Système d'exploitation

Le 08 février 2013 , SVN-ID 242

Table des matières

1 Introduction aux processeurs Intel 86

1.1 Présentation

1.1.1 Introduction

I8086 16/20 bits, segmentation sommaire (1978)

I80286 16/24 bits, mode protégé (1982)

I80386 32/32 bits, mode protégé, pagination, mode 8086 (1985)

I80486 32/32 bits, mode protégé, pagination, mode 8086, FPU (1989)

Pentium 32/64 bits, (1993)

Pentium 4 32/64 bits, hyperthreading, MMX, SSE, SSE2 (2000)

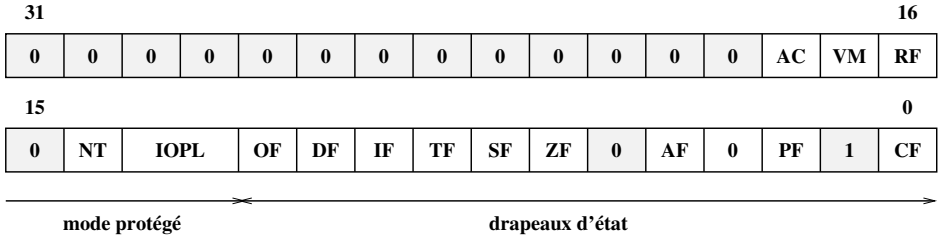
Core Duo 64/64 bits, (2006)

Core Quad 64/64 bits, (2007)

1.1.2 Registres

Voir les figures 1 et 2.

1.1.3 Drapeaux



CF: retenue	NT: tâche Imbriquée
PF: parité	IOPL: Niveau de privilège
AF: retenue auxiliaire	RF: trace
ZF: zero	VM: mode 8086
SF: négatif	AC: vérification d'alignement
OF: overflow	

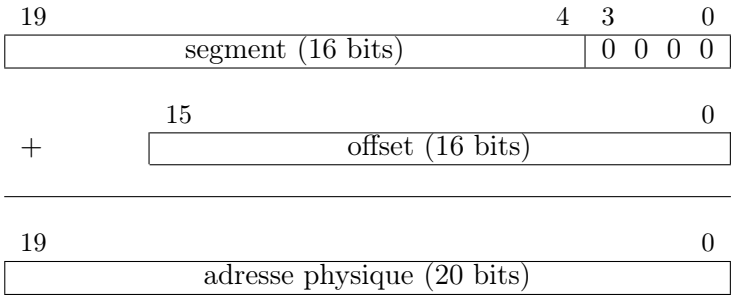
IF,TF,TF: traps et interruptions

1.2 Mode Réel et 8086

C'est le fonctionnement du 8086, il est supporté par tous les autres processeurs de la famille (286, 386, ...).

1.2.1 Adresses segmentées

$\Longrightarrow$ une adresse (20 bits) c'est un segment (16 bits) plus offset (16 bits)
$\Longrightarrow$ DS:AX



1.2.2 Modes d'adressage

- immédiat
- registre
- DS:(BX [+ SI] [+ dépl])
- DS:(BX [+ DI] [+ dépl])
- DS:(SI [+ dépl])
- DS:(DI [+ dépl])
- DS:(dépl)
- SS:(BP + SI [+ dépl])
- SS:(BP + DI [+ dépl])

CS, DS, ES, SS peuvent en général remplacer DS, SS.

1.2.3 Instructions

- Instruction à 0 opérande (ret, retf iret, cli, sti)
- Instruction à 1 opérande (8 ou 16 bits)
  - `pop <operand>, push<operand>`
  - `inc <operand>, dec<operand>`
  - `Jcond <imme>,  int <imme>`

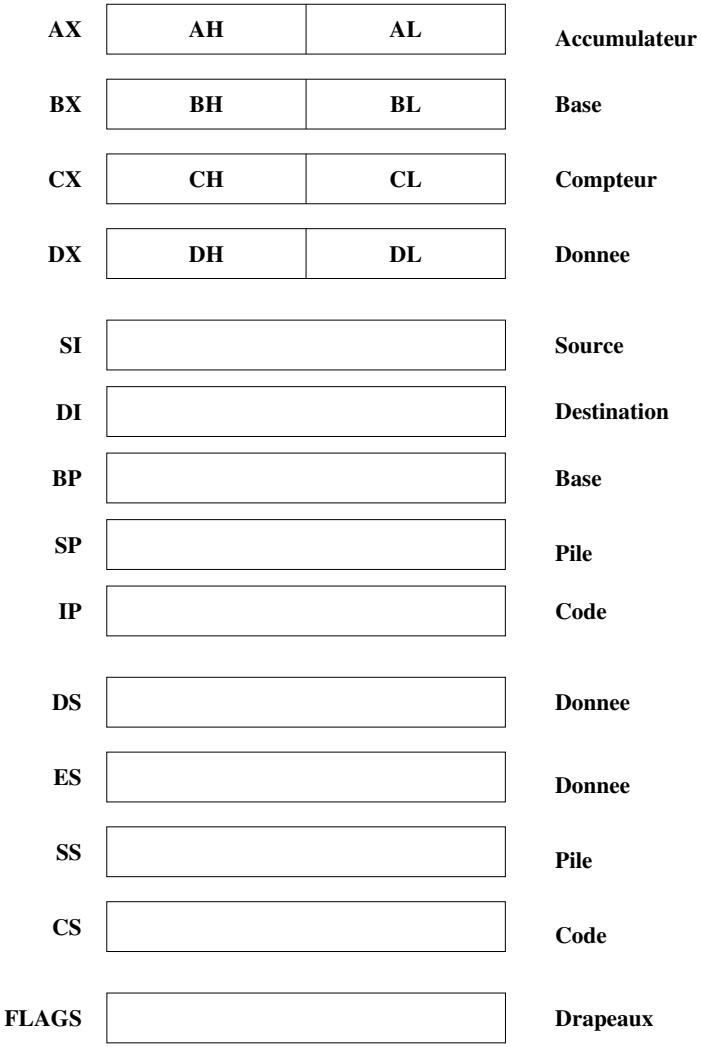


FIGURE 1: Registres du I8086

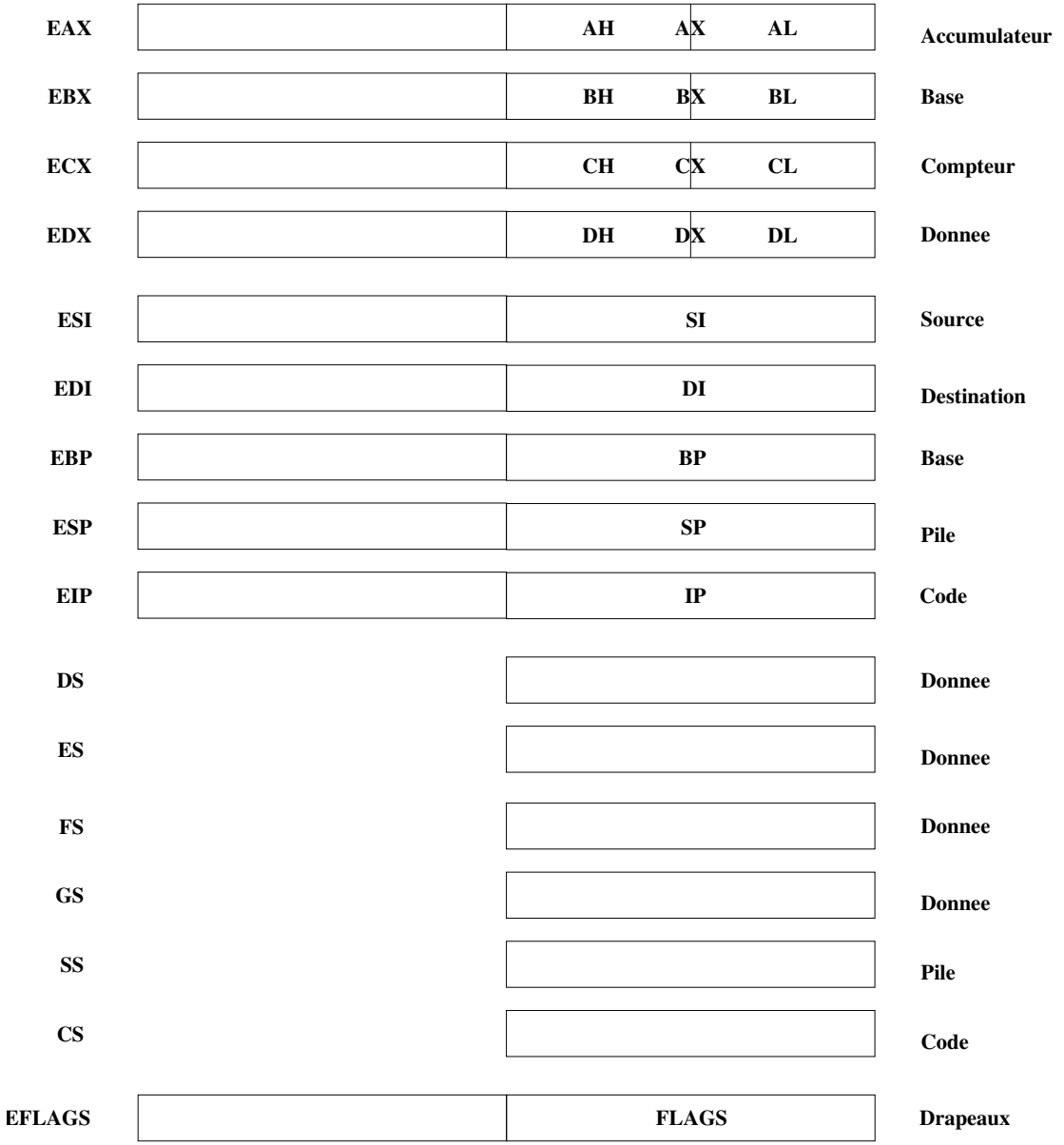
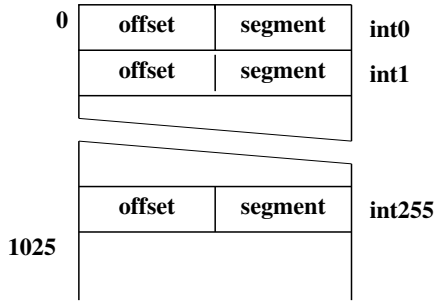


FIGURE 2: Registres du I386

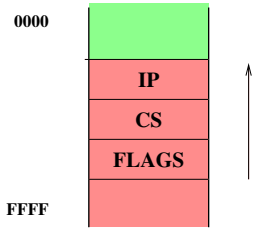
  - `jmp <depl>, jmp <imme:imme>, jmp <operand> (far)`
  - `call <depl>, call <imme:imme>, call <operand> (far)`
- Instruction à 2 opérandes (8 ou 16 bits)
  - reg $\Longleftarrow$ reg (ex: `mov AL, BH, add BX, AX`)
  - reg $\Longleftarrow$ immé (ex: `mov AL, 123, add BL, 12`)
  - reg $\Longleftarrow$ mem (ex: `mov AL, [DX], sub BX, [ES:BP+SI+12]`)
  - mem $\Longleftarrow$ reg (ex: `mov [EX+DI+12], AX, or BX, [12]`)
  - E/S $\Longleftarrow$ AX (`out <imme>,AX` ou `out DX, AX`)
  - AX $\Longleftarrow$ E/S (`in AX,<imme>` ou `in AX, DX`)
- Instructions spéciales
  - à 2 opérandes mémoire post-inc/décrémenté (DF=0/1)
    `movsb/w` $\rightarrow DS:[SI] \Longrightarrow ES:[DI]$
    `movsb/w ES:[DI],CS[SI]` $\rightarrow CS:[SI] \Longrightarrow ES:[DI]$
  - préffixes
    `rep` $\rightarrow$ CX!=0 (ex: rep movsb)
    `CS`, `ES`, `SS`, `DS` $\rightarrow$ segment (ex: CS mobw)
    `M16`, `M32`, $\rightarrow$ mode (ex: C32 xor ax,ax)

1.2.4 Interruptions

- Quelques exceptions non masquables.
- 1 seul niveau d'interruptions vectorisées
  - masquables (cli, sti)
  - 256 vecteurs (0 à 255)
- table des vecteurs des exceptions et interruptions toujours à l'adresse $\emptyset$



structure

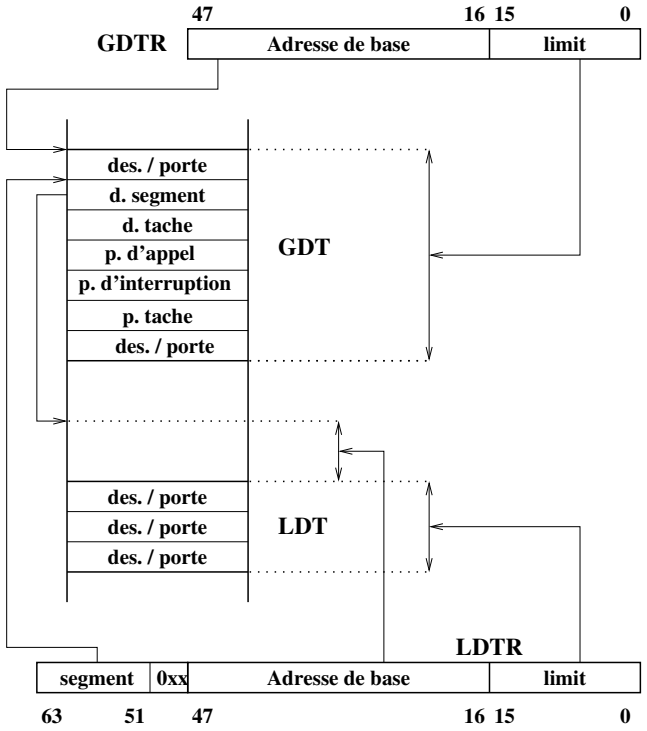


pile

1.3 Mode Protégé et 486

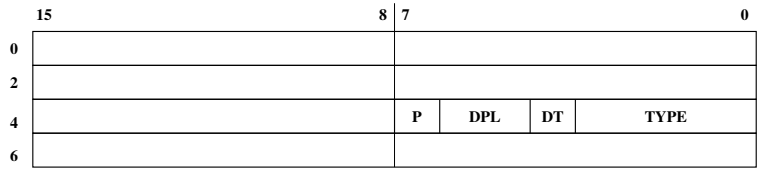
Fonctionnement en mode 32 bits (386 et suivant), il y a aussi un mode protégé 16 bits supporté à partir du 286.

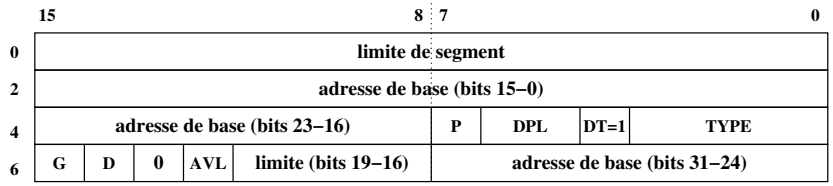
1.3.1 Table des descripteurs



1.3.2 Descripteurs de la GDT et de LDT

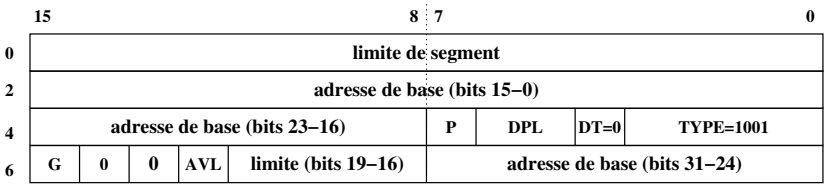
Le format général d'un descripteur est donné ci-dessous:





G : 0/1=o/4ko | **D** : mode 16/32 bits
DPL : Privilège (00 +, 3 -) | **DT** : 1
AVL : libre pour OS
TYPE : 1CRA/0EWA=segment de code/données
- **C** : exécutable par privilège supérieur.
- **R** : lisible
- **A** : accédé
- **E** : extensible vers le bas `[base-limit,base]`
- **W** : modifiable

FIGURE 3: Format d'un descripteur de segment



	31 — 0
0	retour arrière
4	ESP (CPL0)
8	SS (CPL0) | 0000
12	ESP (CPL1)
16	SS (CPL1) | 0000
20	ESP (CPL2)
24	SS (CPL2) | 0000
28	CR3 (pagination)
32	EIP
36	EFLAGS
40	EAX
44	ECX
48	EDX

	31 — 0	
52	EBX	
56	ESP	
60	EBP	
64	ESI	
68	EDI	
72	ES	0000
76	CS	0000
80	SS	0000
84	DS	0000
88	FS	0000
92	GS	0000
96	LDT	0000
100	T 0000	carte E/S

Le registre RT pointe sur La tâche courante.

