.

XIX-B - Lecture dans un système de fichiers Ext2FS

Au préalable, le noyau doit récupérer les informations décrivant l'organisation globale des données sur le disque. Ces données sont consignées dans le superbloc et dans le tableau de descripteurs de groupes. Ensuite, en se servant

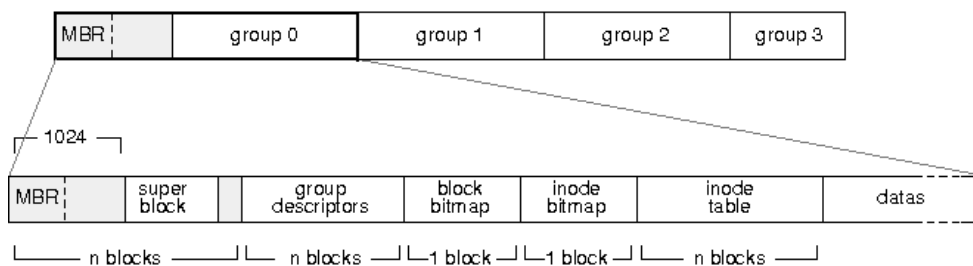
de son numéro d'inode, il n'est pas si difficile que ça de retrouver les informations relatives à un fichier et d'en lire le contenu.



Les fonctions de manipulation du système de fichiers peuvent être testées dans l'espace utilisateur sous Unix avant d'être implémentées au niveau du noyau. Pour ne pas risquer de corrompre un disque logique utilisé par le système, il est plus prudent de créer un disque virtuel dans un fichier qui servira pour l'occasion.

XIX-B-1 - Le superbloc

Le superbloc contient les informations relatives à l'organisation globale du système de fichiers, sa structure est détaillée dans le fichier ext2.h. Le superbloc fait une taille de 1024 octets. Si la taille des blocs est supérieure à 1024 ko, du padding est ajouté afin que l'espace **group descriptors** soit aligné sur cette taille de bloc. L'offset de la zone **group descriptor** dépend de la taille des blocs :



```
#include "types.h"

struct ext2_super_block {
    u32 s_inodes_count; /* Total number of inodes */
    u32 s_blocks_count; /* Total number of blocks */
    u32 s_r_blocks_count; /* Total number of blocks reserved for the super user */
    u32 s_free_blocks_count; /* Total number of free blocks */
    u32 s_free_inodes_count; /* Total number of free inodes */
    u32 s_first_data_block; /* Id of the block containing the superblock structure */
    u32 s_log_block_size; /* Used to compute block size = 1024 << s_log_block_size */
    u32 s_log_frag_size; /* Used to compute fragment size */
    u32 s_blocks_per_group; /* Total number of blocks per group */
    u32 s_frags_per_group; /* Total number of fragments per group */
    u32 s_inodes_per_group; /* Total number of inodes per group */
    u32 s_mtime; /* Last time the file system was mounted */
    u32 s_wtime; /* Last write access to the file system */
    u16 s_mnt_count; /* How many 'mount' since the last was full verification */
    u16 s_max_mnt_count; /* Max count between mount */
    u16 s_magic; /* = 0xEF53 */
    u16 s_state; /* File system state */
    u16 s_errors; /* Behaviour when detecting errors */
    u16 s_minor_rev_level; /* Minor revision level */
    u32 s_lastcheck; /* Last check */
    u32 s_checkinterval; /* Max. time between checks */
    u32 s_creator_os; /* = 5 */
    u32 s_rev_level; /* = 1, Revision level */
    u16 s_def_resuid; /* Default uid for reserved blocks */
    u16 s_def_resgid; /* Default gid for reserved blocks */
    u32 s_first_ino; /* First inode useable for standard files */
    u16 s_inode_size; /* Inode size */
    u16 s_block_group_nr; /* Block group hosting this superblock structure */
    u32 s_feature_compat;
    u32 s_feature_incompat;
    u32 s_feature_ro_compat;
    u8 s_uuid[16]; /* Volume id */
    char s_volume_name[16]; /* Volume name */
    char s_last_mounted[64]; /* Path where the file system was last mounted */
    u32 s_algo_bitmap; /* For compression */
}
```

```

    u8 s_padding[820];
} __attribute__((packed));

struct disk {
    int device;
    struct ext2_super_block *sb;
    u32 blocksize;
    u16 groups; /* Total number of groups */
    struct ext2_group_desc *gd;
};

struct ext2_group_desc {
    u32 bg_block_bitmap; /* Id of the first block of the "block bitmap" */
    u32 bg_inode_bitmap; /* Id of the first block of the "inode bitmap" */
    u32 bg_inode_table; /* Id of the first block of the "inode table" */
    u16 bg_free_blocks_count; /* Total number of free blocks */
    u16 bg_free_inodes_count; /* Total number of free inodes */
    u16 bg_used_dirs_count; /* Number of inodes allocated to directories */
    u16 bg_pad; /* Padding the structure on a 32bit boundary */
    u32 bg_reserved[3]; /* Future implementation */
} __attribute__((packed));

struct ext2_inode {
    u16 i_mode; /* File type + access rights */
    u16 i_uid;
    u32 i_size;
    u32 i_atime;
    u32 i_ctime;
    u32 i_mtime;
    u32 i_dtime;
    u16 i_gid;
    u16 i_links_count;
    u32 i_blocks; /* 512 bytes blocks ! */
    u32 i_flags;
    u32 i_osd1;

    /*
     * [0] -> [11] : block number (32 bits per block)
     * [12] : indirect block number
     * [13] : bi-indirect block number
     * [14] : tri-indirect block number
     */
    u32 i_block[15];

    u32 i_generation;
    u32 i_file_acl;
    u32 i_dir_acl;
    u32 i_faddr;
    u8 i_osd2[12];
} __attribute__((packed));

struct directory_entry {
    u32 inode; /* inode number or 0 (unused) */
    u16 rec_len; /* offset to the next dir. entry */
    u8 name_len; /* name length */
    u8 file_type;
    char name;
} __attribute__((packed));

/* super_block: s_errors */
#define EXT2_ERRORS_CONTINUE 1
#define EXT2_ERRORS_RO 2
#define EXT2_ERRORS_PANIC 3
#define EXT2_ERRORS_DEFAULT 1

/* inode: i_mode */
#define EXT2_S_IFMT 0xF000 /* format mask */
#define EXT2_S_IFSOCK 0xC000 /* socket */
#define EXT2_S_IFLNK 0xA000 /* symbolic link */
#define EXT2_S_IFREG 0x8000 /* regular file */
#define EXT2_S_IFBLK 0x6000 /* block device */

```

```
#define EXT2_S_IFDIR    0x4000 /* directory */
#define EXT2_S_IFCHR    0x2000 /* character device */
#define EXT2_S_IFIFO    0x1000 /* fifo */

#define EXT2_S_ISUID    0x0800 /* SUID */
#define EXT2_S_ISGID    0x0400 /* SGID */
#define EXT2_S_ISVTX    0x0200 /* sticky bit */
#define EXT2_S_IRWXU    0x01C0 /* user access rights mask */
#define EXT2_S_IRUSR    0x0100 /* read */
#define EXT2_S_IWUSR    0x0080 /* write */
#define EXT2_S_IXUSR    0x0040 /* execute */
#define EXT2_S_IRWXG    0x0038 /* group access rights mask */
#define EXT2_S_IRGRP    0x0020 /* read */
#define EXT2_S_IWGRP    0x0010 /* write */
#define EXT2_S_IXGRP    0x0008 /* execute */
#define EXT2_S_IRWXO    0x0007 /* others access rights mask */
#define EXT2_S_IROTH    0x0004 /* read */
#define EXT2_S_IWOTH    0x0002 /* write */
#define EXT2_S_IXOTH    0x0001 /* execute */

struct disk *ext2_get_disk_info(int);
struct ext2_super_block *ext2_read_sb(int);
struct ext2_group_desc *ext2_read_gd(struct disk *);

struct ext2_inode *ext2_read_inode(struct disk *, int);
char *ext2_read_file(struct disk *, struct ext2_inode *);
```

À partir des informations contenues dans le superbloc, on obtient des informations qui sont primordiales pour comprendre l'organisation globale du système de fichiers :

- la taille des blocs : $1024 \ll s_log_block_size$;
- la taille des inodes : s_inode_size ;
- le nombre de groupes, il faut prendre le maximum de :
 - $s_blocks_count / s_blocks_per_group + ((s_blocks_count \% s_blocks_per_group) ? 1 : 0)$,
 - $s_inodes_count / s_inodes_per_group + ((s_inodes_count \% s_inodes_per_group) ? 1 : 0)$.

La fonction `ext2_read_sb()` du fichier `ext2.c` lit le superbloc et calcule à partir de ces informations la taille des blocs et le nombre de groupes.

```
#include "ext2.h"
#include "disk.h"
#include "kmalloc.h"
#include "lib.h"

/*
 * Initialise la structure décrivant le disque logique.
 * Offset correspond au début de la partition.
 */
struct disk *ext2_get_disk_info(int device)
{
    int i, j;
    struct disk *hd;

    hd = (struct disk *) kmalloc(sizeof(struct disk));

    hd->device = device;
    hd->sb = ext2_read_sb(device);
    hd->blocksize = 1024 << hd->sb->s_log_block_size;

    i = (hd->sb->s_blocks_count / hd->sb->s_blocks_per_group) +
        ((hd->sb->s_blocks_count \% hd->sb->s_blocks_per_group) ? 1 : 0);
    j = (hd->sb->s_inodes_count / hd->sb->s_inodes_per_group) +
        ((hd->sb->s_inodes_count \% hd->sb->s_inodes_per_group) ? 1 : 0);
    hd->groups = (i > j) ? i : j;
```

```
    hd->gd = ext2_read_gd(hd);

    return hd;
}

struct ext2_super_block *ext2_read_sb(int device)
{
    struct ext2_super_block *sb;

    sb = (struct ext2_super_block *) kmalloc(sizeof(struct ext2_super_block));
    disk_read(device, 1024, (char *) sb, sizeof(struct ext2_super_block));

    return sb;
}

struct ext2_group_desc *ext2_read_gd(struct disk *hd)
{
    struct ext2_group_desc *gd;
    int offset, gd_size;

    /* localisation du bloc */
    offset = (hd->blocksize == 1024) ? 2048 : hd->blocksize;

    /* taille occupee par les descripteurs */
    gd_size = hd->groups * sizeof(struct ext2_group_desc);

    /* creation du tableau de descripteurs */
    gd = (struct ext2_group_desc *) kmalloc(gd_size);

    disk_read(hd->device, offset, (char *) gd, gd_size);

    return gd;
}

/* Retourne la structure d'inode à partir de son numéro */
struct ext2_inode *ext2_read_inode(struct disk *hd, int i_num)
{
    int gr_num, index, offset;
    struct ext2_inode *inode;

    inode = (struct ext2_inode *) kmalloc(sizeof(struct ext2_inode));

    /* groupe qui contient l'inode */
    gr_num = (i_num - 1) / hd->sb->s_inodes_per_group;

    /* index de l'inode dans le groupe */
    index = (i_num - 1) % hd->sb->s_inodes_per_group;

    /* offset de l'inode sur le disk */
    offset =
        hd->gd[gr_num].bg_inode_table * hd->blocksize + index * hd->sb->s_inode_size;

    /* lecture */
    disk_read(hd->device, offset, (char *) inode, hd->sb->s_inode_size);

    return inode;
}

char *ext2_read_file(struct disk *hd, struct ext2_inode *inode)
{
    char *mmap_base, *mmap_head, *buf;

    int *p, *pp, *ppp;
    int i, j, k;
    int n, size;

    buf = (char *) kmalloc(hd->blocksize);
    p = (int *) kmalloc(hd->blocksize);
    pp = (int *) kmalloc(hd->blocksize);
    ppp = (int *) kmalloc(hd->blocksize);
```

```

/* taille totale du fichier */
size = inode->i_size;
mmap_head = mmap_base = kcalloc(size);

/* direct block number */
for (i = 0; i < 12 && inode->i_block[i]; i++) {
    disk_read(hd->device, inode->i_block[i] * hd->blocksize, buf, hd->blocksize);

    n = ((size > hd->blocksize) ? hd->blocksize : size);
    memcpy(mmap_head, buf, n);
    mmap_head += n;
    size -= n;
}

/* indirect block number */
if (inode->i_block[12]) {
    disk_read(hd->device, inode->i_block[12] * hd->blocksize, (char *) p, hd-
>blocksize);

    for (i = 0; i < hd->blocksize / 4 && p[i]; i++) {
        disk_read(hd->device, p[i] * hd->blocksize, buf, hd->blocksize);

        n = ((size > hd->blocksize) ? hd->blocksize : size);
        memcpy(mmap_head, buf, n);
        mmap_head += n;
        size -= n;
    }
}

/* bi-indirect block number */
if (inode->i_block[13]) {
    disk_read(hd->device, inode->i_block[13] * hd->blocksize, (char *) p, hd-
>blocksize);

    for (i = 0; i < hd->blocksize / 4 && p[i]; i++) {
        disk_read(hd->device, p[i] * hd->blocksize, (char *) pp, hd->blocksize);

        for (j = 0; j < hd->blocksize / 4 && pp[j]; j++) {
            disk_read(hd->device, pp[j] * hd->blocksize, buf, hd->blocksize);

            n = ((size > hd->blocksize) ? hd->blocksize : size);
            memcpy(mmap_head, buf, n);
            mmap_head += n;
            size -= n;
        }
    }
}

/* tri-indirect block number */
if (inode->i_block[14]) {
    disk_read(hd->device, inode->i_block[14] * hd->blocksize, (char *) p, hd-
>blocksize);

    for (i = 0; i < hd->blocksize / 4 && p[i]; i++) {
        disk_read(hd->device, p[i] * hd->blocksize, (char *) pp, hd->blocksize);

        for (j = 0; j < hd->blocksize / 4 && pp[j]; j++) {
            disk_read(hd->device, pp[j] * hd->blocksize, (char *) ppp, hd-
>blocksize);

            for (k = 0; k < hd->blocksize / 4 && ppp[k]; k++) {
                disk_read(hd->device, ppp[k] * hd->blocksize, buf, hd-
>blocksize);

                n = ((size > hd->blocksize) ? hd->blocksize : size);
                memcpy(mmap_head, buf, n);
                mmap_head += n;
                size -= n;
            }
        }
    }
}

```

```
kfree(buf);
kfree(p);
kfree(pp);
kfree(ppp);

return mmap_base;
}
```

XIX-B-2 - Le bloc de descripteurs de groupes

Le disque est subdivisé en groupes. L'organisation de chaque groupe est décrite par un descripteur qui fournit les informations suivantes :

- le numéro de bloc où se situe le **block bitmap** : `bg_block_bitmap` ;
- le numéro de bloc où se situe le **inode bitmap** : `bg_inode_bitmap` ;
- le numéro de bloc où se situe la **table d'inodes** : `bg_inode_table`.

Notez que les offsets ci-dessus sont exprimés en numéro de bloc à partir du début du disque. L'offset en octet se calcule en multipliant ce numéro de bloc par la taille de bloc.

La fonction `ext2_read_gd()` crée un tableau contenant l'ensemble des descripteurs de groupes.

XIX-B-3 - Localiser sur le disque une inode

Une inode contient les informations de base d'un fichier. Une inode est localisable sur le disque à partir des calculs suivants :

- localiser le groupe où se trouve l'inode : $(i_num - 1) / s_inodes_per_group$;
- retrouver l'inode dans le groupe en calculant son index dans la table d'inodes : $(i_num - 1) \% s_inodes_per_group$.

La fonction `ext2_read_inode()` retrouve et renvoie une structure d'inode à partir de son numéro passé en argument.

XIX-B-4 - Lire un fichier

La fonction `ext2_read_file()` renvoie un buffer avec le contenu du fichier correspondant à l'inode passée en argument.

XIX-C - Créer et utiliser une image d'un disque dur sous Unix avec un système de fichier Ext2FS



Les commandes shell décrites ci-dessous doivent être parfaitement comprises avant d'être exécutées sur un système UNIX ! Si une commande ou un concept ne vous semble pas clair, je vous conseille de lire la documentation accessible sur l'excellent [Guide du Rootard](#). Même si cet effort vous semble futile et n'a en apparence pas de rapport avec le développement d'un noyau, je vous garantis que les connaissances acquises vous seront très utiles !

Il faut d'abord créer une image d'un disque dur avec la commande `bximage` :

```
bximage
> hd
> flat
> 2
> c.img
```

Pour créer un système de fichiers :

```
mke2fs c.img
```

Pour voir les caractéristiques du système :

```
dumpe2fs c.img
```

Pour monter le système de fichiers :

```
mount -o loop -t ext2 c.img /mnt/loop
```



Actuellement les systèmes **Ext3**, et **Ext4** sont plutôt utilisés. **Ext3** est complètement rétrocompatible avec **Ext2**, il y ajoute la **journalisation**. Une partition **ext3** peut être montée en **Ext3**. Une partition **Ext4** peut être montée en **Ext3** si l'allocation par **Extent** n'a jamais été utilisée.

XX - Créer et lancer une application au format ELF à partir du système de fichiers

- **Remarques**
- **Anatomie d'un fichier ELF**
- **Charger un exécutable**
- **Un nouvel appel système : exit()**
- **Créer et lancer une application au format ELF à partir du système de fichiers**

Les sources

Le package contenant les sources est téléchargeable ici : [kernel_ELF.tgz](#)

XX-A - Remarques

Dans ce chapitre, nous allons voir comment créer et charger un fichier exécutable au format ELF pour Pépin. Notez que pour le moment :

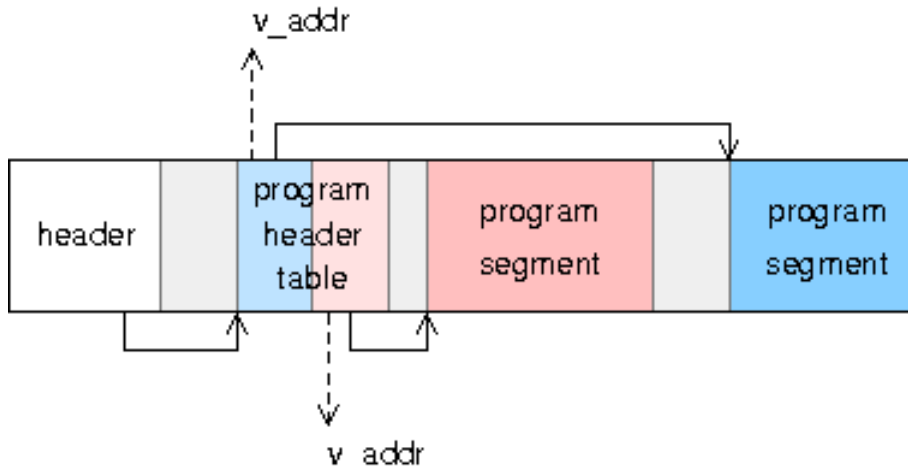
- les fichiers seront chargés à partir de leur numéro d'inode ;
- seuls les fichiers compilés en statique seront supportés ;
- il y a une disquette avec le noyau d'une part et un disque avec des fichiers d'autre part. Patience ! Nous verrons au chapitre suivant comment booter un noyau avec Grub directement depuis un disque IDE.

XX-B - Anatomie d'un fichier ELF

Un fichier au format **ELF** est formé notamment par :

- un premier en-tête qui décrit le type de fichier dont il est question et qui définit l'organisation globale au sein de ce fichier ;
- une table spéciale, la **Program Header Table**, qui décrit et localise les différents segments de code à charger par le noyau.

Le schéma ci-dessous est une illustration d'un fichier ELF comportant deux segments de code :



L'en-tête d'un fichier ELF est défini dans le fichier elf.h par la structure Elf32_Ehdr.

```
typedef struct {
    unsigned char e_ident[16]; /* ELF identification */
    u16 e_type; /* 2 (exec file) */
    u16 e_machine; /* 3 (intel architecture) */
    u32 e_version; /* 1 */
    u32 e_entry; /* starting point */
    u32 e_phoff; /* program header table offset */
    u32 e_shoff; /* section header table offset */
    u32 e_flags; /* various flags */
    u16 e_ehsize; /* ELF header (this) size */

    u16 e_phentsize; /* program header table entry size */
    u16 e_phnum; /* number of entries */

    u16 e_shentsize; /* section header table entry size */
    u16 e_shnum; /* number of entries */

    u16 e_shstrndx; /* index of the section name string table */
} Elf32_Ehdr;
```

- La variable `e_ident[]` sert à identifier le type de fichier et doit commencer par les caractères : 0x7f, 'E', 'L', 'F'.
- La variable `e_entry` définit le point d'entrée du programme et dans notre cas correspond à l'adresse virtuelle 0x40000000 (USER_OFFSET).
- La variable `e_phoff` indique où se situe la **Program Header Table** décrivant les segments de code.
- La variable `e_phnum` indique le nombre d'entrées dans cette table.

Les autres variables ne sont pas importantes pour le moment.

Chaque entrée de la **Program Header Table** est une structure qui donne les caractéristiques d'un segment dans le fichier et où le charger en mémoire. La structure correspondant à chacune de ces entrées est la suivante : Elf32_Phdr.

```
typedef struct {
    u32 p_type; /* type of segment */
    u32 p_offset;
    u32 p_vaddr;
    u32 p_paddr;
    u32 p_filesz;
    u32 p_memsz;
    u32 p_flags;
    u32 p_align;
} Elf32_Phdr;
```

- `p_offset` est l'offset où se situe le segment par rapport au début du fichier.
- `p_filesz` correspond à sa taille.

- `p_vaddr` est l'adresse virtuelle où il doit être chargé par le noyau.
- `p_memsz` correspond à sa taille en mémoire. Notez qu'il est tout à fait possible que la valeur de `p_memsz` soit supérieure à celle de `p_filesz`. Dans ce cas, le noyau doit simplement réserver l'espace suffisant et l'initialiser à 0.

Cette description d'un fichier ELF est très sommaire, car ce format est complexe et je n'ai voulu indiquer que ce qui est indispensable pour notre noyau. Pour en savoir plus, je vous conseille de parcourir la documentation.

XX-C - Charger un exécutable

L'algorithme utilisé par la fonction `load_elf()` pour charger un exécutable au format ELF est très simple :

```
Algorithme : load_elf
Paramètres : buffer contenant le fichier, adresse du répertoire de pages
Début
  Pour chaque entrée dans la table des segments
    Si le segment est de type PT_LOAD alors
      On charge le segment :
        /* p_vaddr et p_memsz permettent de savoir où le segment sera logé en mémoire */
        On contrôle que p_vaddr et p_vaddr + p_memsz sont valides
        Les pages nécessaires sont réservées en mémoire physique
        Le répertoire de pages du processus courant est mis à jour
        Le segment est recopié à l'endroit défini par p_vaddr
        Si p_memsz > p_filesz alors
          Les octets en plus sont mis à 0
Fin
```

En pratique, le fichier `kernel.c` contient le code permettant de charger un fichier exécutable. Pour le moment, on ne peut charger un exécutable que par son numéro d'inode. Celle-ci est chargée via la fonction `ext2_read_inode()`. Ensuite, la fonction `load_task()` charge l'exécutable :

```
inode = ext2_read_inode(hd, gd, 14);
load_task(hd, inode);
```

Ensuite, la fonction `load_task()` fait appel à la fonction `load_elf()` :

```
file = ext2_read_file(hd, inode);
load_elf(file, pd, mmap);
```

XX-D - Un nouvel appel système : `exit()`

Jusqu'à présent, notre noyau disposait d'un appel système permettant à une application d'afficher un message sur la console. Cette partie présente un nouvel appel système permettant à une tâche utilisateur de se terminer proprement. Cet appel est implémenté dans le fichier `syscalls.c`. L'algorithme est très simple :

```
Algorithme : sys_exit()
Synopsis : termine le processus courant
Début
  Désactive les interruptions
  Décrémente le nombre de processus
  Libère l'entrée dans la table des processus : current->state = 0
  Libère les pages mémoires occupées par le code (current->mmap)
  Libère la pile utilisateur (get_p_addr((char *) USER_STACK))
  Libère la pile noyau (current->kstack)
  Libère le répertoire et les tables de pages (current->pd)
  Choix d'une nouvelle tâche (ici la "tâche" noyau)
  Bascule sur la nouvelle tâche : switch_to_task()
Fin
```

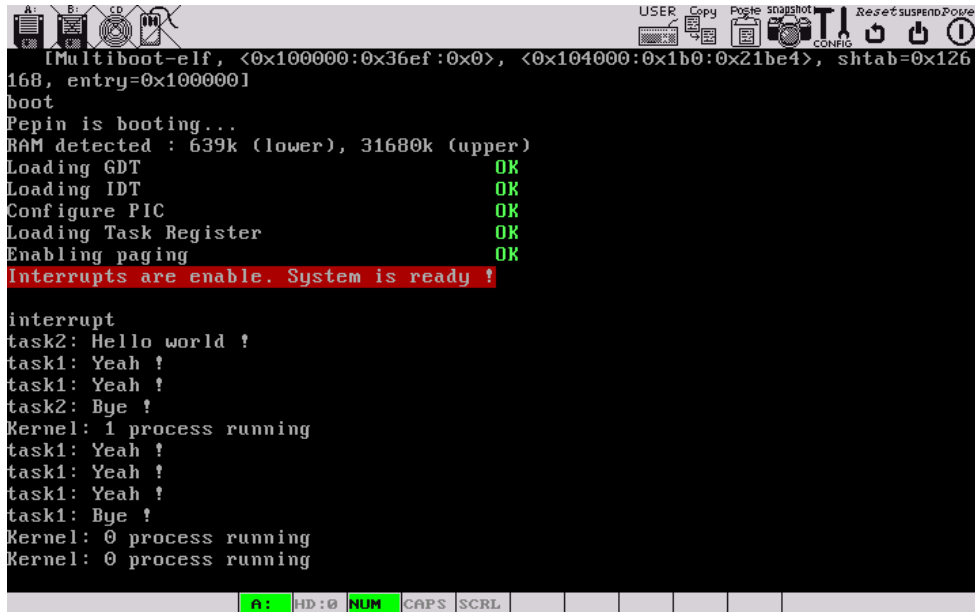
Pour utiliser ce nouvel appel système, l'application devra simplement insérer l'instruction suivante :

```
asm("mov $0x02, ?x; int $0x30");
```

XX-E - Créer et lancer une application au format ELF à partir du système de fichiers

Pour compiler une application au format ELF fonctionnant sous Pépin :

```
gcc -c task1.c
ld -Ttext=4000000 --entry=main task1.o
```



XXI - Booter avec Grub sur un disque IDE

- Créer une image d'un disque IDE partitionné
- Utiliser un disque logique

Les sources

Le package contenant les sources est téléchargeable ici : [kernel_Grub2.tgz](#)

XXI-A - Créer une image d'un disque IDE partitionné

Le fait de booter sur un disque IDE apporte un changement fondamental par rapport au boot sur une disquette : il faut partitionner le disque. Cela apporte une petite complexité supplémentaire dont le contournement est expliqué ci-dessous...