FIGURE 4: Format d'un descripteur de tâche

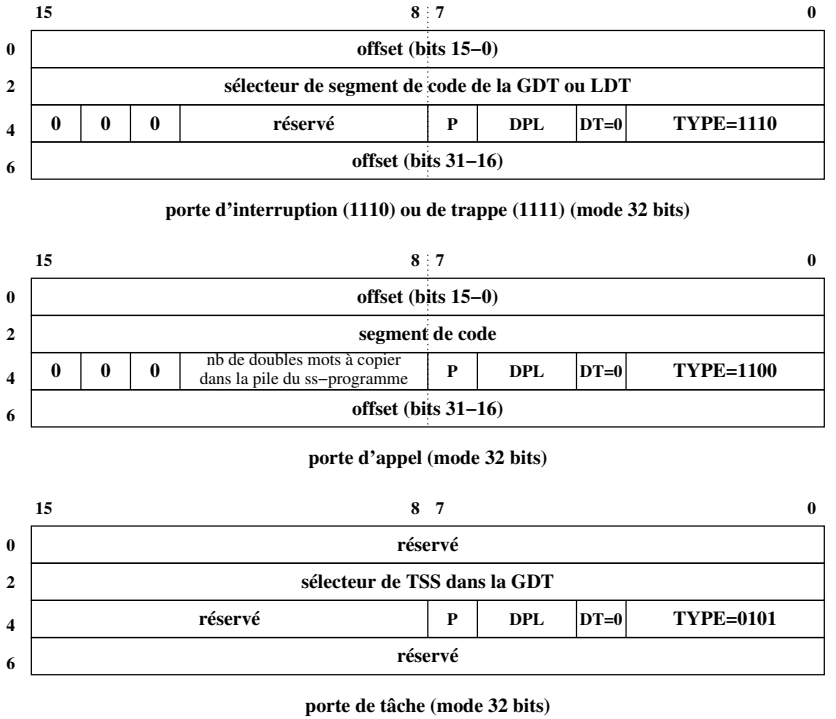


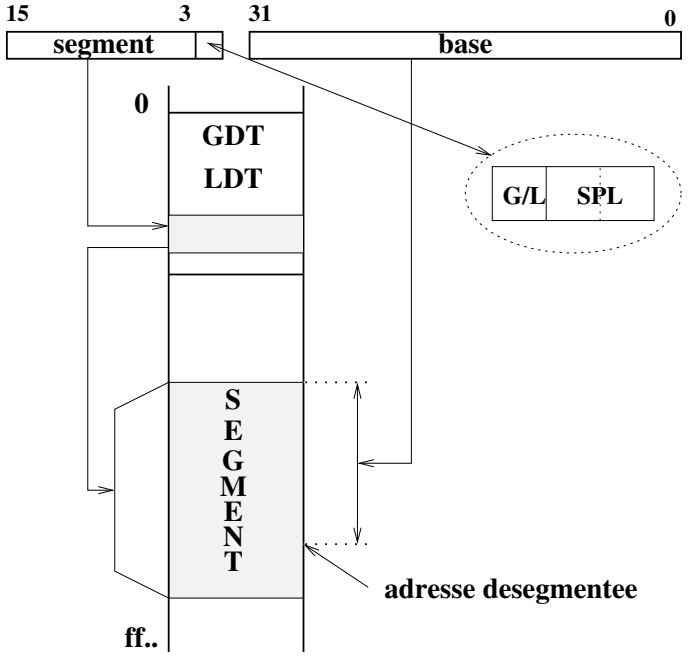
FIGURE 5: Format d'un descripteur de portes

P : Présent	**DPL** : Privilège (00 +, 3 -)
DT : 1/0=segment/système	**TYPE** : dépent de DT

Les principaux descripteurs sont présentés sur les figures 3, 4 et 5

1.3.3 Adresses segmentées

⟹ une adresse (32 bits) c'est un segment (16 bits) plus offset (32 bits)
⟹ DS:EAX



- Recherche le segment [seg_deb, seg_fin]
- seg_deb + base est l'adresse désegmentée
- Des exceptions sont générées:
  - l'entrée dans G/LDT n'existe pas.
  - l'entrée dans G/LDT n'est pas un segment.
  - l'adresse désegmentée n'est pas dans [seg_deb, seg_fin].
  - les privilèges (CPL) sont violés.
  - écriture/lecture/accessibilité violées.

1.3.4 Modes d'adressage

Mode 16 bits idem 8086

Mode 32 bit

- immédiat
- registre
- DS:(BASE [+ INDEX * fac] [+ dépl])
- DS:(dépl)
- SS:(EBP [+ INDEX * fac] [+ dépl])
- SS:(ESP [+ INDEX * fac] [+ dépl])

BASE: EAX, ECX, EBX, EDX, ESI, EDI
INDEX: EAX, ECX, EBX, EDX, EBP, ESI, EDI
CS, DS, ES, ES, FS, SS peuvent en général remplacer DS, SS.

1.3.5 Instructions

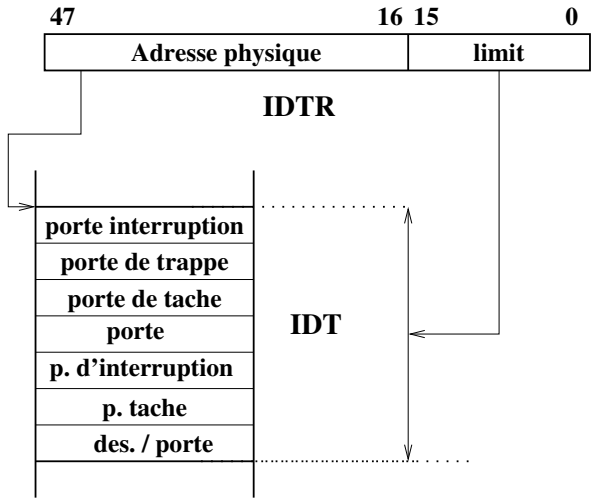
3 modes: 16 bits réel, 16 bits protégé, et 32 bits protégé

- Même instructions que 8086 mais étendues à:
  - calcul d'adresse sur 32 bits mov [EBX+ESI],AX
  - nouveaux modes d'adressage
    mov [ESI+EDI*4],BH).
  - calcul 32 bits movsb, movsw, movsd
    mov EAX,ESI
- Instructions spécifiques
  - load, store de nouveau registre (GDTR, LDTR).
  - set, clear des nouveaux flags.
- Préfixes:
  - M16: instruction 16 bits quand on est en 32 bits.
  - M32: instruction 32 bits quand on est en 16 bits.
  - rep
  - CS, DS, ES, GS, FS, SS

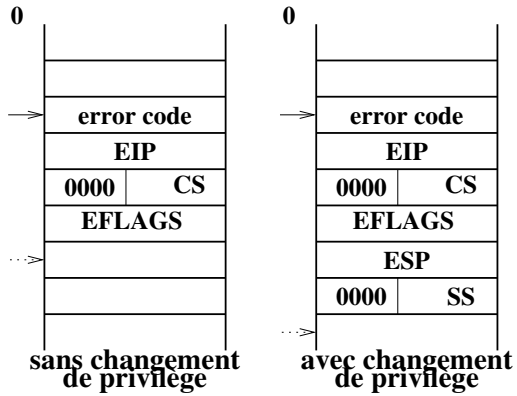
1.3.6 Interruptions

- Une vingtaine d'exceptions non masquables.
- 1 seul niveau d'interruptions vectorisées
  - masquables (cli, sti)
  - 256 vecteurs (0 à 255)

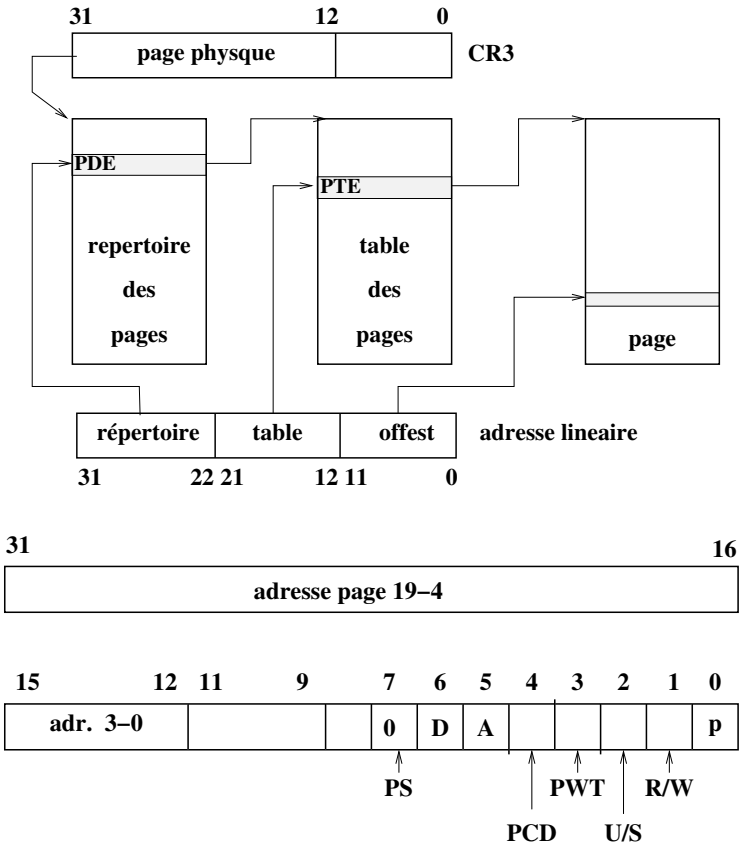
– ridt donne l'adresse de la table et sa taille en octet (48 bits).
– table des vecteurs des exceptions et interruptions



– pile: le code d'erreur n'est pas empilé pour les traps et interruptions standards



1.3.7 Pagination



P 1/0:chargée/absente

R/W protection en écriture

PCD désactiver le cache pour la page

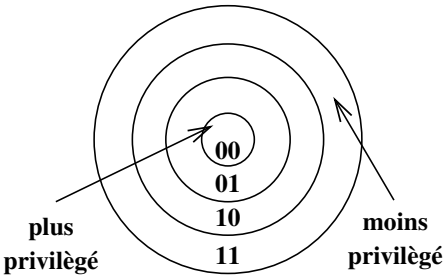
A 1=page accèdée

D 1=sale

AVL libre pour OS

1.4 Système de protection

– 4 niveaux de privilège (00 le plus prvilègié, 11 le moins)



– CPL de CS donne le privilège courant.
– Protection mémoire au niveau segment
une tâche ne peut accèder des segments plus privilègiés.
– Des mécanismes pour passer d'un niveau à l'autre

1. Commutation de tâche (jump/call/int sur une porte de tâche)
2. Interruption (externe ou int)
3. jmp, call sur une porte
4. iret (retour)

Les sections suivantes donne le comportement du processeur pour les instructions `int`, `jump SEG:OFFSET` et `call SEG:OFFSET`. Celui-ci dépend du type du descripteur pointé par `SEG`.

1.4.1 Le descripteur est un segment de données

erreur

1.4.2 Le descripteur est un segment de code

autorisation

privilège		docile	action
=	DPL = CPL		saut
-	DPL > CPL		erreur
+	DPL < CPL	non	erreur
+	DPL < CPL	oui	saut sans changement de privilège

opération

CS:EIP $\Longleftarrow$ CS:EIP de l'instruction
retour par ret pour call.

1.4.3 Le descripteur est une tâche ou une porte de tâche

autorisation

privilège		action
-	DPL $\geq$ CPL	commutation
+	DPL < CPL	erreur

opération

1. sauvegarde des registres dans le TSS courant.
2. mise à jour du lien arrière dans le TSS cible (pour call).
3. restauration des registres à partir du TSS cible.
4. charge TR avec le sélecteur de la tâche cible.
5. charge LDTR avec les données du TSS la tâche cible.
6. charge PDBR (CR3) avec les données du TSS la tâche cible.
7. donne le contrôle à la tache cible.

notes EIP destination n'est pas utilisé, le niveau de privilège est celui de la tâche cible, retour par iret pour call.

1.4.4 Le descripteur est une porte d'appel

La porte d'appel contient un segment (CS_{pa}), un offset ($offset_{pa}$) et un nombre de paramètres ($param_{pa}$). Le segment CS_{pa} est décrit par un descripteur de segment avec un DPL ($DPL_{cs\,cible}$) et qui indique si le segment est docile ou pas.

autorisation

privilège		doc.	action
DPL < CPL			
			erreur
DPL $\geq$ CPL			
=	$DPL_{cs\,cible}$ = CPL		saut
-	$DPL_{cs\,cible}$ > CPL		erreur
+	$DPL_{cs\,cible}$ < CPL	oui	saut sans changement de privilège
+	$DPL_{cs\,cible}$ < CPL	non	jmp: erreur; call: saut avec changement de privilège

opération

1. changement de pile (pile du niveau de privilège $DPL_{cs\,cible}$).
2. empilement de SS (32 bits), ESP
3. empilement des paramètres copiés de l'ancienne pile
4. empilement de CS (32 bits), EIP
5. saut

notes EIP destination n'est pas utilisé. Retour par textcolorbule`ret N`.

1.4.5 Le descripteur est une porte de trap ou d'interruption

La porte de trap ou d'interruption contient un segment (CS_{pi}), un offset ($offset_{pi}$) et un nombre de paramètres ($param_{pi}$). Le segment CS_{pi} est décrit par un descripteur de segment avec un DPL ($DPL_{cs\,cible}$) et qui indique si le segment est docile ou pas.

autorisation

privilège		doc.	action
DPL < CPL			
			erreur si trap
interruption ou trap avec DPL ≥ CPL			
=	$DPL_{cs\,cible}$ = CPL		saut
-	$DPL_{cs\,cible}$ > CPL		erreur
+	$DPL_{cs\,cible}$ < CPL	oui	saut sans changement de privilège
+	$DPL_{cs\,cible}$ < CPL	non	saut avec changement de privilège

opération sans changement de privilège

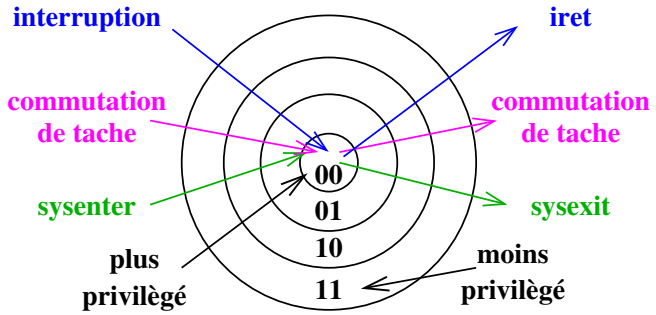
1. empilement de EFLAG
2. empilement de CS (32 bits), EIP
3. empilement d'un code d'erreur si exception.
4. saut

opération avec changement de privilège

1. changement de pile (pile du niveau de privilège $DPL_{cs\,cible}$).
2. empilement de SS (32 bits), ESP
3. empilement de EFLAG
4. empilement de CS (32 bits), EIP
5. empilement d'un code d'erreur si exception.
6. saut

notes EIP destination n'est pas utilisé, retour par iret après avoir dépilé le code d'erreur.

1.4.6 Changements de privilège usuels



1.5 Changement de privilège par sysenter/exit

- changement de privilège par descripteur jugé lent.
- introduites par AMD `syscall/sysret`
- sous linux: appel système des programmes statiques utilisent les trappes, les programmes dynamique utilisent `sysenter` et `sysexit`
- 64-bit Model Specific Registers (MSR)
  instruction protègée:
  WRMSR $\Longrightarrow MSR[ECX] = EDX : EAX$
  quelques registres:
  `SYSENTER_CS_MSR = 0x174`
  `SYSENTER_ESP_MSR = 0x175`
  `SYSENTER_EIP_MSR = 0x176`
- Instruction SYSENTER (non protègée)
  `CS` $\leftarrow MSR[SYSENTER\_CS\_MSR]$ au niveau de privilège 0
  `EIP` $\leftarrow MSR[SYSENTER\_EIP\_MSR]$
  `SS` $\leftarrow 8 + MSR[SYSENTER\_CS\_MSR]$
  `ESP` $\leftarrow MSR[SYSENTER\_ESP\_MSR]$

- Instruction SYSEXIT
  `CS` $\leftarrow 16 + MSR[SYSENTER\_CS\_MSR]$ au niveau de privilège 3
  `EIP` $\leftarrow EDX$
  `SS` $\leftarrow 24 + MSR[SYSENTER\_CS\_MSR]$
  `EIP` $\leftarrow ECX$
- Exemple d'utilisation:
  - Initialisation

```
stack:
    ...
gdt:
    ...
    # entry N
    0x...       # segment text système (CS)
    0x...       # segment data système (SS, DS, ES, ...)
    0x...       # segment text user    (CS)
    0x...       # segment data user    (SS, DS, ES, ...)

main:
   ...
   # init MSR[SYSENTER_CS_MSR] to N
   ...
   # init MSR[SYSENTER_EIP_MSR] to syscall
   ...
   # init MSR[SYSENTER_ESP_MSR] to stack
   ...
```

  - Utilisation

mode non privilègié

```
    push ECX
    push EDX
    move ESP,ECX
    move $reprise,EDX
    sysenter
reprise:
    pop EDX
    pop ECX
```

mode privilègié

```
syscall:
    # récupérer user-ESP user-EIP
    # sauver les registres
    push ...
    # traitement
    ...
    # restaurer les registres
    move user-ESP,ECX
    move user-EIP,EDX
    sysexit
```

- Limitation
  - `SSsys=CSsys+8`, `CSusr=CSsys+16` et `SSusr=CSsys+24` $\Longrightarrow$ `CSsys`, `SSsys`, `CSusr` et `SSusr` contigüs dans la G/LDT.
  - Ne peux fonctionner que entre les niveaux 3 et 0.

1.6 Exercices

1.1 Ecrire une fonction assembleur en mode réel qui écrit une chaîne de caractères terminée par 0 dont l'adresse est dans DX. On ne modifiera aucun registre et on dispose d'une fonction "*ecrit*" qui affiche le caractère qui se trouve dans DH et qui ne modifie aucun registre.

1.2 Même exercice que précédemment mais le nombre de caractères à écrire est contenu dans le registre AH.

1.3 Ecrire la fonction qui copie les AX octets de l'adresse DS:BX à l'adresse 0x54321. La fonction ne modifiera pas les registres, on supposera que les zones mémoire source et destination ne se chevauchent pas.

1. D'abord à la 68000.
2. Puis à l'Intel.

Quelle est la copie faite si DS:BX+AX sort du segment DS?

1.4 Même question que précédemment mais elle marche même si DS:BX+AX dépasse la limite du segment. On peut utiliser la fonction précédente.