On commence par créer une image avec bxiimage :

```
bxiimage
> hd
> flat
> 2
> c.img
```

Cela a pour effet de créer un fichier nommé ici c.img et affiche le message suivant à l'écran :

```
The following line should appear in your bochssrc:
```

```
ata0-master: type=disk, path="c.img", mode=flat, cylinders=4, heads=16, spt=63
```

Cette ligne est à reporter dans le fichier `.bochsrc`. N'oubliez pas aussi de préciser le périphérique de boot :

```
boot: disk
```

Il faut ensuite partitionner l'image du disque. Les commandes `c`, `h` et `s` indiquent quelle est la géométrie du disque :

```
fdisk c.img
> x
> c 4
> h 16
> s 63
> r
> n, p, 1
> a 1
> w
```

Après ça, il faut obtenir le numéro de secteur de la partition créée :

```
fdisk -l -u c.img
Disk c.img: 0 MB, 0 bytes
16 heads, 63 sectors/track, 0 cylinders, total 0 sectors
Units = sectors of 1 * 512 = 512 bytes
Disk identifier: 0x7337b4ff

Device Boot      Start         End      Blocks   Id  System
c.img1    *           63         4031       1984+   83   Linux
```

Nous allons maintenant associer la partition créée à un **loop device**, qui est un pseudo périphérique qui permet à un fichier d'être accédé comme s'il s'agissait d'un périphérique de type bloc. On utilise pour cela la commande `losetup` avec comme argument l'offset, en octet, où se trouve le début de la partition (c'est la raison de la commande précédente). L'offset calculé est de $63 * 512 = 32356$:

```
losetup -o 32256 /dev/loop0 c.img
```

Pour détacher le loop device de l'image :



```
losetup -d /dev/loop0
```

À partir de maintenant, la partition de l'image disque est vue comme un vrai périphérique de type bloc. On peut donc déposer un système de fichiers dessus :

```
mke2fs /dev/loop0
```

Ensuite, pour monter la partition :

```
mount -t ext2 /dev/loop0 /mnt/loop
```

XXI-A-1 - Installer grub sur une image bootable

On ajoute quelques fichiers sur la partition montée :

```
mkdir /mnt/loop/grub
cp /boot/grub/stage* /mnt/loop/grub
cat > /mnt/loop/grub/menu.lst << EOF
title=Pepin
```

```
root (hd0,0)
kernel /kernel
boot
EOF
```

Ensuite, il faut démonter la partition pour installer grub dessus :

```
umount /mnt/loop
grub --device=map=/dev/null << EOF
device (hd0) c.img
root (hd0,0)
setup (hd0)
quit
EOF
```

Ouf ! C'est fait, maintenant, on peut remonter la partition quelque part pour copier le noyau dessus.

XXI-B - Utiliser un disque logique

La table des partitions primaires du **MBR** permet de gérer jusqu'à 4 partitions. Les informations sur ces partitions sont situées à des offsets prédéfinis par rapport au début du disque :

Partition	Offset
Partition 1	0x01BE
Partition 2	0x01CE
Partition 3	0x01DE
Partition 4	0x01EE

Chaque entrée de la table fait 16 octets et obéit au format suivant :

```
struct partition {
    u8    bootable;        /* 0 = no, 0x80 = bootable */
    u8    s_head;          /* Starting head */
    u16    s_sector : 6;    /* Starting sector */
    u16    s_cyl : 10;     /* Starting cylinder */
    u8    id;              /* System ID */
    u8    e_head;          /* Ending head */
    u16    e_sector : 6;    /* Ending sector */
    u16    e_cyl : 10;     /* Ending cylinder */
    u32    s_lba;          /* Starting sector (LBA value) */
    u32    size;           /* Total sector number */
} __attribute__((packed));
```



Sur les systèmes récents, utilisant **UEFI**, la table de partition MBR est remplacée par la table **GPT**. Il y a une forme de rétrocompatibilité, une table GPT contenant une « Protective MBR »

La valeur `s_lba` permet de connaître l'offset où commence une partition par rapport au début du disque. En l'occurrence, les données débiteront à partir de l'offset `s_lba*512` (en octets).

Nous avons vu au chapitre sur l'utilisation d'un **système de fichiers Ext2** comment lire des données telles que le superbloc, une inode, etc. Quelques changements sont à apporter pour que ces fonctions continuent de fonctionner, car les lectures ne se font plus à partir du début du disque, mais à partir du début de la partition. Chaque donnée subit donc un décalage de `s_lba*512` octets :

```
struct partition *p1;
struct ext2_super_block *sb;

p1 = (struct partition*) kmalloc(sizeof(struct partition));
```

```
disk_read(0, 0x01BE, (char*) p1, 16);

printk("partition 1\n start: %d, size: %d\n", p1->s_lba, p1->size);

sb = (struct ext2_super_block*) kmalloc(sizeof(struct ext2_super_block));
disk_read(0, (p1.s_lba * 512) + 1024, (char*) sb, sizeof(struct ext2_super_block));

printk("volume name: %s, block size: %d\n", sb->s_volume_name, (1024 << sb->s_log_block_size));
```

XXII - Quelques structures élémentaires pour gérer les fichiers

- **Organiser les fichiers**
- **De nouveaux appels système**

Les sources

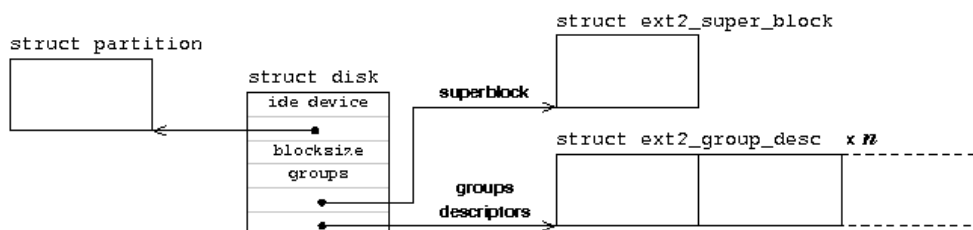
Le package contenant les sources est téléchargeable ici : [kernel_File.tgz](#)

XXII-A - Organiser les fichiers

Jusqu'à présent, nous avons vu comment implémenter au sein d'un noyau les fonctionnalités permettant de lire un fichier dans un système de fichiers de type Ext2FS à partir de son numéro d'inode. Cette partie présente les structures et les fonctions implémentées dans le noyau Pépin afin de pouvoir accéder aux fichiers directement par leur nom.

XXII-A-1 - Le disque logique

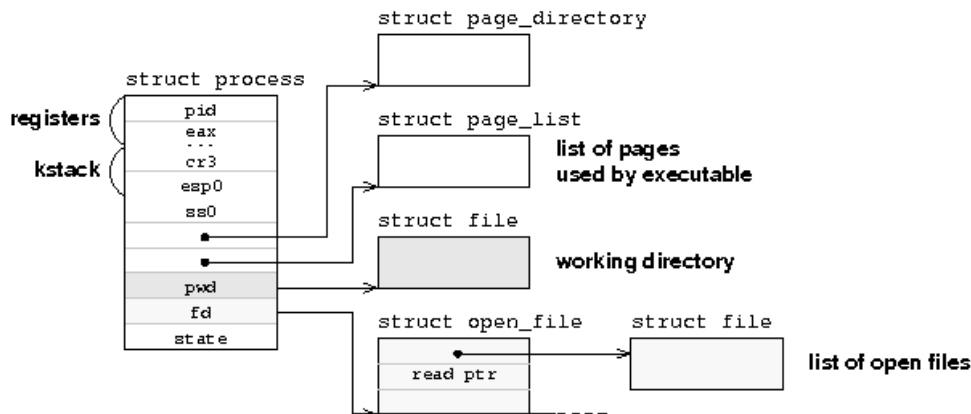
Les données générales relatives à la structure du disque logique et du système de fichiers sont collectées dans la structure struct disk :



XXII-A-2 - Structure d'un fichier et arborescence

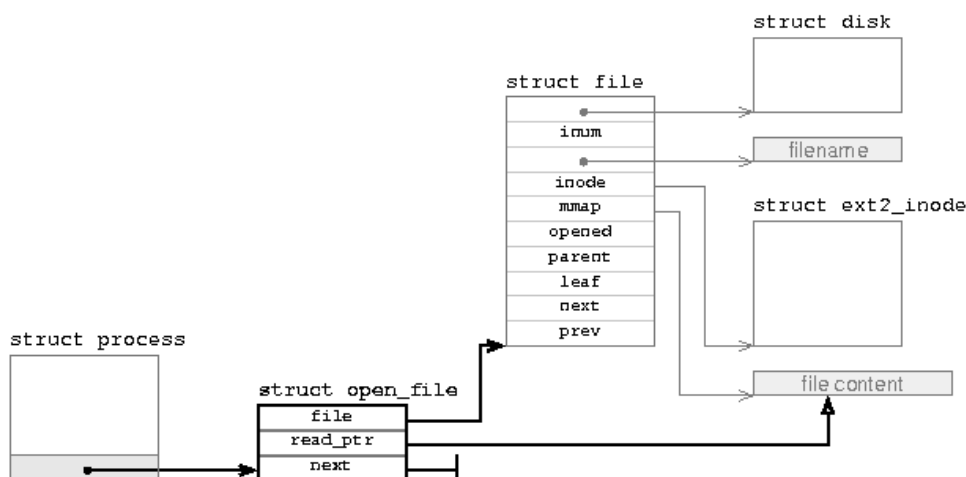
Les fichiers sont organisés sous forme d'arborescence. C'est une donnée très générale que je ne détaillerai pas ici. Afin d'économiser les accès disques, le noyau cache en mémoire une partie de cette arborescence. Chaque nœud de cette arborescence est décrit par la structure struct file :

- le répertoire courant, par le biais du champ `pwd` ;
- la liste des fichiers ouverts, par le biais d'une liste chaînée de descripteurs de fichiers



Le schéma ci-dessous détaille la structure `struct open_file` :

- le champ `file` pointe sur le fichier ouvert ;
- le champ `ptr` pointe sur la position courante dans le fichier. C'est un indicateur qui permet de savoir où en est la lecture de ce fichier par le processus ;
- le champ `next` permet de chaîner les structures dans le cas où plusieurs fichiers sont ouverts.



L'ouverture, la lecture et la fermeture de fichiers sont réalisées par trois nouveaux appels système : `sys_open()`, `sys_read()` et `sys_close()`. Attention, contrairement à un système Unix où tout est fichier, l'ouverture de flux d'entrée, sortie, etc. n'est pas encore possible avec Pépin. Les appels système présentés ici ne concernent donc que de véritables fichiers.

XXIII - Une méthode générique pour gérer les listes chaînées

- [Des listes de données](#)
- [Les listes chaînées sous FreeBSD](#)
- [Les listes chaînées sous Linux](#)
- [Les listes chaînées sous Pépin](#)

Les sources

Le package contenant les sources est téléchargeable ici : [kernel_LinkedList.tgz](#)

XXIII-A - Des listes de données

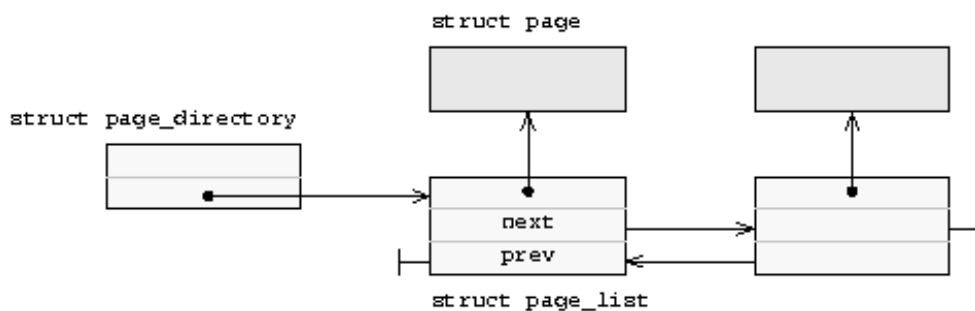
En langage C, l'organisation de données sous forme de liste peut être implémentée en utilisant un tableau ou une liste chaînée. L'utilisation de listes chaînées est très pratique, car contrairement aux tableaux, les listes chaînées sont facilement extensibles et le retrait d'éléments se fait sans perte de mémoire. Les listes chaînées sont donc très utilisées par les noyaux et c'est aussi le cas de Pépin. Par exemple, nous avons défini dans le fichier `mm.h` une structure de données pour décrire une page mémoire. Cette structure contient deux attributs. Le champ `v_addr` correspond à l'adresse en mémoire virtuelle et le champ `p_addr` correspond à l'adresse en mémoire physique :

```
struct page {
    char *v_addr;
    char *p_addr;
};
```

Pour avoir une liste de pages, nous utilisons la structure `struct page_list` ci-dessous :

```
struct page_list {
    struct page *page;
    struct page_list *next;
    struct page_list *prev;
};
```

Ce type de liste est utilisé notamment par la `struct page_directory`, qui décrit un répertoire de pages et indique où sont les différentes tables de pages :



Cette méthode pour implémenter des listes chaînées présente deux inconvénients :

- le premier est que pour chaque type de données à chaîner, il faut créer un type particulier pour organiser la liste. Par exemple, pour faire une liste de processus, il faut créer un nouveau type `struct process_list`. Si nous voulons ensuite faire des listes de fichiers, alors il faut aussi un nouveau type et il en sera ainsi pour chaque type d'objet susceptible d'être organisé par liste ;
- pour ajouter, insérer, enlever, déplacer ou rechercher un élément dans une liste, il est pratique de disposer de fonctions génériques pour ne pas avoir à réécrire à chaque fois les mêmes portions de code. Or, si nous utilisons à chaque fois des types particuliers pour effectuer le chaînage, nous sommes obligés d'utiliser à chaque fois des fonctions particulières.