1.5 On appelle processus une entité qui peut tourner totalement indépendamment des autres. Elle posséde 0.5 MO de code, 0.5 MO de données, 0.5 MO de pile.
Notre machine possède 8 MO de mémoire vive, notre système contient 4 processus: le processus 0 est le sytème d'exploitation (degré de privilège de 0) les autres ont un degré de privilège de 3. Les commutations n'ont lieu que entre le processus système et les processus utilisateurs.
Le but de cet exercice est d'organiser la machine en n'utilisant que la segmentation du I486.

1. Précisez la signification de "totalement indépendamment des autres".
2. Quel est l'espace virtuel d'un processus? Complètez la figure 6.a.
3. Placez les 4 processus en mémoire, ainsi que les tables nécessaires pour la gestion des processus. On complètera les figures 6.b et 6.c.
4. Indiquez de façon précise le contenu de la GDT (figure 7), et des TSS (figure **??**).
   (a) Comment le processus système commute-t-il sur un processus utilisateur. Que se passe-t-il?
   (b) Indiquer comment un processus utilisateur commute sur le processus système. Que se passe-t-il?
5. Sur cette organisation comment gèreriez vous les interruptions?
6. Maintenant on a 6 processus et un disque de swap de 100 MO.
   (a) Tous les processus sauf le 0, exécutent le même programme.

(b) Les processus exécutent des programmes différents. Donnez l'algorithme du système en supposant que le système fait tourner les processus régulièrement dans l'ordre suivant: 1, 2, 3, 4, 5, 1, 2, ... Comment le système peut-il accèder aux segments utilisateurs?

1.6 On appelle processus une entité qui peut tourner totalement indépendamment des autres. Elle posséde 0.5 MO de code, 0.5 MO de données et 0.5 MO de pile.
Notre machine possède 8 MO de mémoire vive, notre système possède 4 processus: le processus 0 est le sytème d'exploitation (degré de privilège de 0) les autres ont un degré de privilège de 3. De plus:
- Les commutations n'ont lieu que entre le processus système et les processus utilisateurs.
- Les processus utilisateurs ont leur code à l'adresse 0, leur données à l'adresse 16 M, et leur pile à l'adresse 32 M.
- Le processus système a son code, ses données et sa pile contigüs à l'adresse 64 M.

Le but de cet exercice est d'organiser la machine en n'utilisant que la pagination (et un peu la segmentation) du I486.

1. Donnez une représentation graphique de l'espace virtuel d'un processus utilisateur et du processus système (figure 8.a).
2. Placez les 4 processus en mémoire, ainsi que les tables nécessaires pour la gestion des processus (figures 8.b et 8.c).
3. Détaillez la GDT, les TSS et les répertoires de pages (figures 9, 10 et 11).
4. Donnez la séquence d'instructions pour commuter d'un processus utilisateur vers le processus système. Donnez les actions nécessaires pour commuter du processus système vers le processus 3.
5. Maintenant on a 6 processus et un disque de swap de 100 MO.
   (a) Tous les processus sauf le 0, exécutent le même programme.
   (b) Les processus exécutent des programmes différents. Donnez l'algorithme du système en supposant que le système fait tourner les processus régulièrement les uns après les autres (ordre: 1, 2, 3, 4, 5, 1, 2, ...) et que les processus sont swappés entierement sur le disque.
6. On a 6 processus qui exécutent des programmes différents, et 100 MO de swapp.
   On a dans l'espace système la fonction `void pgfault(void* a)` qui est appelée à chaque défault de page. *a* contient alors l'adresse désegmentée fautive. Soit l'adresse fautive était valide, dans ce cas la page est chargée en mémoire et le processus qui a causé le défault de page est relancé. Soit l'adresse fautive est invalide et dans ce cas le processus est marqué fautif et la main est donnée au processus 0.
   (a) Donnez l'algorithme du système en supposant que le système fait tourner les processus régulièrement les uns après les autres (ordre: 1, 2, 3, 4, 5, 1, 2, ...) et que le chargement des page est laissé à `pgfault`.
   (b) Donnez l'algorithme de pgfault.
   (c) Indiquez comment il faut configurer le système pour lancer pgfault. L'exeception "page fault" est la $15^{ième}$, et sur la pile le processeur à empiler sur 32 bits dans l'ordre: SS, ESP, EFLAGS, CS, EIP, "code erreur". De plus il a mis dans le registre CR2 l'adresse désegmentée fautive.

processus 0

	31 … 0
0	retour arrière
4	ESP-0=
8	SS-0= | 0000
12	ESP1=
16	SS-1= | 0000
20	ESP2=
24	SS-2= | 0000
28	réservé
32	EIP=
36	EFLAGS=
40	EAX=
44	ECX=
48	EDX=

	31 … 0	
52	EBX=	
56	ESP=	
60	EBP=	
64	ESI=	
68	EDI=	
72	ES=	0000
76	CS=	0000
80	SS=	0000
84	DS=	0000
88	FS=	0000
92	GS=	0000
96	LDT=	0000
100	T 0000	carte E/S

processus 3

	31 … 0
0	retour arrière
4	ESP-0=
8	SS-0= | 0000
12	ESP1=
16	SS-1= | 0000
20	ESP2=
24	SS-2= | 0000
28	réservé
32	EIP=
36	EFLAGS=
40	EAX=
44	ECX=
48	EDX=

	31 … 0	
52	EBX=	
56	ESP=	
60	EBP=	
64	ESI=	
68	EDI=	
72	ES=	0000
76	CS=	0000
80	SS=	0000
84	DS=	0000
88	FS=	0000
92	GS=	0000
96	LDT=	0000
100	T 0000	carte E/S

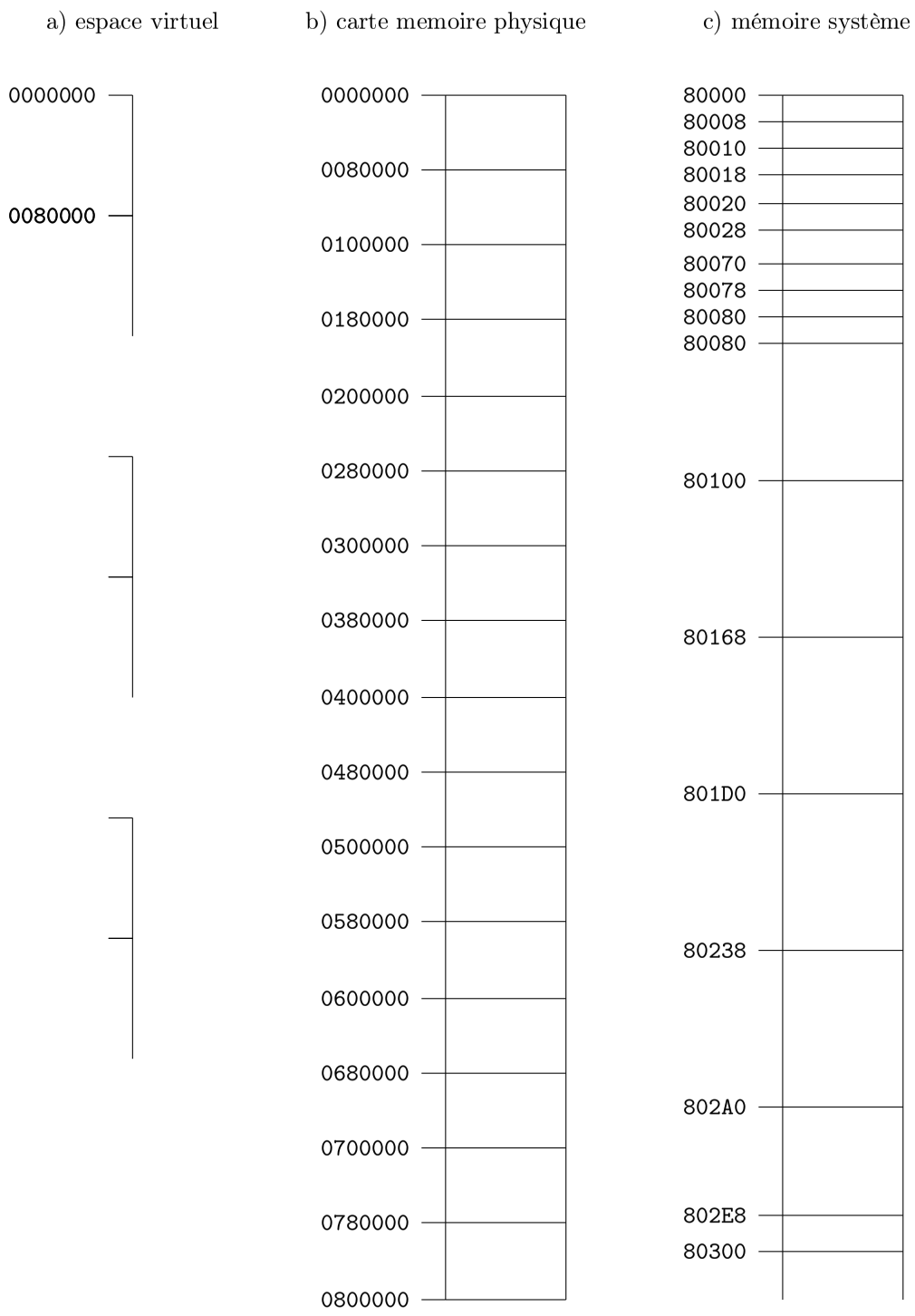


FIGURE 6: Implantation avec les ségments.

0	lim[15:0]=			
2	base[15:0]=			
4	base[23:16]=		P/DPL/DT=	type=
6	G/D/0/AVL=	lim[19:16]=	base[31:24]=	

0	lim[15:0]=			
2	base[15:0]=			
4	base[23:16]=		P/DPL/DT=	type=
6	G/D/0/AVL=	lim[19:16]=	base[31:24]=	

0	lim[15:0]=			
2	base[15:0]=			
4	base[23:16]=		P/DPL/DT=	1CRA=
6	G/D/0/AVL=	lim[19:16]=	base[31:24]=	

0	lim[15:0]=			
2	base[15:0]=			
4	base[23:16]=		P/DPL/DT=	0EWA=
6	G/D/0/AVL=	lim[19:16]=	base[31:24]=	

0	lim[15:0]=			
2	base[15:0]=			
4	base[23:16]=		P/DPL/DT=	0EWA=
6	G/D/0/AVL=	lim[19:16]=	base[31:24]=	

0	lim[15:0]=			
2	base[15:0]=			
4	base[23:16]=		P/DPL/DT=	type=
6	G/D/0/AVL=	lim[19:16]=	base[31:24]=	

0	lim[15:0]=			
2	base[15:0]=			
4	base[23:16]=		P/DPL/DT=	1CRA=
6	G/D/0/AVL=	lim[19:16]=	base[31:24]=	

0	lim[15:0]=			
2	base[15:0]=			
4	base[23:16]=		P/DPL/DT=	0EWA=
6	G/D/0/AVL=	lim[19:16]=	base[31:24]=	

0	lim[15:0]=			
2	base[15:0]=			
4	base[23:16]=		P/DPL/DT=	0EWA=
6	G/D/0/AVL=	lim[19:16]=	base[31:24]=	

...

0	lim[15:0]=			
2	base[15:0]=			
4	base[23:16]=		P/DPL/DT=	0EWA=
6	G/D/0/AVL=	lim[19:16]=	base[31:24]=	

FIGURE 7: GDT de l'implantation avec les ségments.

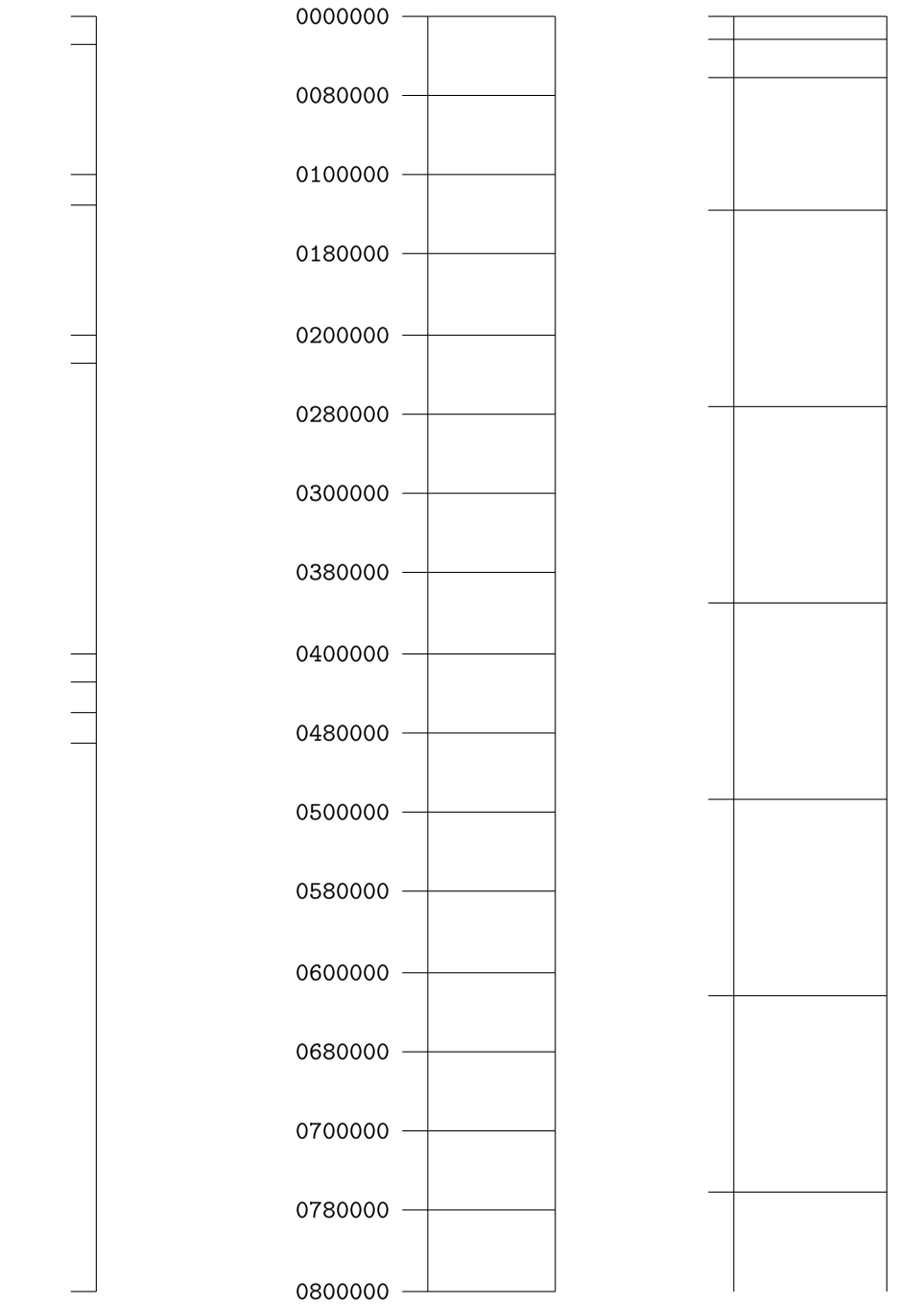


FIGURE 8: Implantation avec la pagination.

0	lim[15:0]=			
2	base[15:0]=			
4	base[23:16]=		P/DPL/DT=	type=
6	G/D/0/AVL=	lim[19:16]=	base[31:24]=	

0	lim[15:0]=			
2	base[15:0]=			
4	base[23:16]=		P/DPL/DT=	1CRA=
6	G/D/0/AVL=	lim[19:16]=	base[31:24]=	

0	lim[15:0]=			
2	base[15:0]=			
4	base[23:16]=		P/DPL/DT=	0EWA=
6	G/D/0/AVL=	lim[19:16]=	base[31:24]=	

0	lim[15:0]=			
2	base[15:0]=			
4	base[23:16]=		P/DPL/DT=	1CRA=
6	G/D/0/AVL=	lim[19:16]=	base[31:24]=	

0	lim[15:0]=			
2	base[15:0]=			
4	base[23:16]=		P/DPL/DT=	0EWA=
6	G/D/0/AVL=	lim[19:16]=	base[31:24]=	

0	lim[15:0]=			
2	base[15:0]=			
4	base[23:16]=		P/DPL/DT=	type=
6	G/D/0/AVL=	lim[19:16]=	base[31:24]=	

0	lim[15:0]=			
2	base[15:0]=			
4	base[23:16]=		P/DPL/DT=	type=
6	G/D/0/AVL=	lim[19:16]=	base[31:24]=	

0	lim[15:0]=			
2	base[15:0]=			
4	base[23:16]=		P/DPL/DT=	type=
6	G/D/0/AVL=	lim[19:16]=	base[31:24]=	

0	lim[15:0]=			
2	base[15:0]=			
4	base[23:16]=		P/DPL/DT=	type=
6	G/D/0/AVL=	lim[19:16]=	base[31:24]=	

FIGURE 9: GDT de l'implantation avec la pagination.

processus 0

	31 — 0
0	retour arrière
4	ESP-0=
8	SS-0= | 0000
12	ESP1=
16	SS-1= | 0000
20	ESP2=
24	SS-2= | 0000
28	réservé
32	EIP=
36	EFLAGS=
40	EAX=
44	ECX=
48	EDX=

	31 — 0
52	EBX=
56	ESP=
60	EBP=
64	ESI=
68	EDI=
72	ES= | 0000
76	CS= | 0000
80	SS= | 0000
84	DS= | 0000
88	FS= | 0000
92	GS= | 0000
96	LDT= | 0000
100	T | 0000 | carte E/S

processus 3

	31 — 0
0	retour arrière
4	ESP-0=
8	SS-0= | 0000
12	ESP1=
16	SS-1= | 0000
20	ESP2=
24	SS-2= | 0000
28	réservé
32	EIP=
36	EFLAGS=
40	EAX=
44	ECX=
48	EDX=

	31 — 0
52	EBX=
56	ESP=
60	EBP=
64	ESI=
68	EDI=
72	ES= | 0000
76	CS= | 0000
80	SS= | 0000
84	DS= | 0000
88	FS= | 0000
92	GS= | 0000
96	LDT= | 0000
100	T | 0000 | carte E/S

FIGURE 10: TSS de l'implantation avec la pagination.

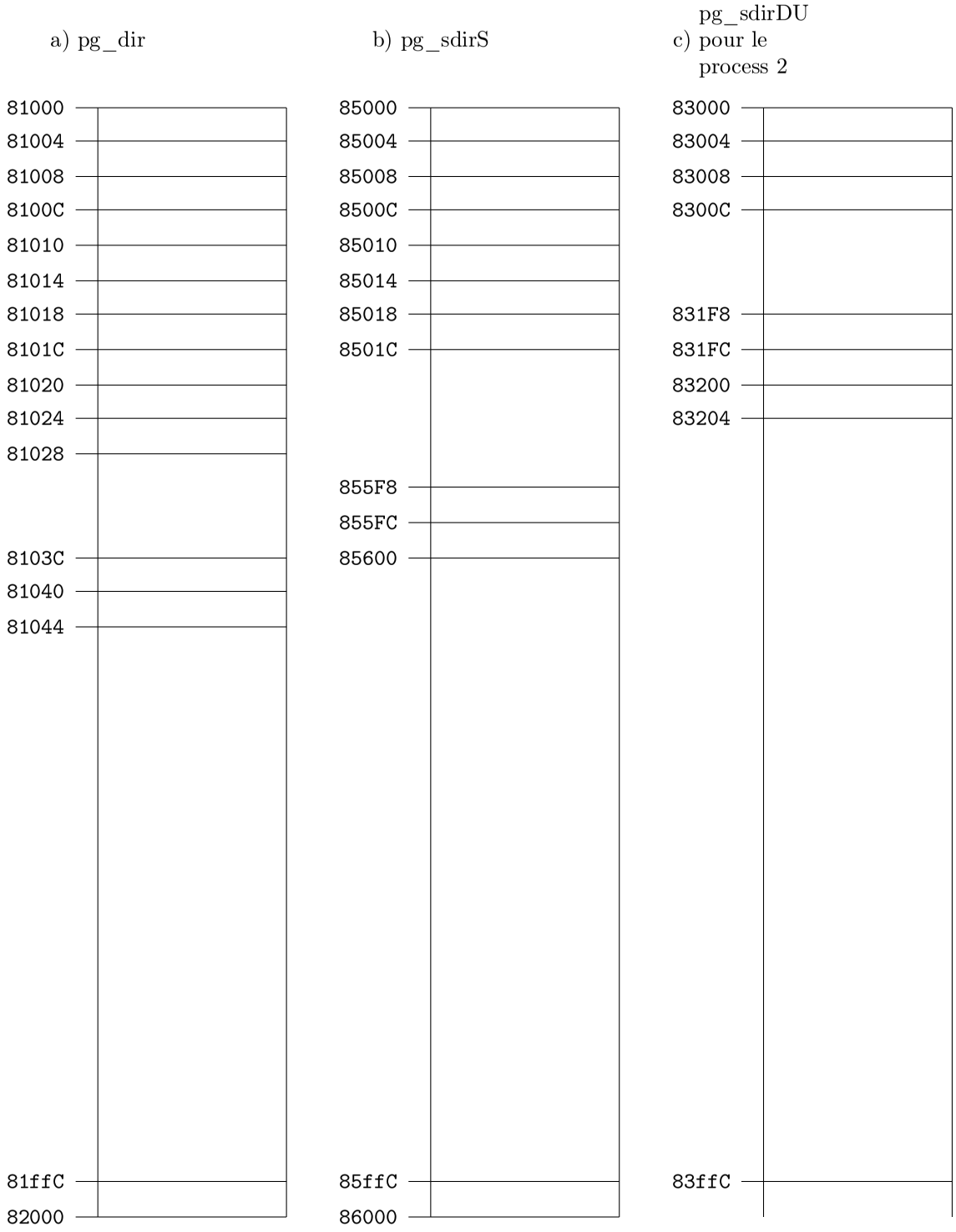


FIGURE 11: Implantation des tables de pages.

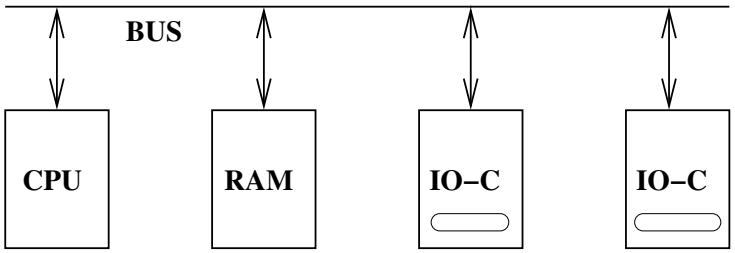


FIGURE 12: Architecture d'un ordinateur

2 Introduction aux systèmes

Pré-requis: Lisez les pages de "*man*" de "*ar*" et "*tar*" et créez quelques bibliothèques et archives.

2.1 Système d'exploitation

La figure 12 représente l'architecture macroscopique d'un ordinateur. Le rôle du système d'exploitation est

- de donner l'accès aux ressources physiques, de les organiser,
- dans le cas de système multi-tâches et multi-utilisateurs de partager les ressources physiques,
- d'assurer des protections entre les différentes tâches et les différents utilisateurs.

Les qualités d'un système sont la fiabilité, l'intégrité, l'efficacité et la compatibilité. La figure 13 représente les différentes couches logicielles qui entourent le matériel. On y trouve

- le moniteur qui est un "petit" programme en ROM qui prend la main au reset de la machine.
- le noyau, il offre un ensemble de primitives de bas niveau **incontournables** pour accèder aux ressources physiques.
- le système, c'est un ensemble de programmes, de librairies qui sont l'interface utilisateur aux primitives du noyau.

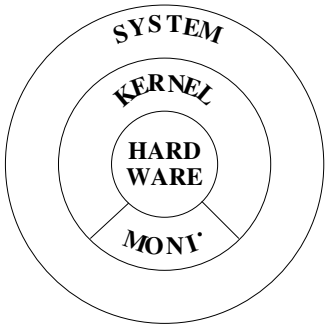
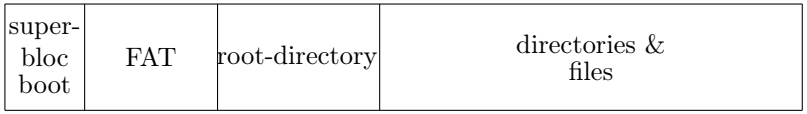


FIGURE 13: Organisation système

créer fichier	dir, name, att	ajoute un fichier vide de nom "name" avec les attributs "att" dans le répertoire "dir"
détruire fichier		
ouvrir fichier		
fermer fichier		
lire fichier		
écrire fichier		
positionner fichier		

TABLE 1: Opérations d'un système de fichiers.



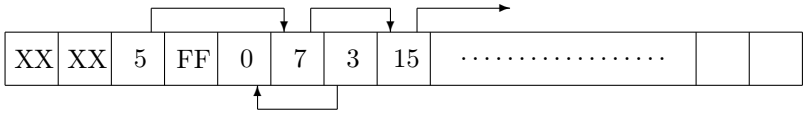
a) structure générale d'un volume

adr		taille
0	jump boot	3
3	nom du fabricant	8
11	nombre de bytes par secteur	2
13	nombre de secteurs par cluster	1
14	secteurs réservés	2
16	nombre de copies de FAT	1
17	nombre d'entrées dans root-dir	2
19	nombre de secteurs du volume	2
21	géométrie du disque, ...	41
62	routine de boot	482

b) super-bloc/boot

adr		taille
0	nom	8
8	extension	3
11	attribut	1
22	date/heure	4
26	1^{er} cluster	2
28	taille en bytes	4

c) entrée de répertoire



- la partie "directories & files" est découpée en N clusters numérotés de 0 à N-1.
- la FAT est un tableau de mots de 16 bits.
- la valeur 0 dans une case de la FAT correspond à un cluster libre.
- la valeur FFFF dans une case de la FAT correspond à fin de fichier.
- une autre valeur dans une case de la FAT correspond au numéro du cluster suivant.

d) structure de la FAT

FIGURE 14: Système de fichiers MSDOS 16 bits.

2.2 Système de fichiers

Le (ou les) système de fichiers est une pièce maîtresse du système d'exploitation. Il permet soit la sauvegarde permanente de données diverses et variées, soit le stockage temporaire de grandes quantités de données qui ne tiennent pas en mémoire centrale. L'organisation usuelle d'un système de fichiers est une arborescence, les noeuds terminaux étant les fichiers (dont ceux des utilisateurs), les noeuds non terminaux étant des répertoires (fichiers gérés par le système). La table 1 énumère les opérations élémentaires que fournit un système.

Le support naturel d'un système de fichiers est le disque dur. Un disque dur peut être vu dans une première approche comme un tableau de secteurs (blocs de 0.5, 1, 2, 4 ko). Le noyau chaîne entre eux les secteurs du disque pour créer une structure de données qui reflète la structure arborescente du système de fichiers. La figure 14 représente cette structure de données pour le système MSDOS 16 bits.
On appelle formatage logique la création d'une structure de données vide sur un disque dur. Il est le préalable à toute opération sur les fichiers.

2.3 Disques durs

Un disque dur est composé de plusieurs plateaux solidaires tournant autour d'un axe de rotation orthogonal aux plateaux et passant par leurs centres. Chaque plateau est muni d'une tête de lecture/écriture mobile. Les têtes sont solidaires et peuvent se déplacer sur un rayon des plateaux. On appelle:

head$_i$ La i^{ème} tête de lecture, les têtes étant numérotées de 0 à NH-1.

piste$_i$ Une piste est la surface passant sous 1 tête immobile. C'est donc une couronne d'un plateau. Les pistes sont numérotées de 0 à NP-1.

cylinder$_i$ Un cylindre regroupe l'ensemble des NH pistes accessibles pour une position des têtes. Ils sont numérotés de 0 à NC-1.

sector$_i$ Une piste est découpée en NPS secteurs. C'est l'unité de base de lecture et d'écriture du disque. Ils sont numérotés de 0 à NS-1.

Pour accèder à un secteur il faut fournir suivant le contrôleur de disque soit (cylindre, tête, secteur), soit (piste, secteur), soit (secteur). Ce dernier étant appelé l'adressage linéaire. Avant de pouvoir être lu ou écrit, un disque doit être formaté physiquement. Les paramètres principaux sont le nombre de cylindres, le nombre de secteurs par piste, la taille du secteur. Lors de ce formatage, le contrôleur de disque:

– écrit une entête de repérage devant chaque secteur,
– teste les secteurs,
– établit une table de redirection des secteurs défectueux vers une piste de réserve.

2.4 Exercices

Exercices sur les systèmes d'exploitation

OS.1 Illustrer par des exemples le rôle et les qualités d'un système d'exploitation. On explicitera comment assurer l'intégrité mémoire de 2 tâches sur un I386.

OS.2 Placer sur le schéma de la figure 13 (BIOS, DOS, command.com, dir, edit) pour un PC sous MSDOS, (BIOS, vmunix, shell, libc.a, ls, vim) pour un PC sous UNIX.

OS.3 Donner le moniteur minimum, pour pouvoir initialiser et démarrer une machine dans les configurations suivantes:

1. Une carte ethernet et un disque dur.
2. Un disque dur et un lecteur de disquette.

OS.4 Donner l'agorithme général d'une fonction du noyau quelconque comme par exemple:
int une_fonction(int c, void* data, void* result)

OS.5 Sur la figure 15 sont données quelques fonctions système du BIOS. Comment les exécute-t-on? Pourquoi un tel mécanisme?
Écrire les quelques lignes d'assembleur qui écrivent "salut" à la position courante du curseur et repositionnent le curseur au début de la ligne suivante.
Écrire les quelques lignes d'assembleur qui lisent les secteurs 1200 et 1201 sur le second disque en 9A010 et qui en cas d'erreur branchent au label "error". On supposera que le disque a 10 têtes et 100 cylindres de 10 secteurs.

Exercices sur les disques durs et les systèmes de fichiers

Ecriture d'un caractère INT=10, FCT=0E	
entrées	AH=0E
sorties	AL= code ASCII du caractère BL= couleur du premier plan pas de sortie

Information d'un disque INT=13, FCT=08	
entreées	AH=08 DL=disque (80,81)
sorties	Carry=1 Erreur DL= Nombre de disques connectés DH= Nombre de têtes (0 à N-1) CH= Numéro cylindre max (bits [7:0]) CL[5:0]= Numéro secteur max CL[7:6]= Numéro cylindre max[9:8]

Lecture d'un disque INT=13, FCT=02	
entrées	AH=02 AL= nombre de secteurs DH= numéro de la tête (0 à N-1) DL=disque (0,1,80,81) CH= numéro du cylindre (0 à N-1) (bit[7:0]) CL[7:6]= numéro du cylindre (bits[9:8]) CL[5:0]= numéro du secteur (1 à N)
sorties	Carry=1 Erreur Carry=0 blocs copiés en ES:BX

FIGURE 15: Quelques fonctions du BIOS.

FS.1 Complèter le tableau des opérations sur un système de fichiers.

FS.2 Pour le système de fichiers msdos (figure 14) donnez

1. La taille maximum d'un fichier.
2. Le nombre d'accès disque et la complexité au meilleur et pire cas pour lire le n^{ième} secteur d'un fichier ouvert.
3. L'étendue des dégâts pour un secteur perdu.

FS.3 Donner les formules permettant de passer d'un adressage linéaire à un adressage (cylindre, tête, secteur) et (piste, secteur).

FS.4 Les tableaux ci-dessous donnent la structure du premier secteur d'un disque pour le BIOS.

1. Quelle est l'organisation d'un disque?
2. Quel est le rôle des partitions?
3. Donner l'algorithme qu'exécute le BIOS au démarrage.
4. Donner l'algorithme standard du master-boot.

premier secteur

000 →	Master-boot
1BE →	partition 1
1CE →	partition 2
1DE →	partition 3
1EE →	partition 4
1FE →	55 AA

structure des partitions

	signification	byte
000 →	00=non active 80=boot	1
001 →	debut: tête	1
002 →	debut: secteur-cylindre	2
004 →	00= non utilisée 01= DOS 12bits 04= DOS 16bits 83= linux-native 82= linux-swap	1
005 →	fin: tête	1
006 →	fin: secteur-cylindre	2
008 →	sect du super-bloc	4
00C →	taille en secteurs	4

FS.5 Classer les commandes suivantes:

```
fdformat /dev/fd0h1440 | ar r /dev/fd0 gnat bee gnu
cp gnat /dev/fd0       | tar cvf /dev/fd0 gnat bee gnu
mformat a:             | mkfs.ext2 /dev/fd0
```

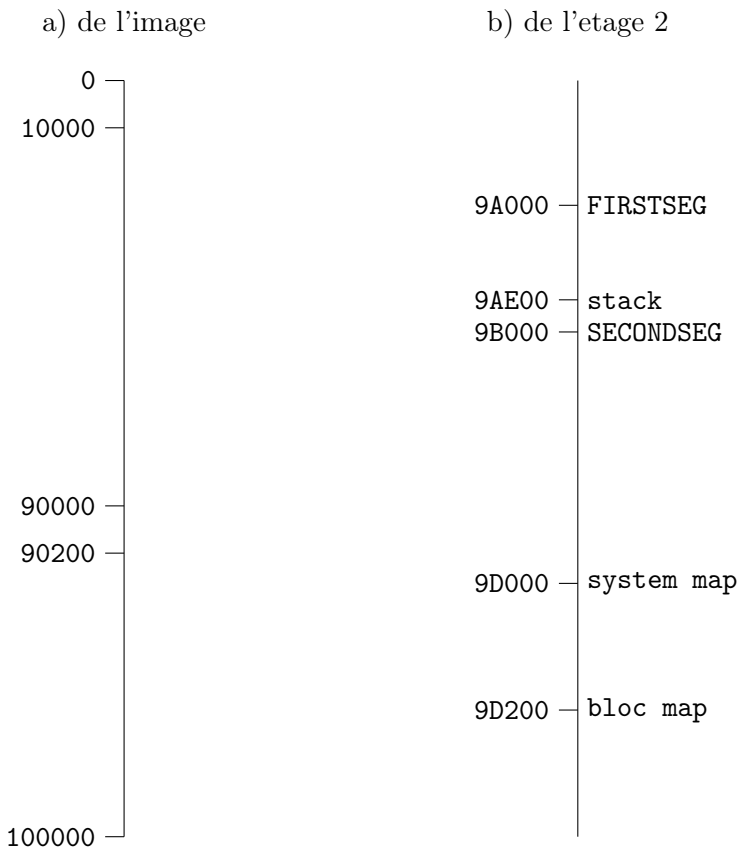


FIGURE 16: Cartographie mémoire.