Heureusement, les noyaux Linux et FreeBSD proposent chacun une méthode pour implémenter de façon générique les listes chaînées. C'est ce que je vais essayer de présenter ci-dessous.

XXIII-B - Les listes chaînées sous FreeBSD

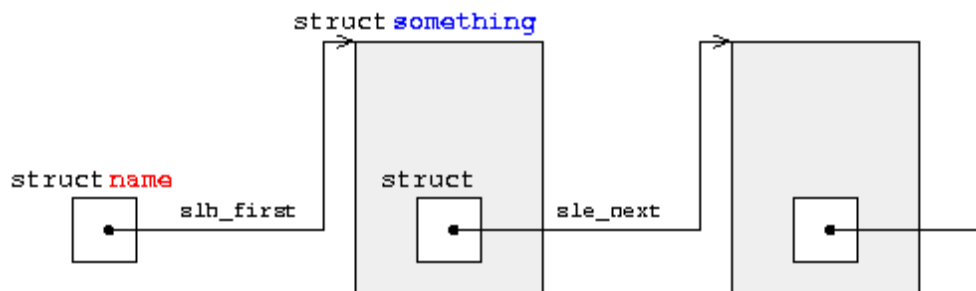
Les systèmes d'exploitation de la famille BSD utilisent une implémentation générique des listes chaînées située dans le fichier `sys/queue.h`. Cette implémentation, qui propose quatre types de listes, est décrite de façon détaillée sur <https://www.freebsd.org/cgi/man.cgi?query=queue&sektion=3>. Bien que la famille *BSD propose quatre types différents de listes, celles-ci partagent des caractéristiques communes :

- la structure permettant le chaînage est intégrée à la structure des objets à chaîner ;
- le programmeur décide du nom et du type de la structure permettant le chaînage ;
- chaque liste est pointée par une « tête de liste » ;
- la définition et la manipulation des listes reposent uniquement sur un jeu de macro (cf. #define).

XXIII-B-1 - Le type SLIST

Le premier type de liste, le type **SLIST**, est une interface permettant de créer des listes simplement chaînées. Supposons que l'on veuille avoir une liste de `struct something` :

- une structure anonyme permettant le chaînage est intégrée au sein de la structure ;
- une tête de liste de type `struct name` pointe sur le premier élément.



L'utilisation des macros pour gérer la liste est assez intuitive. Par exemple :

```
#include <stdio.h>
#include <stdlib.h>
#include "queue.h"

struct page {
    char *v_addr;
    char *p_addr;
    SLIST_ENTRY(page) plist;
};

void main(void)
{
    struct page *p1, *p2, *pp;
    SLIST_HEAD(my_pglist, page) pglist;           /* Declare the list. */

    SLIST_INIT(&pglist);                          /* Initialize the list. */

    p1 = malloc(sizeof(struct page));
    p1->v_addr = (char*) 0x1234;
    p1->p_addr = (char*) 0xffff1234;

    SLIST_INSERT_HEAD(&pglist, p1, plist);         /* Insert at the head. */

    p2 = malloc(sizeof(struct page));
    p2->v_addr = (char*) 0x5678;
    p2->p_addr = (char*) 0xffff5678;

    SLIST_INSERT_AFTER(p1, p2, plist);             /* Insert after. */

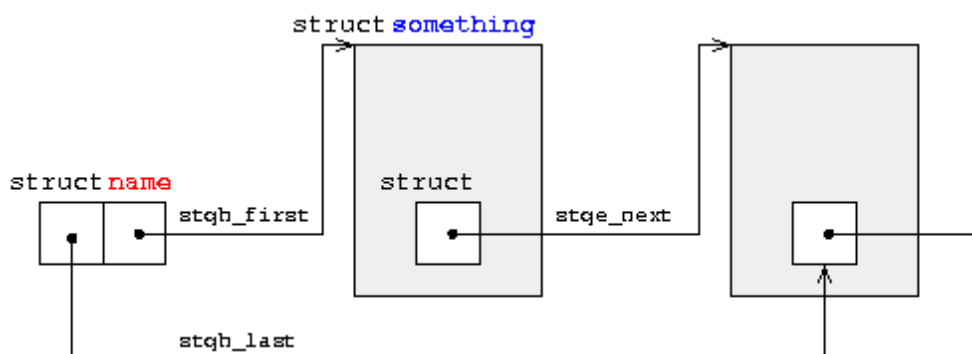
    pp = SLIST_FIRST(&pglist);                    /* Get the first item */

    SLIST_FOREACH(pp, &pglist, plist)             /* Forward traversal. */
        printf("%p -> %p\n", pp->v_addr, pp->p_addr);

    SLIST_REMOVE(&pglist, p2, page, plist); /* Deletion. */
    free(p2);
}
```

XXIII-B-2 - Le type STAILQ

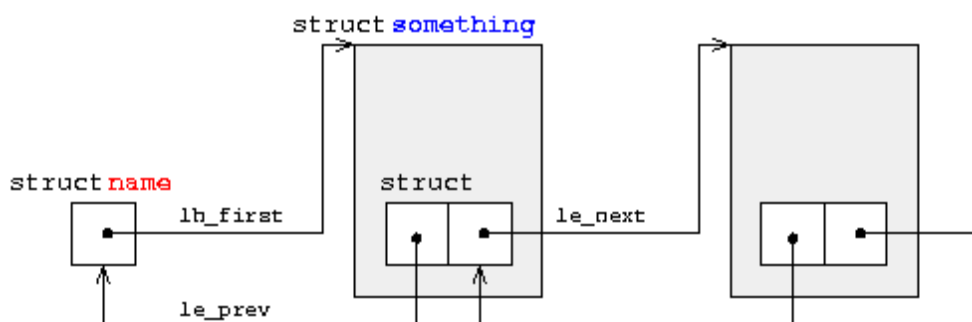
Le type STAILQ, est similaire au type précédent avec en plus un pointeur vers la fin de la liste, ce qui permet d'y ajouter un élément sans avoir à traverser toute la liste :



XXIII-B-3 - Le type LIST

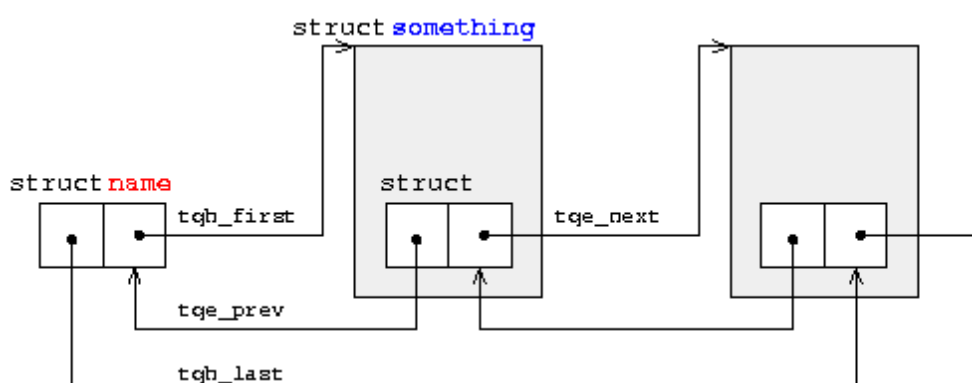
Le type LIST est doublement chaîné de façon à ce que :

- un élément puisse être enlevé sans avoir à traverser toute la liste ;
- un élément puisse être inséré avant ou après n'importe quel autre élément sans avoir à traverser toute la liste ;
- la liste puisse être parcourue dans les deux sens.



XXIII-B-4 - Le type TAILQ

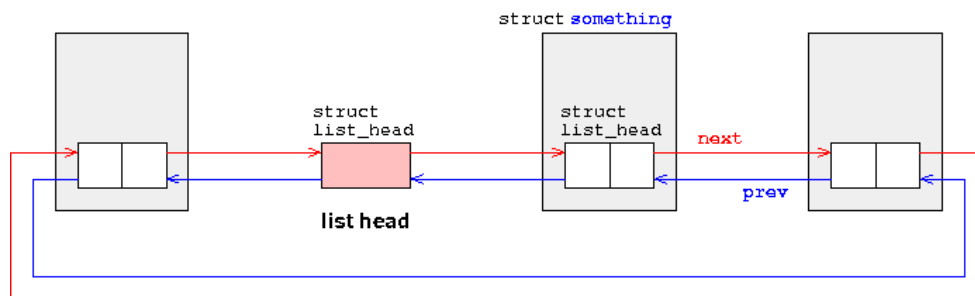
Le type TAILQ est similaire au type précédent avec en plus un pointeur vers la fin de la liste, ce qui permet d'y ajouter un élément sans avoir à traverser toute la liste :



XXIII-C - Les listes chaînées sous Linux

Linux propose une implémentation très originale de liste circulaire doublement chaînée et dont les sources sont dans le fichier `include/linux/list.h`. Cette implémentation a certaines caractéristiques :

- la structure permettant le chaînage est intégrée à la structure des objets à chaîner (c'est d'ailleurs le seul point commun avec les listes de type `*BSD`) ;
- la tête de liste fait partie intégrante de la liste !
- la structure permettant le chaînage est du type défini `struct list_head` ;
- la définition et la manipulation des listes reposent essentiellement sur un jeu de fonctions `inline`

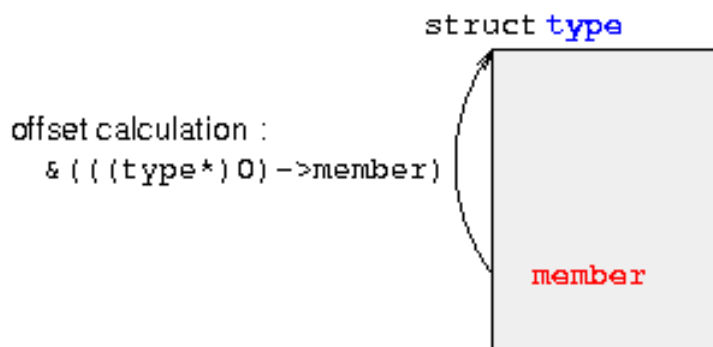


Ce schéma fait apparaître une autre différence fondamentale entre l'implémentation de type `*BSD` et l'implémentation Linux. Dans l'implémentation `*BSD`, chaque structure de liste pointe sur l'objet suivant. Alors que dans l'implémentation Linux, chaque structure de liste pointe sur la structure de liste suivante. Dans l'implémentation Linux, nous avons donc une liste chaînée de structures de type `list_head`. Mais ce que nous voulons manipuler finalement, ce sont bien les objets eux-mêmes : comment retrouver l'adresse d'un objet à partir du pointeur vers sa structure de liste ?

La fonction `list_entry()` fait ce travail :

```
#define list_entry(ptr, type, member) \
    (type*) ((char*) ptr - (char*) &((type*)0)->member)
```

Son implémentation repose sur une astuce expliquée dans les [FAQ du langage C \(question 2.14\)](#). Le schéma ci-dessous met en évidence la formule permettant de calculer l'offset entre l'adresse du début de la structure et l'adresse d'un membre en son sein :



L'utilisation des fonctions pour gérer ce type de liste est ensuite assez intuitive. Par exemple :

```
#include <stdio.h>
#include <stdlib.h>
#include "list.h"

struct page {
```

```

char *v_addr;
char *p_addr;
struct list_head plist;
};

void main(void)
{
    struct page *p1, *p2, *pp;
    struct list_head pglist, *l;

    INIT_LIST_HEAD(&pglist);                /* Initialize the list. */

    p1 = malloc(sizeof(struct page));
    p1->v_addr = (char*) 0x1234;
    p1->p_addr = (char*) 0xffff1234;

    list_add(&p1->plist, &pglist);           /* Insert after the head. */

    p2 = malloc(sizeof(struct page));
    p2->v_addr = (char*) 0x5678;
    p2->p_addr = (char*) 0xffff5678;

    list_add(&p2->plist, &p1->plist);         /* Insert after */

    pp = list_first_entry(&pglist, struct page, plist); /* Get the first item */

    list_for_each(l, &pglist) {              /* Forward traversal. */
        pp = list_entry(l, struct page, plist);
        printf("%p -> %p\n", pp->v_addr, pp->p_addr);
    }

    list_for_each_entry(pp, &pglist, plist)  /* Forward traversal again ! */
        printf("%p -> %p\n", pp->v_addr, pp->p_addr);
}

```

XXIII-D - Les listes chaînées sous Pépin

Le noyau Pépin utilise la même implémentation de listes que pour le noyau Linux. Le code a été légèrement simplifié pour être rendu plus accessible et le nombre de fonctions de manipulation de listes est moindre, mais fonctionnellement, les deux implémentations sont identiques : [list.h](#)

```

#ifndef __LIST__
#define __LIST__

struct list_head {
    struct list_head *next, *prev;
};

#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) \
    struct list_head name = LIST_HEAD_INIT(name)

static inline void INIT_LIST_HEAD(struct list_head *list)
{
    list->next = list;
    list->prev = list;
}

static inline void list_add(struct list_head *new, struct list_head *head)
{
    new->next = head->next;
    new->prev = head;
    head->next->prev = new;
    head->next = new;
}

```

```
static inline void list_del(struct list_head *p)
{
    p->next->prev = p->prev;
    p->prev->next = p->next;
    p->next = 0;
    p->prev = 0;
}

static inline int list_empty(const struct list_head *head)
{
    return head->next == head;
}

#define list_entry(ptr, type, member) \
    (type*) ((char*) ptr - (char*) &((type*)0)->member)

#define list_first_entry(head, type, member) \
    list_entry((head)->next, type, member)

#define list_for_each(p, head) \
    for (p = (head)->next; p != (head); p = p->next)

#define list_for_each_safe(p, n, head) \
    for (p = (head)->next, n = p->next; p != (head); p = n, n = n->next)

#define list_for_each_entry(p, head, member) \
    for (p = list_entry((head)->next, typeof(*p), member); \
         &p->member != (head); \
         p = list_entry(p->member.next, typeof(*p), member)) \

#endif /* __LIST__ */
```

Le code du noyau a été entièrement réécrit en prenant en considération cette nouvelle implémentation. Même si cela a demandé un certain travail, de grandes portions de code ont pu être simplifiées. Par exemple, dans la fonction `load_elf()`, le code suivant :

```
if (get_p_addr((char *) v_addr) == 0) {
    if (pglist->page) {
        pglist->next = (struct page_list *) kmalloc(sizeof (struct page_list));
        pglist->next->next = 0;
        pglist->next->prev = pglist;
        pglist = pglist->next;
    }
    pglist->page = (struct page *) kmalloc(sizeof(struct page));
    pglist->page->p_addr = get_page_frame();
    pglist->page->v_addr = (char *) (v_addr & 0xFFFFF000);
    pd_add_page(pglist->page->v_addr, pglist->page->p_addr, PG_USER, pd);
}
```

a été remplacé par le code ci-dessous, plus court et donc un peu plus lisible :

```
if (get_p_addr((char *) v_addr) == 0) {
    pg = (struct page *) kmalloc(sizeof(struct page));
    pg->p_addr = get_page_frame();
    pg->v_addr = (char *) (v_addr & 0xFFFFF000);

    list_add(&pg->list, pglist);
    pd_add_page(pg->v_addr, pg->p_addr, PG_USER, pd);
}
```

XXIV - Un premier shell

- **Un processus qui utilise la console**
- **Créer un nouveau processus**
- **Allouer dynamiquement de la mémoire à un processus avec `malloc()`**

- **Le noyau et un premier shell**

Les sources

Le package contenant les sources est téléchargeable ici : [kernel_Shell.tgz](#)

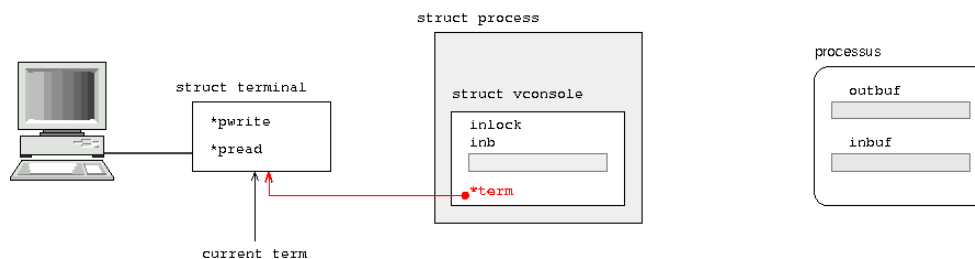
XXIV-A - Un processus qui utilise la console

Nous avons vu dans un chapitre précédent comment gérer les interruptions du clavier pour saisir des caractères et afficher un message directement sur la console. La réalisation d'un shell demande d'aller un peu plus loin dans le traitement de ces interruptions pour que les caractères saisis soient effectivement « lus » par le shell. Cette partie montre une implémentation simple permettant à de multiples processus d'utiliser une console pour saisir et afficher des caractères.

XXIV-A-1 - Attacher une console à un processus pour gérer les entrées/sorties d'un terminal

Au démarrage, le noyau initialise une console par défaut grâce à la structure `struct terminal`. Cette structure contient deux champs, `pread` et `pwrite`, qui pointent vers les processus qui utilisent la console en lecture ou en écriture. Dans cette architecture, un seul processus peut donc lire ou écrire à la fois.

Le pointeur `current_term` indique quel est le terminal actuellement en cours d'utilisation. L'implémentation actuelle n'utilise qu'un seul terminal, mais ce pointeur devient indispensable dans l'hypothèse où plusieurs terminaux existent. À sa création, un processus est rattaché à la console en cours par le biais du champ `term` :



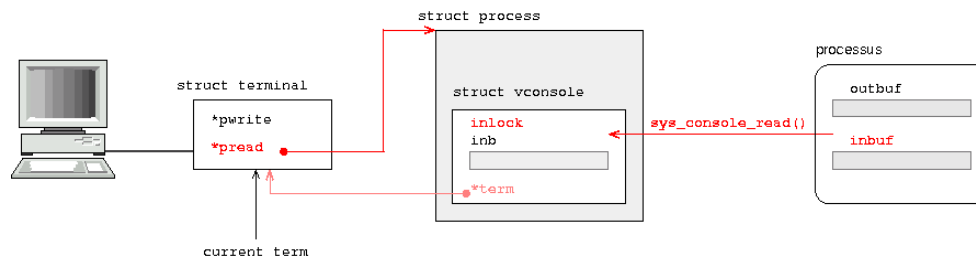
XXIV-A-2 - Utiliser l'appel système `sys_console_read()` pour entrer des données au clavier

Quand un processus souhaite lire des données entrées au clavier, il invoque l'appel système `sys_console_read()` en passant en paramètre l'adresse d'un buffer d'entrée (`inbuf` dans le schéma ci-dessous). Si un autre processus utilise déjà la console en lecture, le processus se met en attente jusqu'à ce qu'elle se libère. Ensuite, l'appel système `sys_console_read()` fait pointer le champ `pread` de la structure du terminal sur le processus en attente de caractères à lire :

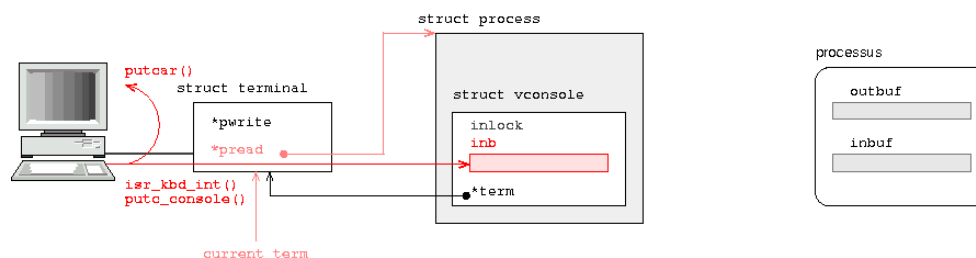
```
/* Bloque si la console est déjà utilisée */
while (current_term->pread);

current->console->term->pread = current;
current->console->inlock = 1;

/* Bloque jusqu'à ce que le buffer utilisateur soit rempli */
while (current->console->inlock == 1);
```



Quand une touche est pressée, une IRQ est levée et la fonction `isr_kbd_int()` est activée. Cette fonction lit le code émis par le clavier. Si le code correspond à la saisie d'un caractère, la fonction `putc_console()` est appelée. En mode buffer (le mode le plus général), `putc_console()` affiche le caractère à l'écran et l'ajoute dans le buffer de la console, `console->inb`, du processus en lecture :



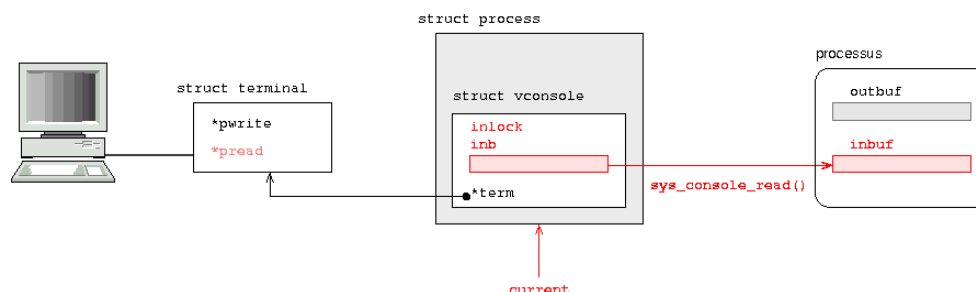
Une fois que le caractère `\n` est saisi, le processus est détaché de la console et `pread` prend la valeur NULL. C'est seulement dans un deuxième temps que le contenu du buffer `inb` est copié dans le buffer d'entrée de l'espace utilisateur du processus :

```
/* Bloque jusqu'à ce que le buffer utilisateur soit rempli */
while (current->console->inlock == 1);

strcpy(u_buf, current->console->inb);

return strlen(u_buf);
```

Pourquoi ne copions-nous pas directement le caractère dans le buffer `inbuf` du processus en attente de lecture ? Quand un caractère est saisi, une interruption est levée et interrompt le processus en cours qui n'est pas forcément le processus en lecture. Pour copier le caractère dans le tampon `inbuf` du processus en lecture, il faudrait alors changer d'espace d'adressage, copier le caractère, puis revenir dans l'espace d'adressage du processus courant. J'ai préféré l'autre solution qui est de copier le caractère dans un buffer intermédiaire situé directement dans la structure `struct console`, donc dans l'espace du noyau :



Le code de l'appel système `sys_console_read()` : `syscalls/sys_console_read.c`

Le code de la fonction `putc_console()` : `console.c`

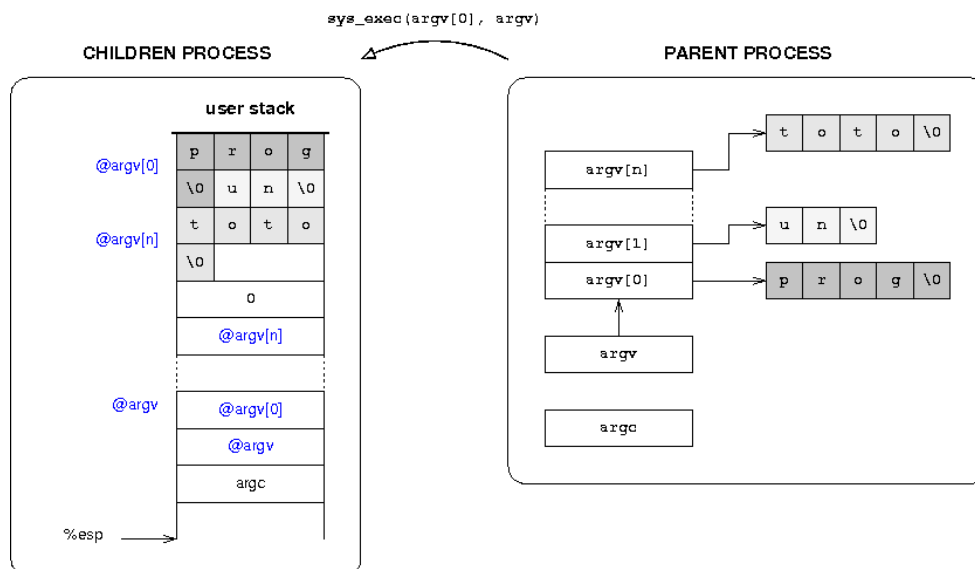
XXIV-B - Créer un nouveau processus

L'appel système `sys_exec()` (syscalls/sys_exec.c) permet de créer et d'exécuter un nouveau processus. C'est un appel système très simple qui :

- 1 Vérifie que le nom du fichier exécutable passé en argument correspond bien à un fichier ;
- 2 Appelle la fonction `load_task()`

XXIV-B-1 - Passer des arguments à un programme

La fonction `load_task()` a été modifiée pour permettre le passage d'arguments du processus parent au processus enfant. La fonction principale `main()` prend ses arguments sur la pile (comme d'ailleurs toute fonction). Y placer ces arguments pour les rendre disponibles au nouveau processus ne pose pas de difficulté particulière. Mais où copier les chaînes de caractères à passer en paramètre ? Une solution est de les copier à un extrême de l'espace d'adressage afin de ne pas gêner le développement de la pile ou du heap. Deux emplacements semblent alors possibles : soit avant le heap utilisateur, soit au sommet de la pile. Cette dernière solution, standard sous Unix, est celle que nous avons implémentée :

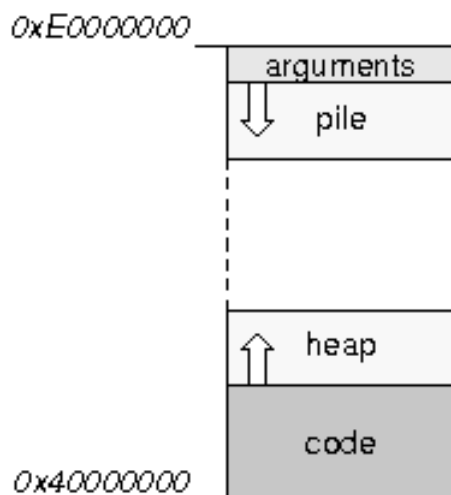


Le code qui copie les données d'espace utilisateur à espace utilisateur (ici du processus parent au processus enfant) comporte une subtilité ! À un moment donné, on ne peut accéder qu'à un seul espace. Il faut donc faire transiter ces données par l'espace du noyau, qui est commun. La copie se fait donc en deux temps. La création du nouveau processus se fait ensuite comme à l'accoutumée.