2.5 Travaux dirigés: Etude du multi-boot LILO

1. Pour lilo une image est un fichier contenant 3 parties appelées boot, setup et system. Indiquez en étudiant le fichier "*lilo.conf*" et les constantes ci-dessous:
   (a) La commande "boot=device" indique le disque sur lequel est installé le boot. Si "/dev/hda2" est choisi comme dans l'exemple que faudra-t-il faire pour le rendre actif? Que se passe-t-il si on choisit "/dev/hda" ou "/dev/fd0"?
   (b) Le rôle des 3 parties.
   (c) En quoi consiste le boot d'une image comme celle appelée "linux" dans le ficher "*lilo.conf*". On complètera la figure 16.a.
   (d) En quoi consiste le boot d'un "other" comme celui appelé "dos" dans le ficher "*lilo.conf*".

```
BOOTSEG   = 0x07C0 ! original address of boot-sector
FIRSTSEG  = 0x9A00
INITSEG   = 0x9000 ! we move boot here - out of the way 0x9000
SETUPSEG  = 0x9020 ! setup starts here
SYSSEG    = 0x1000 | system loaded at 0x10000 (65536)
STACKSEG  = 0x9000
STACK     = 0xB000 ! top of stack
SECOND    = 0x1000 ! second stage loader load address
SECONDSEG = 0x9B00 ! second stage loader segment
DESCR     = 0x2200 ! descriptor load area offset
MAP       = 0x2000 ! map load area offset
CODE_START_1=    100
CODE_START_2=    10
DSC_OFF=         52
ADDR_OFFS=       32
```

2. first.S est installé par LILO sur le premier bloc du périphérique spécifié par "boot=device". "`d_addr`" est un tableau d'éléments dont la structure est donnée sur la table ci-dessous. Sachant que le secteur de boot est chargé en 0x7c00, indiquer l'algorithme général de first.S.

word	byte	byte	byte
secteur	disque	secteur	nombre de
(bits [15:0])	disque	(bits[23:16])	secteur

   – Qui charge first.S ?
   – Pourquoi l'image n'est-elle pas chargée directement?
   – La cartographie de la mémoire à la ligne 130 est donnée sur la figure 16.b. Quelle est la différence entre INITSEG et FIRSTSEG?
   – Détailler le calcul fait aux lignes 168-202.

3. Second étage: La fonction du second étage est de charger en mémoire les 3 parties de l'image (boot,setup,noyau) choisie par l'utilisateur. Pour pouvoir effectuer ceci il a une table dite des descripteurs d'images dont la structure est:

00 →	nom de l'image (FF=fin de table)	16
10 →	password de l'image	16
20 →	nombre d'octets du ramdisk	4
24 →	1^{er} secteur de la table des secteurs du ramdisk	4
28 →	nombre de secteurs de la table précédente (ignoré)	1
29 →	1^{er} secteur de la table des secteurs de l'image	4
2D →	nombre de secteurs de la table précédente (ignoré)	1
31 →	page de chargement (high) de l'image (0 chargé bas)	2
32 →	flag	2

Les numéros de blocs contenant les images sont stockés dans des blocs disque dont la structure est une table dont les éléments sont:

word	byte	byte	byte
secteur	disque	secteur	nombre de
(bits [15:0])	disque	(bits[23:16])	secteurs

La table s'arrête des qu'une entrée est nulle, le dernier numéro de la table correspond à un bloc disque contenant la suite de la table.
Après avoir chargé les composantes de l'image, il termine par:

```
mov ax,#INITSEG ! adjust segment registers
mov ds,ax
mov es,ax
jmpi 0,SETUPSEG ! start the setup
```

   – Où sont stocker ces tables (figure 16.b)? Qui les a chargées et comment? Qui les a créées?
   – Donnez l'algorithme général du second étage.
   – Dans le cas de "linux", que fait le setup?

4. lilo La commande de base est

```
lilo -C config-file -m map-file -i boot-sector
```

Sachant que boot-sector est la concaténation des deux étages et que map-file est un fichier généré par lilo, donnez:

- l'algorithme général de "*lilo*".
- les raisons qui ont poussé les concepteurs à choisir un mécanisme d'addressage par bloc physique, plutôt que de lire le fichier dans le système de fichiers.
- le contenu du boot de l'image.

Fichier: lilo.conf

```
# LILO configuration file

# Start LILO global section
boot = /dev/hda2
delay = 300

# Linux bootable partition
image = /vmlinuz
label = linux
root = /dev/hda2
read-only # linux root is mounted read-only for checking

# DOS bootable partition config begins
other = /dev/hda1
label = dos
table = /dev/hda
```

Fichier: first.S

```
/* first.S  -  LILO first stage boot loader */

/* Copyright 1992-1997 Werner Almesberger. See file COPYING for details. */
... ... ...

  .text
  .globl  _main
  .org    0
_main:    jmp start
  .org    2
! Boot device parameters. They are set by the installer.
  .ascii  "LILO"
... ... ...

d_addr:                   ! second stage sector addresses
  .org    CODE_START_1
ext_si:   .word   0       ! external interface
ext_es:   .word   0
ext_bx:   .word   0
ext_dl:   .byte   0

start:    mov ax,#BOOTSEG ! set DS
  mov ds,ax
  mov ext_es,es   ! copy possible external parameters
  mov ext_si,si
  mov ext_bx,bx
  mov ext_dl,dl
  mov ax,#FIRSTSEG        ! beam us up ...
  mov es,ax
  mov cx,#256
  sub si,si
  sub di,di
  cld
  rep
  movsw
  jmpi    go,FIRSTSEG

go:   cli         ! no interrupts
  mov ds,ax       ! AX is already set
  mov es,ax       ! (ES may be wrong when restarting)
  mov sp,#STACK   ! set the stack
  mov ax,#STACKSEG
  mov ss,ax
  sti             ! now it is safe

  mov  al,#0x0d    ! gimme a CR ...
  call display
  mov  al,#0x0a    ! ... an LF ...
  call display
  mov  al,#0x4c    ! ... an 'L' ...
  call display

lagain:  mov  si,#d_addr  ! ready to load the second stage loader
         mov  bx,#SECOND
         cld
sload:   lodsw           ! get CX
         mov  cx,ax
         lodsw           ! get DX
         mov  dx,ax
         or   ax,cx      ! at EOF ?
         jz   done       ! yes -> start it
         inc  si         ! skip the length byte
         call cread
         jc   error      ! error -> start over again
         add  bx,#512    ! next sector
         jmp  sload
error:
   ... ... ...

done: mov al,#0x49    ! display an 'I'
  call    display
  jmpi    0,SECONDSEG ! start the second stage loader
   ... ... ...

display:xor   bh,bh   ! display on screen
  mov ah,#14
  int 0x10
  ret
   ... ... ...

cread:    test dl,#LINEAR_FLAG      ! linear address ?
          jz   readsect             ! no -> go on
          and  dl,#0xff-LINEAR_FLAG ! remove flag
   ... ... ...
! Translate the linear address into a sector/track/cylinder address
          push  bx      ! BX is used as scratch
```

```
163           push  cx      ! LSW
164           push  dx      ! MSW with drive
165           mov   ah,#8   ! get drive geometry (do not clobber ES:DI)
166           int   0x13
167           jc    linerr  ! error -> quit
    ... ... ...
203 readsect:
204   mov ax,#0x201   ! read one sector
205   int 0x13
206   ret             ! quit, possibly with errors
    ... ... ...
220 /* Here are at least 66 bytes of free space. This is reserved for the
221    partition table and the boot signature. */
223 theend:
```

2.6 Travaux pratiques 1

Attention: Les questions de ce TP sont à faire **séquentiellement**. De manière concrête, la question i ne sera pas notée si les questions 1 à $i-1$ n'ont pas été faites correctement.

Pour ce TP, vous avez à disposition le fichier "*boot.bin*" qui est un boot, le fichier "*setup.bin*" qui est un setup, le fichier "*image.bin*" qui est une image au sens "*`lilo`*", 1 ou 2 PC sans disque dur.

1. A l'aide d'une disquette, faites démarrer votre ordinateur avec le programme "*boot.bin*".
2. Ajoutez une image à "*`lilo`*" qui lance le programme "*setup.bin*".
   Note: *Le message du "`lilo`" indiquant que l'image n'est pas trouvée est aussi généré quand le fichier ne peut pas être une image.*
3. Créez une partition DOS (fdisk, mkfs), montez la (mount), puis ajoutez une image à "*`lilo`*" qui lance le programme "*image.bin*". Le programme chargé devant être physiquement sur le disque DOS.
4. Faites une disquette au format UNIX ext2. Installez sur la disquette un multi-boot qui lance les programmes "*image.bin*" et "*setup.bin*". Testez votre disquette sur le(s) PC sans disque dur.
5. Créez une image de sauvegarde, pour cela copiez l'image contenant le noyau Linux dans "*/root*" et ajoutez une image dans lilo qui la démarre. Vérifiez qu'elle fonctionne.
6. Faites les expérimentations suivantes:
   - Enchainez les commandes "*`rm /boot/vmlinux`*" et "*`reboot`*". Rebootez sur le noyau normal, expliquez pourquoi ca démarre sans problème. Remettez votre système dans un état stable.
   - Ecrivez un programme C (set100ko) qui réécrit des 'A' sur les 100 premiers ko du fichier passé en argument. Enchainez les commandes "*`set100ko /boot/vmlinux`*" et "*`reboot`*". Rebootez sur le noyau normal. Remettez votre système dans un état stable.
7. Ecrivez un programme C qui affiche en clair les partitions actives d'un disque dur, les types reconnus étant EXT2, EXT3, SWAP et FAT.
8. Complètez le programme précédent pour qu'il affiche aussi le numéro du bloc du super-bloc et la taille en mo des partitions actives. Comparez vos résultats avec ceux donnés "*`fdisk`*".

2.7 Travaux pratiques 2

Attention: Les questions de ce TP sont à faire **séquentiellement**. De manière concrête, la question i ne sera pas notée si les questions 1 à $i-1$ n'ont pas été faites correctement.

Pour ce TP, vous avez à disposition le fichier "*cc86*" qui est un script shell, le fichier "*boot.S*" qui est le source du boot de lilo à utiliser comme référenc syntaxique, 1 ou 2 PC sans disque dur.

1. Ouvrez le script cc86 et répondez aux questions suivantes:
   - Que fait ce script?
   - Pourquoi enlève-t-on les 32 premiers octets?
   - Pourquoi n'utilise-t-on pas cc?
   - La taille du fichier généré est un multiple d'une certaine valeur. Quelle est cette valeur et pourquoi cette valeur?
2. Ecrivez un boot "*`essai1`*" qui écrit "1111 bonjour" puis qui boucle indéfiniment. Vérifiez son fonctionnement avec une disquette.
   NB: "bonjour" doit être entré dans le programme par <<`.ascii "bonjour"`>>.
3. Ecrivez un setup "*`essai2`*" qui écrit "2222 au revoir" puis qui boucle indéfiniment. Vérifiez son fonctionnement avec une image dont le boot est "*`essai1`*".
   NB: "au revoir" doit être entré dans le programme par <<`.ascii "au revoir"`>>.

4. Faites une image dont:
   – le boot est le "*essai1*" précédent,
   – le setup écrit "je suis le setup" puis donne la main au noyau, Ce setup aura une taille de 2 ko.
   – le noyau écrit "je suis le systeme" puis boucle indéfiniment.

   NB: les chaînes de caractères doivent être entrées dans le programme avec la primitive <<.ascii>>.
5. Faites une image dont: Faites une image analogue à la précédente mais avec un setup de 512 octets et l'octet 497 du boot qui contient la valeur 1 (boot[497] avec boot un tableau de 512 caractères).

3 Compilation - Lancement - Initialisation du noyau

3.1 Cours

Pré-requis: Lancez la commande "*cc*" pour compiler un programme de votre choix. Relancez la avec l'option -v, recompilez votre programme en enchaînant manuellement les différentes étapes de la compilation (sans utiliser "*cc*".
Générer manuellement un exécutable composé 2 de fichiers sources (.c).

3.1.1 Compilation et édition de liens

Les compilateurs et les éditeurs de liens génèrent des fichiers dits objets (object files). Ces fichiers commencent par une entête. L'entête pour les fichiers de type "a.out" est la suivante:

- `unsigned a_info` Il est composé d'un magic number (2 octets) qui n'a rien de magique mais qui permet d'identifier ce type d'objet, un flag (1 octet) identifiant la machine et un autre flag (1 octet) pour des options. Ainsi "0407 0000 0144 0000", "0314 0000 0144 0000" et "0421 0000 0144 0000" correspondent respectivement à un objet non exécutable, exécutable et un core au format "a.out" sur une machine I386.
- `unsigned a_text, a_data, a_bss` La longueur en bytes des 3 segments.
- `unsigned a_syms` La taille en bytes de la table des symboles.
- `unsigned a_entry` L'adresse de l'instruction qui doit être exécutée lors du lancement de l'objet.
- `unsigned a_trsize,a_drsize` La taille des tables de réallocation des segments text et data.

Derrière l'entête, suivent les différentes sections. 3 sections reflètent directement l'image d'un processus, ce sont le segment TEXT qui correspond aux instructions, le segment DATA qui correspond aux données initialisées, le segment BSS qui correspond aux données non initialisées.

A titre d'exemple la figure 17 présente en A) un programme 'C'. Celui-ci a été compilé en utilisant la commande "*cc -c exemple.c*" qui a créé un fichier "*exemple.o*" dont une partie du contenu est donnée en B). Les trois segments ont été créés.

- Le segment TEXT a sa taille connue ainsi que son contenu qui est donné dans la partie "Disassembly" à l'exception des champs d'adresse qui font référence aux variables globales qui sont mises à 0. Cependant dans la table de réallocation il est inscrit qu'aux adresses 7 et 13 du TEXT il faudra y mettre les adresses des symboles "glv" et "iglv" sur 32 bits. Dans la section "SYMBOL TABLE" on peut voir que le symbol "func" se trouve à l'adresse 0 du segment de TEXT.
- le segment DATA a sa taille connue (4 octets) ainsi que son contenu (voir "Contents of section .data"). C'est le symbole "iglv". Ce symbole est aussi indiqué dans la table des symboles comme étant à l'adresse 0 du segment. Par contre l'adresse du segment est inconnue.
- Le segment BSS est juste défini pour l'instant comme existant avec une taille nulle. Par contre dans la table SYMBOL TABLE on peut voir que "gvl" existe bien.

Cet objet n'est pas exécutable mais il peut être combiné avec d'autres objets de même type pour construire un objet exécutable.

En C) sur la figure 17 est représentée une partie du contenu du fichier exemple qui a été obtenu par la commande:

```
ld  -m elf_i386 -Ttext 0x100000 -e func  -o example example.o
```

Cette commande génère un fichier objet "*example*" au format `elf_i386` à partir du fichier objet "*example.o*" [1] Dans ce fichier le segment TEXT est placé à l'adresse 0x100000, puis suit le segment DATA puis le segment BSS. On peut le voir sur la figure dans "Section". On peut remarquer également que dans la section "Disassembly of section .text", les références aux variables ont été mises à jour. Dans la table des symboles, on peut voir que les symboles "`func`", "`glv`" et "`iglv`" du programmme 'C' "*exemple.c*" ont maintenant une adresse, mais que d'autres symboles ont poussé spontanément, il s'agit de "`_etext`" qui donne l'adresse de fin du segment TEXT, de "`__bss_start`" et de "`_edata`"

1. Ce fichier n'est pas forcément au format `elf_i386`.

A)

```
int  glv;
int  iglv=1;

void func(int x)
{
  glv= x;
  iglv= 100;
}
```

B)

```
Sections:
Idx Name          Size      VMA       LMA       File off
  0 .text         00000017  00000000  00000000  00000034
  1 .data         00000004  00000000  00000000  0000004c
  2 .bss          00000000  00000000  00000000  00000050

SYMBOL TABLE:
00000000 g     O .data  00000004 iglv
00000000 g     F .text  00000017 func
00000004       O *COM*  00000004 glv

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000007 R_386_32          glv
0000000d R_386_32          iglv

Disassembly of section .text:
00000000 <func>:
   0:   55                      pushl  %ebp
   1:   89 e5                   movl   %esp,%ebp
   3:   8b 45 08                movl   0x8(%ebp),%eax
   6:   a3 00 00 00 00          movl   %eax,0x0
   b:   c7 05 00 00 00 00 64    movl   $0x64,0x0
  12:   00 00 00
  15:   c9                      leave
  16:   c3                      ret

Contents of section .data:
0000 01000000
```

C)

```
Sections:
Idx Name          Size      VMA       LMA       File off
  0 .text         00000017  00100000  00100000  00001000
  1 .data         00000004  00101018  00101018  00001018
  2 .bss          00000004  0010101c  0010101c  0000101c

SYMBOL TABLE:
00100017 g     O *ABS*  00000000 _etext
0010101c g     O *ABS*  00000000 __bss_start
0010101c g     O .bss   00000004 glv
00100000 g     F .text  00000017 func
00101018 g     O .data  00000004 iglv
0010101c g     O *ABS*  00000000 _edata
00101020 g     O *ABS*  00000000 _end

Disassembly of section .text:
   0:   55                      pushl  %ebp
   1:   89 e5                   movl   %esp,%ebp
   3:   8b 45 08                movl   0x8(%ebp),%eax
   6:   a3 1C 10 10 00          movl   %eax,0x10101c
   b:   c7 05 18 10 10 00 64    movl   $0x64,0x101018
  12:   00 00 00
  15:   c9                      leave
  16:   c3                      ret

Contents of section .data:
101018 01000000
```

FIGURE 17: Exemple de fichiers objet

```
typedef struct {
    uint   ebx,edx;
    ushort es,ds;
} Treg;
Treg tmp;

Treg a;
getTreg_tmp();
a=tmp;

getTreg_tmp:
  movl %ebx,tmp
  movl %edx,tmp+4
  movw %es, tmp+8
  movw %ds, tmp+10
  ret
```

```
typedef struct {
    uint   ebx,edx;
    ushort es,ds;
} Treg;
Treg tmp;

Treg a;
getTreg(&a);

getTreg:
  movl 4(%esp),%eax
  movl %ebx,(%eax)
  movl %edx,4(%eax)
  movw %es,8(%eax)
  movw %ds,10(%eax)
  ret
```

```
uint ebx,edx;
ushort es,ds;
getregs(&ebx,&edx,
        &es,&ds);

getregs:
  movl 4(%esp),%eax
  movl %ebx,(%eax)
  movl 8(%esp),%eax
  movl %edx,(%eax)
  movl 12(%esp),%eax
  movw %es,(%eax)
  movl 16(%esp),%eax
  movw %ds,(%eax)
  ret
```

FIGURE 18: Différentes façons de récupérer 4 registres

qui donne l'adresse de la fin du segment DATA et de "`_end`" qui donne la fin du segment BSS.