XXIV-C - Allouer dynamiquement de la mémoire à un processus avec `malloc()`

L'implémentation de `malloc()` repose sur l'appel système `sys_sbrk()` : syscalls/sys_sbrk.c

Cet appel système très simple ne fait que mettre à jour le pointeur qui pointe sur le sommet du heap. Mais où commence le heap ? La fonction `load_elf()`, qui charge l'exécutable, a été modifiée pour mettre à jour les pointeurs sur le début et la fin des zones de code et de données. À partir de ces valeurs, la fonction `load_task()` peut évaluer la base du heap. L'utilisation de la mémoire virtuelle par le processus correspond alors au schéma suivant :



L'implémentation des fonctions `malloc()` et `free()` repose sur les mêmes principes que pour les fonctions `kmalloc()` et `kfree()` du noyau. Attention au fait qu'il s'agit là d'une implémentation très simple pour gérer des blocs de mémoire au niveau de l'espace utilisateur !

XXIV-D - Le noyau et un premier shell

Les sources du shell sont dans `Shell/userland`. Le répertoire contient également les sources d'une minibibliothèque de fonctions et les sources de la commande `cat` qui permet d'afficher le contenu d'un fichier.

The screenshot shows a terminal window with the following output:

```
[Multiboot-elf, <0x100000:0x4e21:0x0>, <0x105000:0x1b0:0x21d90>, shtab=0x127
Za8, entry=0x100000]
boot
Pepin is booting...
RAM detected : 639k (lower), 31680k (upper)
Loading GDT OK
Loading IDT OK
Configure PIC OK
Loading Task Register OK
Enabling paging OK
Partition found on block: 63, size: 3969 blocks, bootable: 0x80
Mount root partition (ext2fs) OK
Interrupts are enable. System is ready !

interrupt
minishell> ls
.
..
grub
kernel
bin
tmp
cat
foo.txt
minishell> cat foo.txt
```

XXV - Implémenter les signaux

- Un simple problème d'affichage, conséquences en termes d'architecture
- Implémenter la filiation entre processus
- Attendre la fin d'un processus avec l'appel système `sys_wait()`
- Implémenter les signaux

Les sources

Le package contenant les sources est téléchargeable ici : [kernel_Signals.tgz](#)

XXV-A - Un simple problème d'affichage, conséquences en termes d'architecture

Un simple problème d'affichage

Au chapitre précédent, nous avons implémenté un minishell et une commande `cat`. Bien que l'ensemble fonctionne convenablement, il y a un problème d'affichage, car le shell affiche directement l'invite de saisie sans attendre la fin de la commande `cat` :

```
minishell> cat foo.txt
minishell> un
deux
trois
```

Attendre la fin d'un processus : l'appel `sys_wait()`

Nous voudrions ici que le shell attende la fin de la commande `cat` avant de continuer. Plusieurs solutions sont envisageables. L'appel système `sys_exec()` pourrait par exemple bloquer en attendant la fin de la commande exécutée. Mais cela poserait un nouveau problème, car le shell bloquerait alors pour toutes les commandes lancées et ça n'est pas nécessairement ce que nous voulons. En fait, nous voudrions que le shell bloque pour certaines commandes, comme la commande `cat`, mais pas pour d'autres, par exemple dans le cas du lancement d'un serveur http ou ftp. Une solution est d'implémenter un nouvel appel qui permette d'attendre explicitement la fin d'un processus : l'appel système `sys_wait()`.

Processus parent et enfant, une question de vocabulaire ?

Un processus peut en créer un nouveau grâce à l'appel `sys_exec()` puis attendre éventuellement sa terminaison avec `sys_wait()`. Pour distinguer les deux processus, on parle communément de processus parent et de processus enfant. Le processus parent qui invoque l'appel `sys_wait()` bloque jusqu'à la terminaison du processus fils.

Informé le processus père du bon déroulement grâce à un code de retour

La terminaison d'un processus peut être normale ou bien consécutive à une erreur. Il est souvent important pour un processus père d'être tenu informé des raisons ayant conduit à la terminaison de son processus fils. Une solution est que le fils transmette un code de retour au père au moment où il se termine. Mais comment ? Nous avons déjà vu que l'appel `sys_exit()` sert à clore proprement un processus. Cet appel va en plus positionner la valeur de retour dans un nouveau champ de la struct `process`, le champ `status`. Ensuite, et nous verrons comment, le processus père va récupérer cette valeur grâce à l'appel `sys_wait()`. Notez que par convention, dans le monde Unix, une valeur de retour de 0 indique un déroulement normal du processus tandis que toute autre valeur signifie une erreur.

Un processus père qui a plusieurs processus enfants

Jusque là, nous avons évoqué le cas d'un processus père qui attend la fin de son fils. Mais pourquoi se limiter à un seul processus fils ? Un processus devrait pouvoir lancer en parallèle plusieurs fils. Mais cette fonctionnalité a pour conséquence que l'appel `sys_wait()` soit générique, au sens où le père ne va pas attendre la terminaison d'un processus fils en particulier, mais celle de n'importe lequel de ses fils. Le processus père doit donc disposer d'une liste de ses fils. La notion de filiation n'est donc plus une simple question de vocabulaire ! Quant à l'appel `sys_wait()`, en plus du code de retour, il doit remonter le pid du processus fils qui s'est terminé :

```
pid = wait(&status); /* wait remonte le pid et le code de retour */
```

Terminaison d'un processus et état zombie

Sous Unix, pour dire qu'un processus s'est terminé, on dit qu'il est mort. Processus « parent », processus « enfant », « mort » d'un processus... le vocabulaire issu du système Unix est très imagé, mais vous verrez que cela ne s'arrête pas là !

Un processus père qui utilise `sys_wait()` pour attendre la fin de l'un de ses fils doit pouvoir :

- savoir lequel de ses fils s'est terminé ;
- récupérer le code de retour de ce fils mort.

Le moyen qui permet cela est assez logique et simple à implémenter, mais je vais avoir besoin d'un tout petit rappel !

Jusqu'à présent, la `struct process` d'un processus contient un champ `state` qui indique son état. La valeur 0 signifie que l'entrée est libre, et il n'y a en réalité plus de processus associé à la structure, alors que la valeur 1 signifie que le processus est en cours d'exécution ou prêt à être exécuté. Quelle valeur associer à un processus qui vient de se terminer ?

Quand un fils est mort, il n'est plus exécutable, donc les ressources qu'il utilisait peuvent être rendues au système, à une exception près. La `struct process`, qui contient notamment le champ `status`, ne peut être libérée tant que le père n'a pas pris connaissance de l'état de son fils. Le fils est donc dans un état intermédiaire. Bien que mort, l'entrée qu'il utilise dans la table des processus n'est pas libérée. On dit qu'il est à l'état de « zombie » !

Pour indiquer cet état de zombie, le champ `state` est mis à -1. C'est une valeur pratique dans le sens où les processus exécutables ont leur champ `state` avec une valeur supérieure à 0 et les processus non exécutables une valeur inférieure ou égale.

Avertir un processus d'un événement à l'aide d'un signal

Un processus père possède la liste de ses fils et il peut donc savoir lesquels sont à l'état de zombie grâce au champ `state` de leur `struct process`. Il ne serait cependant pas efficace pour le père de scruter continuellement l'état de ses fils pour savoir lesquels sont morts. Nous avons donc besoin d'un système de signalisation. Dans Pépin, le champ `signal` de la `struct process` est utilisé pour signaler un événement au processus en question. Ce champ est vérifié par l'ordonnanceur au moment où le processus est préempté. Au cas où un signal a été envoyé, le comportement du processus est adapté en conséquence. Notez que l'utilisation de signaux pour avertir un processus de la mort de l'un de ses fils n'est pas la seule utilisation possible d'un système de signalisation ! Dans Pépin, le champ `signal` contient plusieurs bits et à chaque bit correspond un signal particulier lié à un événement particulier.

Cycle de vie et mort d'un processus

Le scénario de vie d'un processus est donc le suivant :

- 1 Un processus père, à l'aide de l'appel système `sys_exec()`, crée un processus fils ;
- 2 Le fils s'exécute ;
- 3 Quand le fils se termine, il appelle `sys_exit()` grâce auquel il rend ses ressources au système, dépose un code de retour dans le champ `status` de son entrée de la table des processus, et devient un zombie ;
- 4 Un signal est envoyé au processus père pour le prévenir que l'un de ses enfants est mort ;
- 5 Le père scrute la liste de ses enfants, repère le zombie, recueille son code de retour, le libère en mettant son champ `state` à 0 puis il l'enlève de sa liste de processus enfants ;
- 6 Après que le père a récupéré le `pid` et le code de retour, l'appel `sys_wait()` retourne.

Une routine personnalisée pour réagir à un signal : l'appel `sys_sigaction()`

Le problème de cette architecture est que si le père n'attend pas son fils via l'appel `sys_wait()`, le fils restera indéfiniment un zombie. Cela pose un problème, car les zombies utilisent des entrées dans la table des processus au risque de provoquer une pénurie. Il faut donc que le père utilise l'appel `sys_wait()`. Mais le père peut avoir autre chose à faire que d'attendre la terminaison de son fils. Comment résoudre ce problème ?

Nous avons évoqué le fait qu'un signal puisse être envoyé à un processus pour le prévenir d'un événement particulier, par exemple la mort d'un enfant. À ce moment, nous aimerions bien que le père invoque d'une façon ou d'une autre

l'appel `sys_wait()` afin de libérer son fils de son état de zombie. Il suffit pour le père, à la réception du signal indiquant la mort de l'un de ses enfants, d'exécuter une routine personnalisée contenant l'appel à `sys_wait()`. Cette solution, standard dans le monde Unix, est implémentée au moyen d'un nouvel appel système : `sys_sigaction()`.

Mort d'un processus père

Deux problèmes se posent quand un processus père meurt :

- le premier est de savoir qui désormais va s'occuper de ses enfants !
- le second concerne le premier processus, qui par définition n'a pas de parents et donc aucun processus pour le libérer.

La solution imaginée pour Pépin est de créer un premier processus permanent, qui ne meurt jamais et qui adopte les processus orphelins.

XXV-B - Implémenter la filiation entre processus

XXV-B-1 - De nouveaux champs dans la struct process

Les champs suivants sont ajoutés à la struct process :

- le champ `parent` pointe sur le processus parent ;
- le champ `child` pointe sur la liste chaînée des processus enfants ;
- le champ `sibling` pointe sur les processus frères.

```
struct process *parent;          /* Processus parent */
struct list_head child;          /* Processus enfants */
struct list_head sibling;         /* Processus conjoints */
```

XXV-B-2 - À la création du processus

Ces nouveaux champs sont initialisés lors de la création du processus par la fonction `load_task()`. Dans tous les extraits suivants, la variable `previous` pointe sur le processus parent. Le code suivant attribue un parent au nouveau processus :

```
/* Pointe sur le processus parent (ou le pid 0 si le parent est mort) */
if (previous->state != 0)
    p_list[pid].parent = previous;
else
    p_list[pid].parent = &p_list[0];
Initialisation de la liste des enfants. Au départ, cette liste est vide :
/* Initialise la liste des enfants du nouveau processus */
INIT_LIST_HEAD(&p_list[pid].child);
Mise à jour de la liste des enfants du parent :
/* Ajoute le nouveau processus dans la liste des enfants du parent */
if (previous->state != 0)
    list_add(&p_list[pid].sibling, &previous->child);
else
    list_add(&p_list[pid].sibling, &p_list[0].child);
```

XXV-B-3 - À la mort du processus

Quand le processus se termine, l'appel système `sys_exit()` reporte l'information auprès du père et des enfants. En ce qui concerne le père, la mise à jour se fait indirectement (nous verrons plus loin comment), par l'envoi d'un signal de terminaison :

```

/* Met à jour la liste des processus du père */
if (current->parent->state > 0)
    set_signal(&current->parent->signal, SIGCHLD);
else
    printk("WARNING: sys_exit(): process %d without valid parent\n", current->pid);
Les enfants du processus qui se termine se voient attribuer un nouveau père :
/* Donne un nouveau père aux enfants */
list_for_each_safe(p, n, &current->child) {
    proc = list_entry(p, struct process, sibling);
    proc->parent = &p_list[0];
    list_del(p);
    list_add(p, &p_list[0].child);
}

```

XXV-C - Attendre la fin d'un processus avec l'appel système sys_wait()

Il s'agit d'un appel très simple qui bloque le processus jusqu'à ce que l'un de ses enfants soit mort :

```

Algorithme : sys_wait
Paramètre : code de retour &status
Retour : pid
Début
    Tant qu'aucun signal SIGCHLD n'est reçu alors
        le processus ne fait rien
    Pour chaque processus enfant
        Si l'enfant est à l'état zombie alors
            Le pid est récupéré
            Le code de retour est récupéré
            Le fils est libéré
            Le fils est enlevé de la liste des processus fils
            Le signal SIGCHLD est supprimé
        La fonction retourne
Fin

```

XXV-D - Implémenter les signaux

À la base de l'implémentation des signaux, deux champs ont été ajoutés dans la struct process. Le champ signal, sur 32 bits, est destiné à recevoir les signaux. Un signal est reçu quand un bit est mis à 1. Le tableau sigfn pointe sur les routines de service à activer lors de la réception d'un signal :

```

u32 signal;
void* sigfn[32];

```

Le fichier signal.h définit les signaux ainsi que les routines de service par défaut.

```

#define SIGHUP          1
#define SIGINT          2
#define SIGQUIT         3
#define SIGBUS          7
#define SIGKILL         9
#define SIGUSR1        10
#define SIGSEGV        11
#define SIGUSR2        12
#define SIGPIPE        13
#define SIGALRM        14
#define SIGTERM        15
#define SIGCHLD        17
#define SIGCONT        18
#define SIGSTOP        19

#define SIG_DFL         0      /* default signal handling */
#define SIG_IGN         1      /* ignore signal */

#define set_signal(mask, sig)  *(mask) |= ((u32) 1 << (sig - 1))
#define clear_signal(mask, sig) *(mask) &= ~((u32) 1 << (sig - 1))

```

```
#define is_signal(mask, sig)    (mask & ((u32) 1 << (sig - 1)))

int dequeue_signal(int);
int handle_signal(int);
```

XXV-D-1 - Traiter les signaux

Au moment où un processus est préempté pour être exécuté, l'ordonnanceur vérifie si celui-ci a reçu un signal pour le traiter :

```
/* Traite les signaux */
if ((sig = dequeue_signal(current->signal)))
    handle_signal(sig);
```

La fonction `dequeue_signal()` indique quel est le plus bas signal reçu (en partant de 0 jusqu'à 31). Si un signal est reçu, la fonction `handle_signal()` le traite. Ces deux fonctions sont définies dans le fichier `signal.c`.

XXV-D-1-a - Modifier l'algorithme de l'ordonnanceur

La fonction `handle_signal()`, bien que possédant un algorithme simple, est l'une des plus complexes du noyau en raison des difficultés d'implémentation de la gestion des routines personnalisées :

```
Algorithme : handle_signal
Paramètre : signal
Début
Si la routine de service associée au signal est de type ignorer (SIG_IGN) alors
    le signal est effacé
Sinon si la routine de service associée au signal est de type default (SIG_DFL) alors
    Si le signal est SIGHUP, SIGINT ou SIGQUIT alors
        le processus se termine avec l'appel à sys_exit()
    Sinon si le signal est SIGCHLD (mort d'un enfant) alors
        on ne fait rien (l'appel à sys_wait() par le père se chargera du travail !)
    Sinon le signal est effacé
Sinon on exécute la fonction associée au signal et déterminée par l'utilisateur
Fin
```

XXV-D-2 - Utiliser une fonction de traitement de signal personnalisée

XXV-D-2-a - Associer une fonction personnalisée avec l'appel `sys_sigaction()`

L'appel système `sys_sigaction()` permet d'associer à un signal l'adresse d'une fonction utilisateur :

```
asm(" mov %?x, %0    \n \
    mov %?x, %1"
    : "=m"(sig), "=m"(fn) :);

current->sigfn[sig] = fn;
```

XXV-D-2-b - Exécuter une fonction personnalisée - le problème posé

À la réception d'un signal, la méthode pour exécuter une fonction personnalisée est simple : il suffit de modifier le pointeur d'instruction `%eip` en le faisant pointer sur la fonction. Mais cela pose un premier problème, car à l'issue de l'exécution de la fonction, le processus doit reprendre là où il en était et il faut donc sauvegarder quelque part la valeur originelle de `%eip` stockée dans `current->regs->eip`. Le second problème provient du fait que quand le processus est interrompu par l'ordonnanceur alors qu'il exécute cette fonction, les registres de la struct `process` sont écrasés. Ça n'est donc pas uniquement la valeur de `%eip` qu'il faut sauvegarder, mais bien l'ensemble des registres.

XXV-D-2-c - Sauvegarder le contexte sur la pile utilisateur

Une fois la fonction de traitement du signal terminée, le processus doit reprendre son fonctionnement normal. Malheureusement, la sauvegarde des registres dans la struct process a été écrasée lors de l'exécution de la fonction de traitement du signal. Avant d'exécuter cette fonction, il faut donc sauvegarder les registres quelque part. Une solution standard est de sauvegarder les registres de la struct process sur la pile utilisateur. Une fois la fonction de traitement du signal terminée, les registres de la struct process seront restaurés grâce à l'appel système `sys_sigreturn()`.

Cette solution résout-elle tous les problèmes ? Non, car pour rester standard, l'appel à `sys_sigreturn()` ne doit pas être réalisé explicitement dans la fonction utilisateur, mais automatiquement par le noyau.

XXV-D-2-d - Revenir d'une routine avec la technique du stack-smashing

Quand une fonction personnalisée se termine, elle fait comme n'importe quelle fonction : elle récupère sur la pile l'adresse `eip` qui indique où reprendre l'exécution. On peut donc penser qu'il suffit de mettre à cet endroit de la pile l'adresse de `sys_sigreturn()` pour exécuter automatiquement cette fonction. C'est correct ? Eh bien non ! N'oubliez pas que la fonction personnalisée de traitement du signal s'exécute en mode utilisateur alors que la fonction `sys_sigreturn()` est dans l'espace d'adressage du noyau, il faut donc passer par un appel système.

L'appel permettant d'appeler la fonction `sys_sigreturn()` est le suivant :

```
mov eax, 14
int 0x30
```

Au retour de la routine de service, il faudrait que `%eip` pointe sur ce code et qu'en plus, celui-ci soit dans l'espace utilisateur. Une méthode est de copier directement le code assembleur de cet appel sur la pile utilisateur :

```
esp[19] = 0x0030CD00;
esp[18] = 0x0000EB8;
```

La valeur de `%eip` est alors définie sur la pile par :

```
esp[0] = (u32) &esp[18];
```

Ainsi, par cette implémentation, au retour de la fonction de traitement personnalisée, le registre `%eip` pointe sur l'adresse `&esp[18]` où est le code assembleur exécutant l'appel système pour `sys_sigreturn()`.

XXV-D-2-e - Revenir avec `sys_sigreturn()`

La fonction `sys_sigreturn()`, qui est définie dans le fichier `syscalls/sys_sigreturn.c`, ne fait que rétablir les registres dans la struct process à partir de la sauvegarde effectuée sur la pile.

XXVI - Annexe A : Compilation séparée en assembleur sous Unix

XXVI-A - Le programme principal

Le fichier suivant contient le code principal. Il incrémente la variable `mavar` avec la fonction `incr` définie dans un autre fichier. Cette fonction externe doit être déclarée au début du fichier avec la directive `extern`.

Le linker a besoin qu'un point d'entrée soit défini pour déterminer où débute le programme. Par défaut, ce point d'entrée correspond au label `_start` et il doit être déclaré de façon globale (sans quoi il ne serait pas vu par le linker) :

```
extern incr
global _start
```

```

_start:
    push dword mavar
    call incr

mavar: dw 0x01
  
```

XXVI-B - La fonction

La fonction `incr` incrémente un entier sur 4 octets passés en argument (passage par pointeur). Pour que cette fonction soit visible au linker, on la déclare au début du fichier grâce à la directive `global` :

```

global incr

incr:
    push eax ; sauvegarde du registre
    push ebx ; sauvegarde du registre

    push ebp
    mov ebp, esp

    mov ebx, [ebp+16] ; copie de l'adresse de la variable
    mov eax, [ebx]    ; copie de la valeur
    add eax, 1        ; incrémentation la valeur
    mov [ebx], eax    ; modification de la variable pointée

    mov esp, ebp
    pop ebp

    pop ebx
    pop eax
    ret
  
```

XXVI-B-1 - Compilation et édition de liens

On produit à partir de chaque fichier source un fichier objet au format ELF grâce à la commande suivante :

```

nasm -f elf incr.asm -o incr.o
nasm -f elf main.asm -o main.o
  
```

L'édition de liens est faite avec la commande `ld`. L'argument `--oformat binary` permet d'obtenir un binaire pur :

```

ld --oformat binary -Ttext 0 main.o incr.o
  
```

En désassemblant le binaire `a.out` obtenu, on voit très clairement l'assemblage des deux fichiers objets en un seul :

```

ndisasm -u a.out
00000000 680A000000    push dword 0xa
00000005 E806000000    call dword 0x10
0000000A 0100         add [eax],eax
0000000C 90           nop
0000000D 90           nop
0000000E 90           nop
0000000F 90           nop
00000010 50           push eax
00000011 53           push ebx
00000012 55           push ebp
00000013 89E5         mov ebp,esp
00000015 8B5D08       mov ebx,[ebp+0x8]
00000018 8B03         mov eax,[ebx]
0000001A 0501000000   add eax,0x1
0000001F 8903         mov [ebx],eax
00000021 89EC         mov esp,ebp
00000023 5D           pop ebp
00000024 5B           pop ebx
  
```



```
00000025 58          pop eax
00000026 C3          ret
```

XXVI-C - Création d'exécutables sous Unix

XXVI-C-1 - Un programme principal en assembleur

```
extern printf
global main
main:
    push eax
    push dword msg
    call printf
    pop eax
    pop eax
    ret

msg db "Hello world", 13, 10, 0
```

Ce programme est l'équivalent de :

```
#include <stdio.h>

void main()
{
    printf("Hello world\n");
}
```

On le compile de la façon suivante :

```
nasm -f elf main.asm -o main.o
gcc main.o
```

XXVI-C-2 - Un programme principal en C et une fonction en assembleur

XXVI-C-2-a - Le programme principal

```
#include <stdio.h>

extern void incr(int*);

void main()
{
    int a = 1;

    incr(&a);

    printf("%d\n", a);
    return;
}
```

XXVI-C-2-b - Compilation et linkage

Pour compiler le fichier principal et le lier au fichier objet :

```
gcc main.c incr.o
```

XXVII - Annexe B - arithmétique en base 16

XXVII-A - Conversion entre bases

Hexa	Décimal	Puissance de 2	Commentaire
0xA	10		
0xB	11		
0xC	12		
0xD	13		
0xE	14		
0xF	15		
0x10	16	24	
0xFF	255		
0x100	256	28	
0x400	1024	210	1k
0x800	2048	211	2k
0x1000	4096	212	4k
0x10.000	65.536	216	64k
0x100.000	1.048.576	220	1Mo
0x1000.000	16.777.216	224	16 Mo
0x40.000.000	1.073.741.824	230	1G
0x80.000.000	2.147.483.648	231	2G
0xC0.000.000	3.221.225.472		3G
0x100.000.000	4.294.967.296	232	4G

À retenir :

- 0x8 est la moitié de 0x10
- $0xn * 0x10 = 0xn0$ (par exemple $0xEF * 0x10 = 0xEF0$)

XXVIII - Annexe C - Bochs en mode debug

Sous Linux, Bochs doit avoir été compilé avec l'option `--enable-debugger` pour être utilisé en mode debug.