Enfin l'option `-e func` de la commande qui indique le point d'entrée dans l'exécutable (l'adresse de la première instruction à exécuter pour le lancer) n'apparait pas sur la figure mais est stockée dans l'entête du fichier.

3.1.2 Langage 'C' et assembleur

Dans le 'C', en particulier système, pour accéder aux registres ou à certaines instructions ("`cli`") ou bien dans d'autres domaines pour des raisons d'efficacité, on a besoin d'accéder à l'assembleur. La méthode la plus portable est d'écrire une fonction en assembleur et de l'appeler du 'C'. Pour écrire une telle fonction se pose alors le problème du passage des paramètres. La figure 18 donne plusieurs méthodes pour récupérer dans un programme 'C' les valeurs des registres ES, EBX, DS, EDX.

1. une variable globale contient une structure pour ranger les valeurs des registres et une fonction assembleur qui remplit cette structure.
2. on passe cette fois en paramètre la variable où on veut ranger les valeurs des registres.
3. on passe cette fois en paramètre les 4 variables où l'on veut ranger les valeurs des registres.

Toutes ces solutions sont désastreuses au niveau performances car elles doublent ou triplent le nombre d'instructions. Il faut en effet compter aussi l'empilement des paramètres et l'appel de la fonction dans le 'C'. C'est encore pire si par exemple on veut simplement interdire les interruptions sur un I386 où il suffit de faire l'instruction "`cli`".

Pour pallier cette dégradation de performances, la plupart des compilateurs proposent d'introduire directement des instructions assembleurs dans le 'C'. Pour le compilateur "*`gcc`*" ceci est fait par l'instruction "`__asm__( embeded assembly )`". Ainsi pour interdire les interruptions on ferait:

```
#define cli()   __asm__( "cli" )
...
i= j*4+m;
cli();
*p= i;
...
```

L'exemple "`cli`" est loin de la réalité, en effet, en général, il faut passer des valeurs et en récupérer. Là encore "*`gcc`*" fournit une interface, sans doute pas très simple, mais une fois la macro[2] mise au point, elle est indépendante du nombre de variables et de leur ordre de déclaration dans le source 'C'. La syntaxe exacte est:

```
__asm__ (
  "movl %0,    %%ebx\n" /* parametre formel 0 --> ebx  */
  "movl %%ebx, %1\n"    /* ebx --> parametre formel 1 */
  : ...                 /* clause des ecritures */
  : ...                 /* clause des lectures */
  : ...                 /* clause des registres utilises */
);
```

Reprenons l'exemple de lecture des 4 registres EBX, EDX, ES et DS dont le code est présenté ci-dessous. On a utilisé les paramètres formels %0, %1, %2 et %3, les règles `"=m" (expr)` de la clause des écritures les associent dans l'ordre aux expressions qui sur l'exemple sont les variables de la fonction 'C'.

2. Pour la mise au point, la commande `cc -S file.c` génère un fichier "*file.s*" qui permet de vérifier l'assembleur généré.

```
void exemple_2()
{
    uint   EBX,EDX;
    ushort ES, DS;
    ...
    EBX= 101; EDX=102;
    ES=  103; DS= 104;
    __asm__(
      "mov %%ebx,%0\n"
      "mov %%ebx,%1\n"
      "mov %%es, %2\n"
      "mov %%ds, %3\n"
      : "=m" (EBX), "=m" (EDX),
        "=m" (ES),  "=m" (DS)
    );
    ...
}
```

```
exemple_2:
    ...
    movl $101,-4(%ebp)
    movl $102,-8(%ebp)
    movw $103,-10(%ebp)
    movl $104,-8(%ebp)
#APP
    mov %ebx,-4(%ebp)
    mov %ebx,-8(%ebp)
    mov %ES,-10(%ebp)
    mov %ds,-12(%ebp)
#NO_APP
    ...
```

Un exemple complet est présenté sur la figure 19. La clause écriture spécifie que EXB est le paramètre formel %0 et qu'il sera écrit dans `p[3]`, la première règle de la clause lecture associe le paramètre formel à un registre (laissant le choix du registre au compilateur) et l'initialise à `i`, la seconde initialise le paramètre formel %0 (c-a-d: ebx) à `p[2]`, enfin la clause des registres utilisés indiquent les registres modifiés, le compilateur en tient compte dans les instructions "`push`" et "`pop`" de l'entête et du pied de la fonction.

3.1.3 Exercices

Lorsqu'on lance le "*`make zImage`*" pour créer le noyau, après la compilation de tous les fichiers, on arrive à la génération de l'image dont les différentes étapes sont données ci-dessous:

```
 1 ld -m elf_i386 -Ttext 0xC0100000 -e startup_32 \
 2     arch/i386/kernel/head.o \
 3     init/main.o init/version.o arch/i386/kernel/kernel.o ... \
 4     mm/mm.o fs/fs.o ipc/ipc.o net/network.a fs/filesystems.a \
 5     drivers/block/block.a drivers/char/char.a ... \
 6     -o vmlinux
 7 make[2]: Entering directory ‘arch/i386/boot/’
 8 as86 -0 -a -o bootsect.o bootsect.s; ld86 -0 -s -o bootsect bootsect.o
 9 as86 -0 -a -o setup.o setup.s; ld86 -0 -s -o setup setup.o
10 make[2]: Entering directory ‘arch/i386/boot/compressed’
11 tmppiggy=/tmp/$$piggy ; \
12     objcopy -O binary -R .note -R .comment -R .stab -R .stabstr \
13        /usr/src/linux-2.0.30/vmlinux $tmppiggy ; \
14     gzip -f -9 < $tmppiggy > $tmppiggy.gz ; \
15     echo "SECTIONS { .data : { \
16         input_len = .; LONG(input_data_end - input_data) \
17         input_data = .; *(.data) input_data_end = .; }}" > $tmppiggy.lnk; \
18     ld -m elf_i386 -m elf_i386 -r -o piggy.o \
19         -b binary $tmppiggy.gz -b elf32-i386 -T $tmppiggy.lnk;
20 gcc -D__KERNEL__ -I/usr/src/linux-2.0.30/include -traditional -c head.S
21 gcc -D__KERNEL__ -I/usr/src/linux-2.0.30/include -O2 -DSTDC_HEADERS \
22     -c misc.c -o misc.o
23 ld -m elf_i386 -Ttext 0x1000 -e startup_32 -o vmlinux head.o misc.o piggy.o
24 make[2]: Leaving directory ‘arch/i386/boot/compressed’
25 objcopy -O binary -R .note -R .comment -R .stab -R .stabstr \
26     compressed/vmlinux compressed/vmlinux.out;
27 tools/build bootsect setup compressed/vmlinux.out CURRENT > zImage
28 Root device is (8, 20)
29 Boot sector 512 bytes.
30 Setup is 4340 bytes.
31 System is 425 kB
32 sync
33 make[1]: Leaving directory ‘arch/i386/boot’
```

1. Rappeler le rôle d'un linker, ce qu'il prend en entrée et ce qu'il génère en sortie.
2. Indiquer la fonction du programme "*`objcopy`*".
3. Quelle est la différence entre une compilation sous UNIX et la génération de vmlinux?
4. Quelle est la différence entre vmlinux et zImage?
5. Pourquoi le noyau est-il compressé?

```
void exemple_3()
{
int i,*p,*pb;
  ...
  __asm__ (
   "addl %1,%1\n"
   "addl %0,%0\n"
   "addl %1,%0\n"
   :
   /* %0   <-> ebx et
    * p[3] <-- %0 */
    "=b" (p[3])
   :
   /* %1 <-> reg et */
    * i  --> %1 */
    "r" (i),
   /* %0 --> p[2] */
    "0" (p[2])
   : "eax", "ebx",
     "ecx", "edx",
     "esi", "edi"
);
...
}
```

```
exemple_3:
  pushl %ebp
  movl %esp,%ebp
  subl $20,%esp
  ...
  pushl %edi
  pushl %esi
  pushl %ebx
  movl -8(%ebp),%eax
  addl $12,%eax
  movl %eax,-16(%ebp)
  movl -8(%ebp),%ebx
  addl $8,%ebx
  movl (%ebx),%ebx
  movl -4(%ebp),%eax
```

```
#APP
  addl %eax,%eax
  addl %ebx,%ebx
  addl %eax,%ebx
#NO_APP
  movl -16(%ebp),%eax
  movl %ebx,(%eax)
  popl %ebx
  popl %esi
  popl %edi
  ...
  leave
  ret
```

FIGURE 19: Exemple d'assembleur embarqué pour le compilateur 'C' "*`gcc`*".

3.2 Travaux dirigés: Etude de l'initialisation de Linux

3.2.1 Chargement du système

Passage en mode protégé: "boot/compressed/setup.S"

Le rôle de la fonction de "*setup*" est de passer en mode protégé et de lancer le décompresseur. Donnez son algorithme général puis répondez aux questions suivantes:

1. Quel est l'état de la mémoire quand on arrive à "start"? On précisera la différence entre "INITSEG" et "SETUPSEG".
2. "*Setup*" pioche dans le BIOS le maximum d'information. Pourquoi cette fonction lui incombe-t-elle?
3. Etudier le passage en mode protégé 32 bits:
   (a) Que charge t-on en 790-797?
   (b) Décrire la GDT, quelle est la valeur de "__BOOT_CS"?
   (c) Quand passe-t-on en mode protégé? Qu'est ce qui indique le mode 32 bits?

arch/x86/boot/compressed/setup.S (2.6.9)

```
 1 /*
 2  *  setup.S     Copyright (C) 1991, 1992 Linus Torvalds
 3  *
 4  * setup.s is responsible for getting the system data from the BIOS,
 5  * and putting them into the appropriate places in system memory.
 6  * both setup.s and system has been loaded by the bootblock.
 7  *
 8  * This code asks the bios for memory/disk/other parameters, and
 9  * puts them in a "safe" place: 0x90000-0x901FF, ie where the
10  * boot-block used to be. It is then up to the protected mode
11  * system to read them from there before the area is overwritten
12  * for buffer-blocks.
   ... ... ...
47  */
   ... ... ...
61 INITSEG  = DEF_INITSEG   # 0x9000, we move boot here, out of the way
62 SYSSEG   = DEF_SYSSEG    # 0x1000, system loaded at 0x10000 (65536).
63 SETUPSEG = DEF_SETUPSEG  # 0x9020, this is the current segment
64                          # ... and the former contents of CS
   ... ... ...
79 start:
80     jmp trampoline
   ... ... ...
122 code32_start:              # here loaders can put a different
123                            # start address for 32-bit code.
125         .long   0x1000     #   0x1000 = default for zImage
   ... ... ...
166 trampoline: call    start_of_setup
167         .space  1024
   ... ... ...
170 start_of_setup:
   ... ... ...
290 loader_ok:
291 # Get memory size (extended mem, kB)
   ... ... ...
398 # Check for video adapter and its parameters and allow the
399 # user to browse video modes.
   ... ... ...
403 # Get hd0 data...
   ... ... ...
417 # Get hd1 data...
   ... ... ...
494 # Check for PS/2 pointing device
   ... ... ...
579 # Now we want to move to protected mode ...
   ... ... ...
592 # we get the code32 start address and modify the below 'jmpi'
593 # (loader may have changed it)
594     movl    %cs:code32_start, %eax
595     movl    %eax, %cs:code32
   ... ... ...
792 # set up gdt and idt
793     lidt    idt_48          # load idt with 0,0
794     xorl    %eax, %eax      # Compute gdt_base
795     movw    %ds, %ax        # (Convert %ds:gdt to a linear ptr)
796     shll    $4, %eax
797     addl    $gdt, %eax
798     movl    %eax, (gdt_48+2)
799     lgdt    gdt_48          # load gdt with whatever is appropriate
   ... ... ...
810 # well, that went ok, I hope. Now we mask all interrupts - the rest
811 # is done in init_IRQ().
812     movb    $0xFF, %al          # mask all interrupts for now
813     outb    %al, $0xA1
814     call    delay
   ... ... ...
824 # Well, now's the time to actually move into protected mode. To make
825 # things as simple as possible, we do no register set-up or anything,
826 # we let the gnu-compiled 32-bit programs do that. We just jump to
827 # absolute address 0x1000 (or the loader supplied one),
828 # in 32-bit protected mode.
829 #
830 # Note that the short jump isn't strictly needed, although there are
831 # reasons why it might be a good idea. It won't hurt in any case.
```

```
 832     movw    $1, %ax         # protected mode (PE) bit
 833     lmsw    %ax             # This is it!
 834     jmp     flush_instr

 836 flush_instr:
    ... ... ...

 854         .byte 0x66, 0xea  # prefix + jmpi-opcode
 855 code32: .long 0x0000      # will be set to 0x1000 or 0x100000
 857         .word __BOOT_CS
    ... ... ...

 970 # Descriptor tables
    ... ... ...

 982     .align 16
 983 gdt:
 984     .fill GDT_ENTRY_BOOT_CS,8,0

 986     .word   0xFFFF          # 4Gb - (0x100000*0x1000 = 4Gb)
 987     .word   0               # base address = 0
 988     .word   0x9A00          # code read/exec
 989     .word   0x00CF          # granularity = 4096, 386
 990                             #  (+5th nibble of limit)

 992     .word   0xFFFF          # 4Gb - (0x100000*0x1000 = 4Gb)
 993     .word   0               # base address = 0
 994     .word   0x9200          # data read/write
 995     .word   0x00CF          # granularity = 4096, 386
 996                             #  (+5th nibble of limit)
 997 gdt_end:
 998     .align  4

1000     .word   0              # alignment byte
1001 idt_48:
1002     .word   0              # idt limit = 0
1003     .word   0, 0           # idt base = 0L

1005     .word   0              # alignment byte
1006 gdt_48:
1007     .word   gdt_end - gdt - 1 # gdt limit
1008     .word   0, 0           # gdt base (filled in later)
```

Décompression du Noyau: "boot/compress/head.S"

Le rôle de "*head*" est d'appeler la fonction 'C' "*decompress_kernel*" puis de lancer le noyau (enfin!). "*decompress_kernel*" a le noyau compressé dans un gros tableau de données (`input_data` voir la génération page 29) et le décompresse en l'écrivant à partir de l'adresse virtuelle et physique 0x100000 (1 M) .

1. Donner l'état des registres, de l'espace d'adressage et de la pile à la ligne 134.
2. Qu'est-ce le BSS? Pourquoi l'efface-t-on?
3. A quelle adresse physique et virtuelle, va-t-on à la ligne 182, à quoi correspond-elle?

arch/x86/boot/compressed/head_32.S

```
 33 ENTRY(startup_32)
 34     cld
    ... ... ...

 40     cli
 41     movl $(__BOOT_DS),%eax
 42     movl %eax,%ds
 43     movl %eax,%es
 44     movl %eax,%fs
 45     movl %eax,%gs
 46     movl %eax,%ss
    ... ... ...

106     movl    $LOAD_PHYSICAL_ADDR, %ebp
    ... ... ...

121 /* Clear BSS */
122     xorl %eax,%eax
123     leal _edata(%ebx),%edi
124     leal _end(%ebx), %ecx
125     subl %edi,%ecx
126     cld
127     rep
128     stosb


132 /* Setup the stack for the decompressor */
133     leal boot_stack_end(%ebx), %esp

136 /* Do the decompression, and jump to the new kernel. */

    ... ... ...
141     pushl %ebp  # output address
142     movl input_len(%ebx), %eax
143     pushl %eax  # input_len
144     leal input_data(%ebx), %eax
145     pushl %eax  # input_data
    ... ... ...
149     call decompress_kernel
    ... ... ...

181     xorl %ebx,%ebx
182     jmp *%ebp
    ... ... ...

189 boot_stack:
190     .fill BOOT_STACK_SIZE, 1, 0
191 boot_stack_end:
```

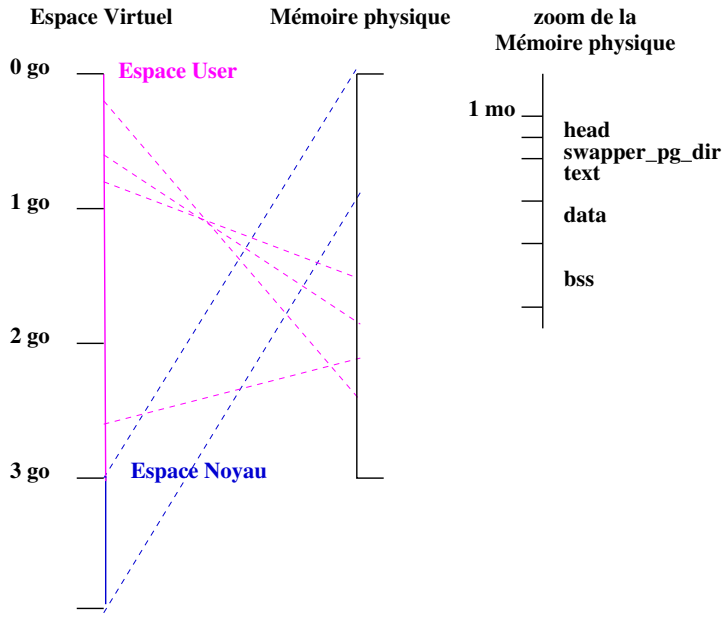


FIGURE 20: Espace virtuel et physique de linux.

3.2.2 Initialisation du noyau

Inititialisation du noyau se fait en 2 étapes. La première, faite par "*head*", s'occupe de tout ce qui est bas niveau, reconnaissance du processeur, récupération des paramètres, de l'initialisation des tables (gdt, ldt, idt, tlb), mise en route de la pagination. La seconde étape est faite par la fonction "*start_kernel*" dans "*init/main.c*" qui finit l'initialisation des tables, crée et/ou initialise toutes les variables et ressources du noyau puis enfin lance le premier[3] processus.

*Préparation de la mémoire: "*kernel/head.S*"*

Donnez les initialisations faites dans "*head*" puis précisez les points suivants:

1. Pourqouoi réinitialise-t-on BSS?
2. Le noyau n'a pas de "main", quels sont ses points d'entrées?
3. Pourquoi les tables système sont-elles déclarées ici et non dans le 'C'?
4. Quand active-t-on la pagination? Quel est le risque de l'instruction qui suit cette activation?
5. La valeur de la macro `pa(x)` est `x-__PAGE_OFFSET` et la définition de `__PAGE_OFFSET` est 0xC0000000 la même valeur que celle utilisée pour générer le noyau (voir figure 3.1.3). Expliquez son utilisation dans le code. Pourquoi les instructions "*jmp*" fonctionnent-elles?
6. La MMU est initialisée pour implanter l'espace virtuel présenté sur la figure 20. Expliquez les avantages de cette solution et comment la protection mémoire sera implantée.

arch/x86/kernel/head_32.S

```
84 ENTRY(startup_32)
   ... ... ...
92 /* Set segments to known values. */
94    movl $(__BOOT_DS),%eax
95    movl %eax,%ds
96    movl %eax,%es
97    movl %eax,%fs
98    movl %eax,%gs
   ... ... ...
103 /* Clear BSS first so that there are no surprises... */
104   cld
```

3. De manière plus précise, on verra plus tard qu'en fait c'est le $2^{ième}$ processus.

```
105    xorl %eax,%eax
106    movl $pa(__bss_start),%edi
107    movl $pa(__bss_stop),%ecx
108    subl %edi,%ecx
109    shrl $2,%ecx
110    rep ; stosl
    ... ... ...
166  * Initialize page tables.  This creates a PDE and a set of page
167  * tables, which are located immediately beyond __brk_base.  The variable
168  * _brk_end is set up to point to the first "safe" location.
169  * Mappings are created both at virtual address 0 (identity mapping)
170  * and PAGE_OFFSET for up to _end.
    ... ... ...
328 /* Enable paging */
329    movl $pa(swapper_pg_dir),%eax
330    movl %eax,%cr3      /* set the page table pointer.. */
331    movl %cr0,%eax
332    orl  $X86_CR0_PG,%eax
333    movl %eax,%cr0      /* ..and set paging (PG) bit */
334    ljmp $__BOOT_CS,$1f /* Clear prefetch and normalize %eip */
335 1:
336    /* Set up the stack pointer */
337    lss stack_start,%esp
    ... ... ...
358    call setup_idt
    ... ... ...
423    lgdt early_gdt_descr
424    lidt idt_descr
425    ljmp $(__KERNEL_CS),$1f
426 1: movl $(__KERNEL_DS),%eax    # reload all the segment registers
427    movl %eax,%ss               # after changing gdt.
429    movl $(__USER_DS),%eax      # DS/ES contains default USER segment
430    movl %eax,%ds
431    movl %eax,%es
433    movl $(__KERNEL_PERCPU), %eax
434    movl %eax,%fs               # set this cpu's percpu
    ... ... ...
468    jmp *(initial_code)
    ... ... ...
489 /*
    ... ... ...
492  *  sets up a idt with 256 entries pointing to
493  *  ignore_int, interrupt gates. It doesn't actually load
494  *  idt - that can be done only after paging has been enabled
495  *  and the kernel moved to PAGE_OFFSET. Interrupts
496  *  are enabled elsewhere, when we can be relatively
497  *  sure everything is ok.
    ... ... ...
500  */
501 setup_idt:
    ... ... ...
```

```
572 /* This is the default interrupt "handler" :-) */
574 ignore_int:
575    cld
577    pushl %eax
578    pushl %ecx
579    pushl %edx
580    pushl %es
581    pushl %ds
582    movl $(__KERNEL_DS),%eax
583    movl %eax,%ds
584    movl %eax,%es
    ... ... ...
588    pushl 16(%esp)
589    pushl 24(%esp)
590    pushl 32(%esp)
591    pushl 40(%esp)
592    pushl $int_msg
593    call printk
    ... ... ...
597    addl $(5*4),%esp
598    popl %ds
599    popl %es
600    popl %edx
601    popl %ecx
602    popl %eax
604    iret
    ... ... ...
608 ENTRY(initial_code)
609    .long i386_start_kernel
    ... ... ...
627 ENTRY(swapper_pg_dir)
628    .fill 1024,4,0
630 swapper_pg_fixmap:
631    .fill 1024,4,0
632 ENTRY(empty_zero_page)
633    .fill 4096,1,0
    ... ... ...
672 int_msg:
673    .asciz "Unknown interrupt or fault at: %p %p %p\n"
    ... ... ...
707 idt_descr:
708    .word IDT_ENTRIES*8-1       # idt contains 256 entries
709    .long idt_table
    ... ... ...
713 ENTRY(early_gdt_descr)
714    .word GDT_ENTRIES*8-1
715    .long per_cpu__gdt_page     /* Overwritten for secondary CPUs */
```

*Démarage du système: "*init/main.c*"*

C'est la fonction "***start_kernel***" qui initialise le noyau et lance le processus "`init`" qui deviendra le père de tous les processus utilisateurs. Parcourez rapidemment son code et donnez les différentes actions exécutées puis precisez

les points suivants:

1. Quel est le contenu de la table "`idt`"? Quel sera le role de l'entrée 0x80?.
2. En regardant la fonction "*`setup_arch`*" précisez les segments d'un processus.
3. Passage de paramètres au noyau:
   – Comment un utilisateur les passe-t-il concrètement?
   – Comment le noyau les récupère-t-il?
4. Quel est le premier processus utilisateur lancé? Comment lancer "*`/bin/bash`*" et que se passe-t-il si on essaye d'exécuter "*`exit`*" sur son prompt?

arch/x86/include/asm/segment.h

```
25 /*
26  * The layout of the per-CPU GDT under Linux:
27  *
28  *   0 - null
29  *   1 - reserved
30  *   2 - reserved
31  *   3 - reserved
32  *
33  *   4 - unused <==== new cacheline
34  *   5 - unused
35  *
36  *  ------- start of TLS (Thread-Local Storage) segments:
37  *
38  *   6 - TLS segment #1 [ glibc's TLS segment ]
39  *   7 - TLS segment #2 [ Wine's %fs Win32 segment ]
40  *   8 - TLS segment #3
41  *   9 - reserved
42  *  10 - reserved
43  *  11 - reserved
44  *
45  *  ------- start of kernel segments:
46  *
47  *  12 - kernel code segment <==== new cacheline
48  *  13 - kernel data segment
49  *  14 - default user CS
50  *  15 - default user DS
51  *  16 - TSS
52  *  17 - LDT
53  *  18 - PNPBIOS support (16->32 gate)
54  *  19 - PNPBIOS support
55  *  20 - PNPBIOS support
56  *  21 - PNPBIOS support
57  *  22 - PNPBIOS support
58  *  23 - APM BIOS support
59  *  24 - APM BIOS support
60  *  25 - APM BIOS support
61  *
62  *  26 - ESPFIX small SS
63  *  27 - per-cpu [ offset to per-cpu data area ]
64  *  28 - stack_canary-20 [ for stack protector ]
65  *  29 - unused
66  *  30 - unused
67  *  31 - TSS for double fault handler
```

init/main.c

```
187 static char * argv_init[MAX_INIT_ARGS+2] = { "init", NULL, };
188 char * envp_init[MAX_INIT_ENVS+2] = { "HOME=/", "TERM=linux", NULL, };
    ... ... ...
318 static int __init init_setup(char *str)
319 {
    ... ... ...
322     execute_command = str;
    ... ... ...
332 }
333 __setup("init=", init_setup);
    ... ... ...
451 static noinline void __init_refok rest_init(void)
453 {
    ... ... ...
456     kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
    ... ... ...
469     schedule();
    ... ... ...
473     cpu_idle();
474 }
    ... ... ...
536 asmlinkage void __init start_kernel(void)
537 {
    ... ... ...
567     boot_cpu_init();
568     page_address_init();
569     printk(KERN_NOTICE "%s", linux_banner);
570     setup_arch(&command_line);
571     mm_init_owner(&init_mm, &init_task);
572     setup_command_line(command_line);
    ... ... ...
582     sched_init();
    ... ... ...
590     printk(KERN_NOTICE "Kernel command line: %s\n", boot_command_line);
591     parse_early_param();
    ... ... ...
601     trap_init();
    ... ... ...
605     init_IRQ();
607     init_timers();
608     hrtimers_init();
```

```
    ... ... ...
611     time_init();
    ... ... ...
699     rest_init();
700 }
    ... ... ...
793 static void run_init_process(char *init_filename)
794 {
795     argv_init[0] = init_filename;
796     kernel_execve(init_filename, argv_init, envp_init);
797 }
    ... ... ...
802 static noinline int init_post(void)
804 {
    ... ... ...
810     system_state = SYSTEM_RUNNING;
    ... ... ...
813     if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
814         printk(KERN_WARNING "Warning: unable to open an initial console.\n");
816     (void) sys_dup(0);
817     (void) sys_dup(0);
819     current->signal->flags |= SIGNAL_UNKILLABLE;
    ... ... ...
833     if (execute_command) {
834         run_init_process(execute_command);
835         printk(KERN_WARNING "Failed to execute %s.  Attempting "
836                     "defaults...\n", execute_command);
837     }
838     run_init_process("/sbin/init");
839     run_init_process("/etc/init");
840     run_init_process("/bin/init");
841     run_init_process("/bin/sh");
843     panic("No init found.  Try passing init= option to kernel.");
844 }
846 static int __init kernel_init(void * unused)
847 {
    ... ... ...
894     init_post();
895     return 0;
896 }
```

arch/x86/kernel/setup.c

```
679 void __init setup_arch(char **cmdline_p)
680 {
    ... ... ...
694     ROOT_DEV = old_decode_dev(boot_params.hdr.root_dev);
695     screen_info = boot_params.screen_info;
    ... ... ...
739     init_mm.start_code = (unsigned long) _text;
740     init_mm.end_code = (unsigned long) _etext;
741     init_mm.end_data = (unsigned long) _edata;
742     init_mm.brk = _brk_end;
    ... ... ...
995 }
```

arch/x86/kernel/traps.c

```
907 void __init trap_init(void)
908 {
    ... ... ...
919     set_intr_gate(0, &divide_error);
920     set_intr_gate_ist(1, &debug, DEBUG_STACK);
921     set_intr_gate_ist(2, &nmi, NMI_STACK);
923     set_system_intr_gate_ist(3, &int3, DEBUG_STACK);
925     set_system_intr_gate(4, &overflow);
926     set_intr_gate(5, &bounds);
927     set_intr_gate(6, &invalid_op);
928     set_intr_gate(7, &device_not_available);
930     set_task_gate(8, GDT_ENTRY_DOUBLEFAULT_TSS);
    ... ... ...
934     set_intr_gate(9, &coprocessor_segment_overrun);
935     set_intr_gate(10, &invalid_TSS);
936     set_intr_gate(11, &segment_not_present);
937     set_intr_gate_ist(12, &stack_segment, STACKFAULT_STACK);
938     set_intr_gate(13, &general_protection);
939     set_intr_gate(14, &page_fault);
940     set_intr_gate(15, &spurious_interrupt_bug);
941     set_intr_gate(16, &coprocessor_error);
942     set_intr_gate(17, &alignment_check);
    ... ... ...
965     set_system_trap_gate(SYSCALL_VECTOR, &system_call);
    ... ... ...
985 }
```

3.3 Travaux pratiques

Attention: Les questions de ce TP sont à faire **séquentiellement**. De manière concrête, la question i ne sera pas notée si les questions 1 à $i-1$ n'ont pas été faites correctement.

Pour certains de ces exercices la commande "*dmesg*" qui affiche les 4 derniers kilo-octets des messages du noyau peut vous être utile. Lorsque vous testez un nouveau noyau, il est conseillé de mettre aussi à jour le fichier "*/boot/System.map*",

1. Générer le noyau et le lancer.
2. Générer un noyau qui affiche une bannière personnelle bien visible du

genre:

```
################################################
########      MA BANIERE A MOI          ########
################################################
```

3. Modifiez le noyau pour qu'il génère 2 interruptions (instruction assembleur `int`). La première devant appeler le traitement par défaut initialisé dans le fichier "*head.S*". La seconde devant appeler l'interruption non reconnue ("`Spurious Interrupt`"). Attention trouvez la routine d'interruption de cette dernière et modifiez la pour qu'elle. affiche quelque chose.
4. Rebootez le system sans le modifier avec une ligne de commande qui lui fera lancer un "*bash*" au lieu de "*init*". Que se passe-t-il si on tape "*CTL-d*". Indiquez où on est passé dans le fichier "*main.c*".
5. Ouvrez le fichier "*kernel/printk.c*" et regardez comment on peut choisir le niveau de "log" avec printk. Modifiez votre noyau pour qu'il affiche au démarrage un message pour chacun des niveaux. Indiquez où est écrit chacun de ces messages (cherchez les dans "*/var/log*" ou "*/var/adm*" ou ...).
   Note: ca se passe dans la fonction "*vprintk*" aux lignes 697-698.
6. Faites un programme utilisateur qui affiche en (hexadecimal) les registres segment "`CS`", "`DS`", "`ES`", "`SS`". Générez un noyau qui affiche les mêmes registre. Donnez la signification de ces valeurs. Indiquez où ces segments sont définis dans le noyau.
7. Générer un noyau qui prouve que "`init`" de "*init/main.c*" est un processus différent du processus initial (celui qui fait idle). Pour cela on utilisera la pile.
8. Modifiez le noyau pour qu'il accepte un paramètre `tutu` qui sera un entier et qui affichera son carré au démarrage.

4 Appels système - Commutation de processus

Pré-requis: Pour la partie en mode utilisateur, étudiez les fonctions "*fopen*", "*fread*", ...et les fonctions "*open*", "*read*", ... On pourra faire les petits programmes suivant:

- Ouvrir un fichier et y écrire 1.000.000 de 'a'.
- Ouvir un fichier et remplacer le 500.000 caractère par un 'b'.

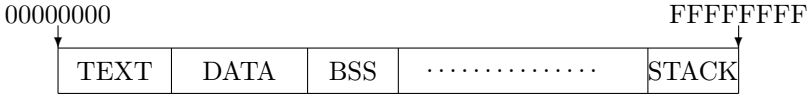


FIGURE 21: Entité processus en mode User.

Pour la seconde partie étudiez les appels système "*fork*", "*signal*", "*wait*", "*kill*".

4.1 Organisation d'un système

4.1.1 Processus

Dans un système d'exploitation multi-tâches, une tâche est représentée par une entité que l'on appelle un processus. Un processus est une zone de mémoire virtuelle (figure 21) comportant plusieurs segments:

- le segment TEXT contient les instructions du processus, il est généralement placé en début de mémoire et il est accessible en lecture uniquement (Read-Only).
- le segment DATA contient les données initialisées du programme, il est généralement placé après le segment TEXT et il est accessible en lecture et écriture. Parfois ce segment est coupé en 2 parties, une n'étant pas accessible en écriture.
- le segment BSS contient les variables non initialisées, il est placé généralement après le segment DATA et il est aussi accessible en lecture et écriture.
- le segment STACK contient la pile utilisateur du processus, il est placé en fin de mémoire quand on empile vers les adresses décroissantes, il est bien sûr accessible en lecture et écriture.

On a vu dans le chapitre 2.1 que la fonction principale d'un système d'exploitation multi-tâches est d'exécuter des tâches de manière indépendante et en toute sécurité:

- Une tâche ne peut pas influencer le déroulement d'une autre sauf temporellement).
- Une tâche ne peut pas faire tomber le noyau sauf pour l'arrêter proprement.

```
int sys_call_i(
  TypeP0 p0,
  TypeP1 p1, ...)
{
  TypeR0 r0;
  TypeR1 r1;
  ...
  si p0 incorect alors
    erreur= BadParam0
    goto err_end
  finsi
  si p1 incorect alors
    erreur= BadParam0
    goto err_end
  finsi
  si p2 incorect alors
    erreur= BadParam0
    goto err_end
  finsi
  ...
  erreur=alloc_R0(&r0,...)
  si erreur alors
    goto err_r0;
  fsi
  erreur=alloc_R1(&r1,...)
  si erreur alors
    goto err_r1;
  fsi
  erreur=alloc_R2(&r2,...)
  si erreur alors
    goto err_r2;
  fsi
  ...
  erreur=sys_call_ido(...)
  ...
err_r2:
  free_R2(r2);
err_r1:
  free_R1(r1);
err_r0:
  free_R0(r0);
err_end
  return erreur;
}
```

FIGURE 22: Algorithme général d'un appel système.

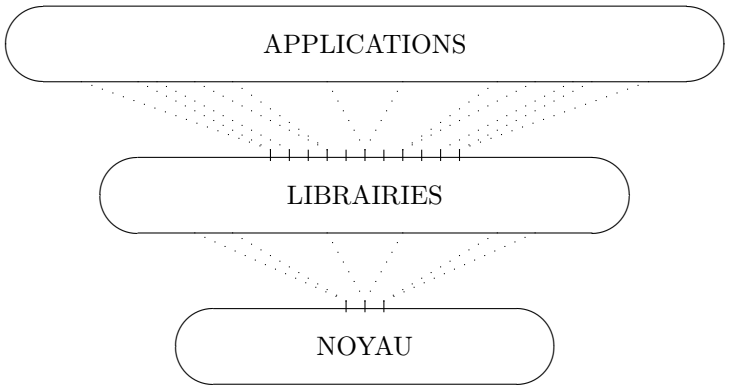


FIGURE 23: Organisation logicielle des systèmes.