XXVIII-A - Commandes utiles

- h - show list of debugger commands

XXVIII-A-1 - Général

- n - execute instruction stepping over subroutines
- s [count] - execute #count instructions (default is one instruction)
- b <addr> - set a physical address instruction breakpoint
- c - continue executing
- r - list of CPU registers and their contents
- info sregs - show segment registers
- info eflags - show decoded EFLAGS register

- `xp xp /nuf <addr>` - examine memory at physical address
- `print-stack [num_words]` - print the num_words top 16 bit words on the stack

XXVIII-A-1-a - Segments, interruptions, gestion de tâches

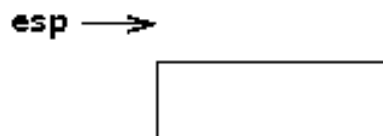
- `info gdt` - show global descriptor table
- `info idt` - show interrupt descriptor table
- `info tss` - show current task state segment

XXVIII-A-1-b - Pagination

- `info tab` - show page tables
- `page <addr>` - calc physical address
- `info creg` - show CR0-CR4 registers
- `lb <addr>` - set a linear address instruction breakpoint
- `x /nuf <addr>` - examine memory at linear address
- `info dirty` - show physical pages dirtied (written to) since last display

XXIX - Annexe D - Gérer les arguments sur la pile avec les Stack Frame

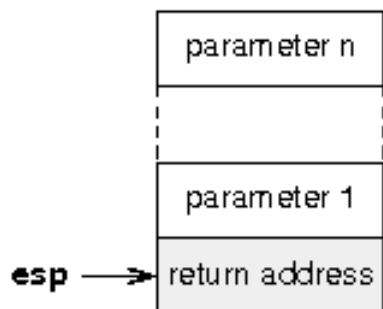
Avant que la pile soit utilisée, le pointeur de pile pointe sur la case mémoire juste avant le début de la pile :



Pour passer des paramètres à une fonction, plusieurs méthodes sont possibles :

- utiliser les registres, mais cela limite le nombre de paramètres qu'on peut passer ;
- empiler les paramètres sur la pile avec l'instruction `push`.

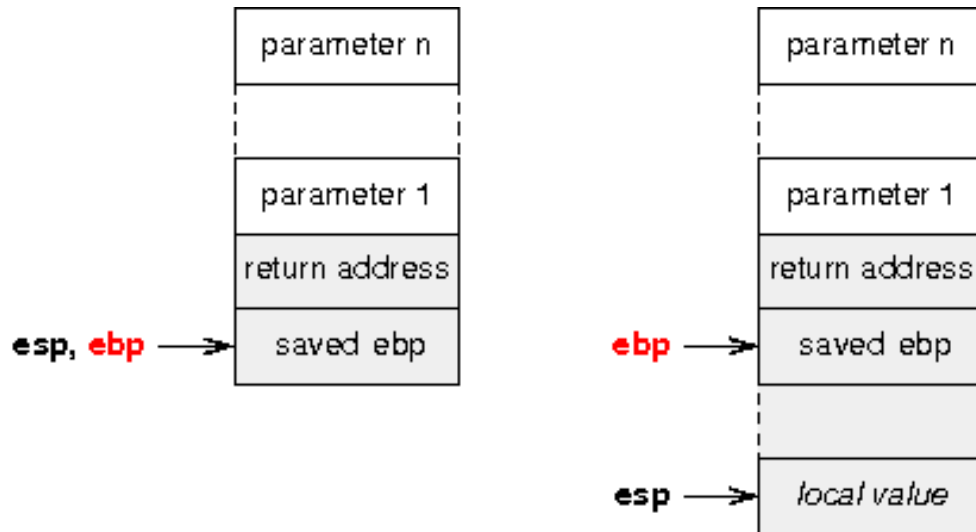
L'instruction `call` permet ensuite d'appeler une fonction. Le processeur empile automatiquement le registre `eip` et éventuellement le registre `cs` afin de pouvoir poursuivre l'exécution du code principal une fois la fonction terminée :



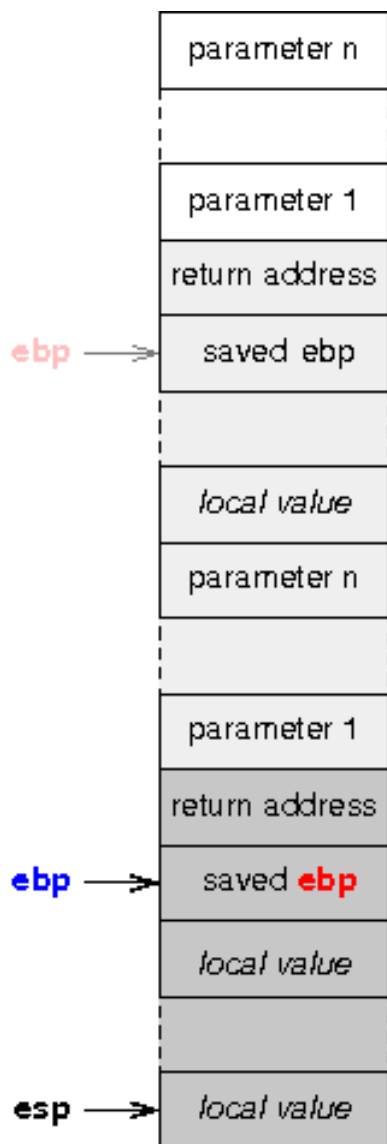
Une fois dans la fonction, pour accéder aux paramètres passés en argument, il est possible de se baser sur la valeur du pointeur de pile `esp`. Mais cela pose un problème, car la fonction peut être amenée à empiler des données, et la valeur de `esp`. Une solution simple est d'utiliser un registre dédié : le registre `ebp`.

La première instruction de la fonction va donc être de sauvegarder le registre `ebp` en l'empilant puis de placer dans `ebp` la valeur courante du pointeur de pile :

```
pushl    ?p
movl     %esp, ?p
```



Si une autre fonction est appelée, toutes les informations sont conservées sur la pile :



XXX - Annexe E - Déboguer le noyau avec gdb

Un noyau lancé avec Bochs ou qemu peut être débogué très simplement avec gdb, et ce, sans manipulation très difficile. Cette annexe ne détaille pas les commandes de **gdb**, mais résume seulement la manipulation permettant son utilisation.

XXX-A - Compiler avec l'option -g

Au préalable, comme pour tout binaire généré par gcc destiné à être débogué par gdb, le noyau doit avoir été compilé avec l'option -g.

XXX-B - Lancer l'émulateur

Il est possible d'utiliser Bochs, mais il devra au préalable avoir été compilé avec l'option `--enable-gdb-stub`. À noter que cette option n'est pas compatible avec l'option `--enable-debugger` qui permet le mode debug. Cela nécessite donc d'avoir deux binaires Bochs (avec par exemple un suffixe différent) : un pour le mode debug, un pour le mode gdb-stub.

Sinon, il est possible d'utiliser qemu. La commande suivante lance l'émulateur en lui précisant d'attendre une connexion avec gdb pour démarrer le CPU :

```
qemu -hda c.img -s -S &
```

XXX-C - utiliser gdb

Le débogueur a besoin des sources pour être vraiment effectif. Une méthode simple est de le lancer directement à partir du répertoire contenant les sources :

```
cd kern/  
gdb
```

Ensuite, il faut indiquer le port réseau fournissant les données à gdb. Par défaut, Qemu utilise le port 1234 :

```
> target remote localhost:1234
```

Il faut fournir en entrée un fichier contenant la table des symboles. Le noyau compilé avec l'option -g fait l'affaire :

```
> symbol-file kernel
```

Voilà ! Il est maintenant possible de déboguer le noyau avec gdb comme s'il s'agissait d'un simple exécutable. On peut par exemple positionner un breakpoint (point d'arrêt) au début de la fonction principale et continuer l'exécution de l'émulateur :

```
> b kmain  
> c
```

XXX-D - Annexe F - Booter avec Grub2

Contrairement à Grub, Grub2 nécessite une partition (on ne peut utiliser le volume disque « brut »).

Pour créer une image de disque avec `bximage` :

```
bximage  
> hd  
> flat  
> 10  
> c.img
```

Pour spécifier les caractéristiques physiques du disque (C/H/S) et créer une partition :

```
fdisk c.img  
> x  
> c 20  
> h 16  
> s 63  
> r  
> n, p, 1  
> a 1  
> w
```

Il faut ensuite associer le disque et la partition avec les loopback afin de pouvoir les manipuler. L'offset (paramètre -o) en octets est obtenu en multipliant le numéro du secteur où débute la partition par 512. L'exemple ci-dessous illustre le cas où il y a 2 partitions avec une première, de 5 Mo, qui débute au secteur 63 :

```
losetup /dev/loop0 c.img
```

```
losetup -o 32256 --sizelimit 5644800 /dev/loop1 c.img
losetup -o 5677056 /dev/loop2 c.img
```

On crée un système de fichiers **ext2** sur la première partition puis on la monte :

```
mke2fs -O ^resize_inode /dev/loop1
mount -t ext2 /dev/loop1 /mnt/virtual
```

L'option `-O ^resize_inode` permet de simplifier la structure du système de fichiers en refusant certaines évolutions.

Ensuite, on installe Grub2. Il faut commencer par décrire les volumes disques :

```
mkdir -p /mnt/virtual/boot/grub
cat > /mnt/virtual/boot/grub/device.map << EOF
(hd0) /dev/loop0
(hd0,1) /dev/loop1
(hd0,2) /dev/loop2
EOF
```

Le fichier suivant décrit comment charger le noyau. Il est possible d'utiliser le standard `multiboot` ou le nouveau standard **multiboot2** :

```
cat > /mnt/virtual/boot/grub/grub.cfg << EOF
set default=0
set timeout=5
set root=(hd0,1)
menuentry "AdaKernel" {
    multiboot /boot/kernel
    boot
}
EOF
cp kern/kernel /mnt/virtual/boot
sudo grub-install --no-floppy --root-directory=/mnt/virtual \
--modules="part_msdos ext2 biosdisk configfile normal multiboot multiboot2" \
/dev/loop0
```

Et voilà ! Grub2 est installé. Suite aux mises à jour du noyau, n'oubliez pas d'utiliser la commande `sync`.

XXXI - GNU Free Documentation License

GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.

51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document « free » in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of « copyleft », which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for freeware, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The « Document », below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as « you ». You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A « Modified Version » of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A « Secondary Section » is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The « Invariant Sections » are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The « Cover Texts » are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A « Transparent » copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not « Transparent » is called « Opaque ».

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The « Title Page » means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, « Title Page » means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section « Entitled XYZ » means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as « Acknowledgements », « Dedications », « Endorsements », or « History ».) To « Preserve the Title » of such a section when you modify the Document means that it remains a section « Entitled XYZ » according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- B List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D Preserve all the copyright notices of the Document.
- E Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H Include an unaltered copy of this License.
- I Preserve the section Entitled « History », Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled « History » in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the « History » section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K For any section Entitled « Acknowledgements » or « Dedications », Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M Delete any section Entitled « Endorsements ». Such a section may not be included in the Modified Version.
- N Do not retitle any existing section to be Entitled « Endorsements » or to conflict in title with any Invariant Section.
- O Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled « Endorsements », provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled « History » in the various original documents, forming one section Entitled « History »; likewise combine any sections Entitled « Acknowledgements », and any sections Entitled « Dedications ». You must delete all sections Entitled « Endorsements ».

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an « aggregate » if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled « Acknowledgements », « Dedications », or « History », the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License « or any later version » applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) YEAR YOUR NAME.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the ...« with...Texts. » line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

XXXII - FAQ : questions souvent posées

- [Le code ne compile pas](#)
- [Le noyau ne fonctionne pas avec mon émulateur](#)
- [La compilation produit l'erreur undefined reference to __stack_chk_fail](#)
- [Pour compiler le noyau sur une plateforme 64 bits](#)

XXXII-A - Réponses

XXXII-A-1 - Le code ne compile pas

Tous les codes en exemple et disponibles ont été compilés et testés sous Linux. Si malgré tous vos efforts vous ne parvenez pas à compiler le code, vous pouvez me contacter en précisant :

- la commande utilisée pour compiler/linker avec les arguments ;
- le message d'erreur.

XXXII-A-2 - Le noyau ne fonctionne pas avec mon émulateur

Tous les noyaux ont été testés avec **Bochs** et **qemu**. Quel que soit votre émulateur, pensez à :

- regarder dans le log généré ;
- vérifier que le fichier de configuration de l'émulateur est correct .

Si votre problème est insoluble, vous pouvez me contacter en précisant :

- la commande utilisée pour compiler/linker avec les arguments ;
- l'émulateur utilisé ;
- le contenu du fichier de logs.

XXXII-A-3 - La compilation produit l'erreur undefined reference to __stack_chk_fail

C'est une erreur causée par gcc qui, pour certaines versions, ajoute par défaut un « canari » dans la pile utilisateur lors de chaque appel de fonction (http://en.wikipedia.org/wiki/Canary_value#Canaries). Pour désactiver ce comportement, il suffit de lui ajouter l'option `-fno-stack-protector`.

XXXII-A-4 - Pour compiler le noyau sur une plateforme 64 bits

Par défaut, sur une distribution Linux 64 bits, les binaires et fichiers objets générés vont être au format 64 bits. Pour générer Pépin, qui est en 32 bits, il faut rajouter certaines options à la compilation et au linkage :

- à la compilation, il faut ajouter l'option `-m32` ;
- au linkage, il faut utiliser l'option `-m elf_i386`.

XXXIII - Conclusion

Document source : **Programmer son propre noyau : une introduction avec Pépin**

Copyright (C) 2008, Arnauld Michelizza

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled « GNU Free Documentation License ».

Cette version comme la version d'origine est sous licence **GFDL**

XXXIV - Note de la rédaction de developpez.com

Quelques ajouts ont été effectués notamment par rapport aux processeurs 64 bits.

La version originale de l'article est disponible [!\[\]\(179f167ede0522ebb4ea025b3ad78ca7_img.jpg\) **ici**](#).

Nous tenons à remercier **Christophe** pour la mise au gabarit et **Claude Leloup** pour la relecture orthographique.