Pour qu'une tâche ne puisse pas influencer une autre tâche, le système d'exploitation restreint l'accès à la mémoire pour le processus associé à la tâche à ses segments en n'allouant pas les autres pages dans la MMU.
Pour qu'une tâche ne puisse pas faire tomber le noyau, le processus associé à la tâche ne peut pas accéder à la mémoire de ce dernier, il peut cependant utiliser les services qu'il fournit en passant par des points d'accès que l'on appelle des "**appels sytème**". Les appels système sont les points d'entrée du noyau, ils sont écrits de manière sécurisée, leur algorithme général est présenté sur la figure 22. Avant de faire quoi que ce soit, la validité des paramètres p_i est vérifiée, puis l'appel systèmes alloue les ressources internes r_i dont il a besoin, puis il effectue le service demandé, puis il désalloue les ressources. Notons enfin que si quelque chose se passe mal, un code d'erreur est renvoyé et toutes les allocations déjà effectuées sont libérées.

4.1.2 Organisation

Le nombre d'appels système est relativement réduit et de bas niveau, les principales raisons sont essentiellement la difficulté d'écrire du code sécurisé, le besoin d'efficacité et le besoin de rester général (non dédié à une application). Pour les premiers points, plus la fonction de l'appel système est complexe plus les nombres de paramètres et de ressources nécessaires à sa réalisation sont élevés, ce qui alourdit d'autant son code, et donc son efficacité.
Pour le dernier point, un noyau comme Unix ou Vms se veut "general purpose". Cela signifie que leurs appels système ne doivent pas être dédiés pour telle ou telle application, mais d'un niveau suffisamment bas pour que l'on puisse construire dessus n'importe quelle application. Par exemple le noyau Unix n'a que 2 appels système pour lire l'heure et la date, un retourne le nombre [4] de secondes écoulées depuis le 1 janvier 1970 à 0 heure, 0 minute, 0 seconde, l'autre retourne le nombre de minute à l'est du méridien de Greenwich. Avec une telle interface, écrire l'heure actuelle ou la date du jour, n'est pas spécialement simple. Mais existe il quelque chose de meilleur? La solution alternative: un appel système qui donne la date sous forme d'un triplet d'entier (j,m,a) et un autre qui donne l'heure sous la forme d'un triplet (h,m,s) n'est pas tellement viable pour diverses raisons:

- Il en faudrait 2 autres pour donner la date et l'heure en TU.
- Si l'horloge de l'ordinateur est en TU, les 2 appels système ne seraient pas efficaces.
- Les Chinois avec leur calendrier non Grégorien seraient ravis.
- Les manipulations (comparaison, addition, soustraction, ...) ne seraientt pas très facile.
- D'autres informations relatives à une date manqueraient: jour de la semaine, jour ouvré, le saint, la lune, ...
- Pour un autre système, il n'y a aucune raison que les appels système soient exactement les mêmes: même fonction mêmes paramètres, même ordre des

4. En première approximation pour la signification exacte voir POSIX.1 Annex B 2.2.2.

paramètres, même gestion des erreurs.

Les systèmes sont donc organisés comme le montre la figure 23, un petit nombre d'appels système sécurisés de bas niveau. Pour qu'ils soient tout de même facilement utilisables d'une part, et utilisés avec efficacité d'autre part, une bibliothèque de fonctions sert d'interface. Le code et les variables de ces fonctions feront partie du processus. Par exemple la bibliothèque standard UNIX contient une dizaine de fonctions pour manipuler les dates et les heures dans tous les sens.

4.1.3 Exercices

1. Quand aura lieu le bug de l'an 2000 pour Linux?
2. Pourquoi les segments sont-ils placés dans cet ordre?
3. Qui fait les appels système?
4. Qui traite les appels système?
5. Comparez la famille "*fopen*", "*fread*", ... à la famille "*open*", "*read*", ... Ci-dessous vous avez une implantation simplifiée de la première famille. Complètez l'algorithme de "*fread*".

```
1 struct _MYFILE {
2   char * buffer;   /* buffer de lecture */
3   int    size;     /* taille en octet de buffer */
4   int    position; /* ptr de lecture dnas buffer */
5   int    fd;       /* fichier */
6   int    fin;      /* ptr de remplissage du buffer */
7 };
8 MYFILE * fopen (char * name, int size) {
9   struct _MYFILE * myfile;
10  ...
11  ...
12  ...
13  ...
14  ...
15  return myfile;
16 }
17 int fread (MYFILE  * fichier, char * dest, int nboctets){
18  int i;
19  for ( i=0; i < nboctets; i++ ) {
20    ...
21      ...
22      ...
23      ...
24      ...
25    ...
26    ...
27  }
28  return i;
29 }
```

6. Qu'est-ce "*libc.a*"?
7. Pourquoi n'a-t-on pas l'interface ci-dessous pour la lecture d'un fichier?
   `int read(char* filename, int pos, int nb, void* buffer)`
8. Pourquoi un appel système coûte cher (revenir à cette question après avoir vu les appels système de l'intérieur)?

4.2 Travaux dirigés: Déclenchement de l'appel système

Rappelez la fonction de l'appel système "*open(char*name,int, mode_t)*". Cette fonction est une fonction assembleur de la bibliothèque "*libc.a*".

1. Dans le fichier "*syscall.h*", on trouve les définitions suivantes:
   #define SYS_setup 0 /* Used only by init, to get system going. */
   #define SYS_exit 1
   #define SYS_fork 2
   #define SYS_read 3
   #define SYS_write 4
   #define SYS_open 5

   Dans le fichier "*__open.S*", on trouve le code:
   `PSEUDO(__libc_open,open,3) ret`
   Donnez l'assembleur complet de la fonction "*open*" en utilisant les définitions de macros de la figure 24.
2. Que fait-on au début de l'appel système?
3. Comment sont passés les paramètres de l'appel système?
4. Où branche l'instruction "`int 0x80`" (voir trap.c)? Quelle instruction devra être exécutée pour reprendre juste derrière?
5. Comment détecte-t-on une erreur? Quel est le rôle de la variable "`errno`"?
6. Ecrire la fonction "`errno_location`". Pourquoi est-ce une fonction?
7. Donnez les lignes C standards qui lancent un appel système et qui s'il s'est mal passé, affichent l'erreur en clair et termine le processus.

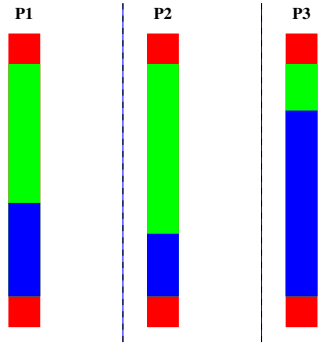
4.3 Travaux dirigés: Processus et scheduling

4.3.1 Processus utilisateur et système

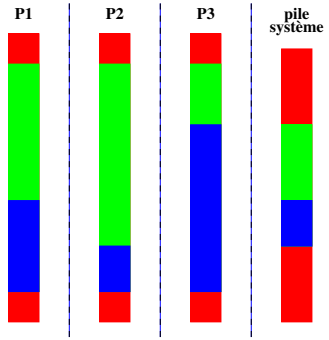
1. Quel est l'ensemble minimal d'entités qui définit un processus? Une entité appartient à cet ensemble si deux processus ne peuvent pas la partagée.

```
 1 #define SYMBOL_NAME_LABEL(X) X##:
 2 #define L(X) .L##X
 3 #define LL(X) .L##X##:

 4 #define    PSEUDO(name, syscall_name, args)     \
 5   .text;                                        \
 6   .globl name;                                  \
 7   .align 4;                                     \
 8   SYMBOL_NAME_LABEL(name)                       \
 9   ENTRY (name)                                  \
10     pushl %ebp;                                 \
11     movl  %esp,%ebp;                            \
12     PUSH_##args                                 \
13     movl  $(SYS_##syscall_name), %eax;          \
14     MOVE_##args                                 \
15     int   $0x80;                                \
16     movl  %eax,%edx;                            \
17     test  %edx,%edx;                            \
18     jge   L(Lexit);                             \
19     negl  %edx;                                 \
20     pushl %edx;                                 \
21     call __errno_location;                      \
22     popl  %edx;                                 \
23     movl  %edx,(%eax);                          \
24     movl  $-1,%eax;                             \
25   LL(Lexit)                                     \
26     POP_##args                                  \
27     movl %ebp,%esp;                             \
28     popl %ebp;

29 #define PUSH_0    /* No arguments to push.  */
30 #define PUSH_1    pushl %ebx;
31 #define PUSH_2    PUSH_1
32 #define PUSH_3    PUSH_1
33 #define PUSH_4    pushl %esi; PUSH_3
34 #define PUSH_5    pushl %edi; PUSH_4

35 #define    MOVE_0    /* No arguments to move.  */
36 #define    MOVE_1    movl 8(%ebp),%ebx;
37 #define    MOVE_2    MOVE_1 movl 12(%ebp),%ecx;
38 #define    MOVE_3    MOVE_2 movl 16(%ebp),%edx;
39 #define    MOVE_4    MOVE_3 movl 20(%ebp),%esi;
40 #define    MOVE_5    MOVE_4 movl 24(%ebp),%edi;

41 #define POP_0
42 #define POP_1    popl %ebx;
43 #define POP_2    POP_1
44 #define POP_3    POP_1
45 #define POP_4    POP_3 popl %esi;
46 #define POP_5    POP_4 popl %edi;
```

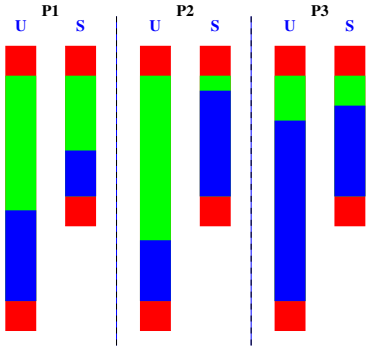
FIGURE 24: Les macros définissant les appels système.



a) Une pile utilisateur/système par processus



b) Une pile utilisateur par processus et une pile système partagée



c) Deux piles par processus une utilisateur et une système

FIGURE 25: Différentes organisation de piles

2. Etudiez les organisations présentées sur la figure 25 au niveau fonctionnel et robustesse et remplissez la table ci-dessous (HW=nécessite du matériel spécifique, C=complexité de mise en œuvre, R=Possibilité de robustesse):

type	R	C	HW	domaine d'application
N.U				
N.U + 1.S				
N.U + N.S				

3. Pour le modèle à 2 piles par processus indiquez si les propositions suivantes sont vraies ou fausses.

   Proposition 1 Le processus système traite les appels système.

   Proposition 2 Les commutations entre processus n'ont lieu qu'en mode système.

   Proposition 3 Lorsqu'un processus a le CPU, c'est lui qui décide quand donner la main.

   Proposition 4 Les interruptions sont traités par le processus système.

   Proposition 5 Lorsqu'un processus segfaulte c'est un autre processus qui traite sa fin.

   Proposition 6 La dernière action d'un processus est de commuter.

4.3.2 Processus Linux

Pour le noyau un processus est représenté par le type "`struct task_struct`". A chaque processus est associé une variable de ce type, le noyau y trouve soit directement, soit en suivant des pointeurs toutes les informations sur le processus: où il se trouve en mémoire ou sur le disque, les ressources utilisées par le processus (fichiers ouverts, sémaphores attachés, ...).
Le type "`struct task_struct`" est défini dans le fichier "*sched.h*", étudiez ce type puis répondez aux questions suivantes:

1. Quels sont les 3 identifiants d'utilisateur et de groupe?
2. L'état d'un processus est défini par le champ "`state`". Indiquez la signification des états suivants:

   TASK_RUNNING:

   TASK_INTERRUPTIBLE:

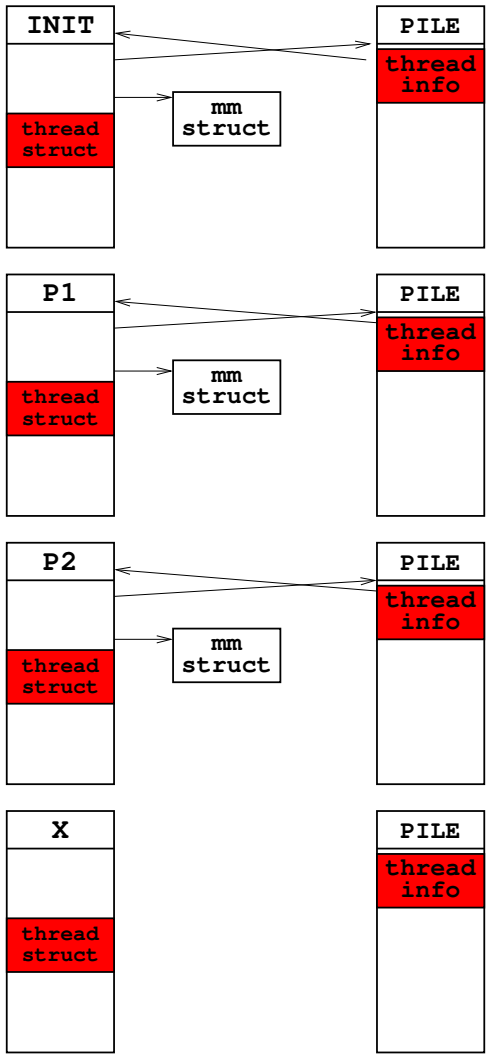


FIGURE 26: Structures de données

```
#define PAGE_SIZE    4096
#define THREAD_ORDER 0  // ou 1
#define THREAD_SIZE (PAGE_SIZE << THREAD_ORDER)
#define current     get_current()
#define alloc_thread_info(tsk) kmalloc(THREAD_SIZE, GFP_KERNEL)

struct thread_info {
   struct task_struct *task;   /* main task structure */
   ...                         /* + quelques champs */
   __u32               flags;  /* low level flags */
   __u32               cpu;    /* current CPU */
   ...                         /* + quelques champs */
};

static inline struct thread_info *current_thread_info(void)
{
   struct thread_info *ti;
   __asm__("andl %%esp,%0; ":"=r" (ti) : "0" (~(THREAD_SIZE - 1)));
   return ti;
}

// idem a utilier en assembleur
#define GET_THREAD_INFO(reg) movl $-THREAD_SIZE, reg; andl %esp, reg

// Linux 2.6.9
static inline struct task_struct * get_current(void)
  { return current_thread_info()->task; }

// Linux 2.6.30 NO-SMP
extern struct task* per_cpu_current;
#define get_current() per_cpu_current

// Linux 2.6.30 SMP
extern <<attribut pour placer dans un segment special>>
  struct task* per_cpu_current;
static __always_inline struct task_struct *get_current(void)
{
  struct task_struct ret__;
  asm( "movl %%fs:per_cpu_current, %0\n" : "=r" (ret__);)
  return ret__
}
```

FIGURE 27: Récupération du pointeur sur la structure "`task_struct`" du processus courant.

TASK_DEAD/ZOMBIE:

TASK_UNINTERRUPT:

3. Le schéma sur la figure 26 représente une vue simplifiée de la structure de données liant la pile système et les types "`task_struct`", "`thread_info`" "`threaf_struct`". Complètez cette figure dans les cas suivants:
   (a) le processus P2 "fork" et son fils est le processus X.
   (b) le processus P2 "thread" et la thread est le processus X.
4. On a N processus utilisateur qui n'ont pas créé de thread et M processus utilisateurs qui ont créé 2 threads chacun. Complètez la table suivante:

task_struct	mm_struct	pile U	seg. pile U	pile S
N+3*M+1				

5. Les variables "`init_task`" et "`init_mm`" décrivent le premier processus (voir chapitre 3.2.2).
   – En quoi ce processus se distingue-t-il des autres? Comment sa structure "`task_struct`" est t-elle allouée et initialisée.
   – A-t-on un processus kernel qui traite les appels système?
   – Quelle est l'utilité de ce processus?
6. Dans le noyau, on utilise la <<variable>> "`current`" pour référencer la tâche courante. En utilisant la figure 27, indiquez comment ce pointeur est récupéré. Quel est l'intérêt de cette méthode?

4.3.3 scheduling

La fonction "*`schedule`*" présentée ci-dessous est le centre de l'allocation de la ressource "processeur". Elle implante un algorithme dit de "round robin", elle est appelée lorsqu'une commutation est pressentie. Elle cherche alors le processus le plus prioritaire et lui donne le processeur.

```
void schedule()
{
  struct task_struct* prev;
  struct task_struct* next;
  prev= current;
  next= processus le plus prioritaire parmi les processus running.
  si next->quantum==0 alors
    recalculer les priorités des processus
  fsi
  si next==prev alors
    return;
  sinon
```

```
    switch_mm(prev->mm,next->mm)
    switch_to(prev,next,prev);
  fsi
}
```

Les règles de base de l'algorithme d'ordonnancement "round robin" sont:

- Sauf événement exceptionnel un processus qui prend le CPU le conserve pendant un quantum de temps.
- Sauf événement exceptionnel un processus doit rendre le CPU quand son quantum de temps est épuisée.
- Sauf événement exceptionnel quand tous les processus actifs ont épuisé leur quantum de temps, ils sont réinitialisés.

Pour implanter cet algorithme, il faut une variable "`need_resched`" qui indique si le processus doit commuter.

1. Indiquez comment mettre à jour la variable "`need_resched`"?
2. Donner l'instruction qu'exécute régulièrement un processus poli (soucieux de laisser travailler les autres processus).

4.3.4 Commutation de processus

La commutation de processus présente 2 aspects, le premier est purement technique: Comment commuter?, le second moins terre-à-terre est: Quand commuter?

Comment commuter?

La macro "`switch_to`" et la fonction "*`__switch_to`*" (figure 28) implantent la commutation de processus. Sachant que `tss->esp0=n->thread->esp0` est l'extension de la macro "`load_esp0(tss,n)`", Etudiez ces fonctions puis répondez aux questions suivantes.

1. Où reprend un processus?
2. Quels sont les registres sauvegardés/restaurés par ces fonctions?
3. Les registres EAX, ECX, EDX ne sont pas sauvegardés/restaurés par ces fonctions. Pourquoi ceci n'est-il souvent pas nécessaire?
4. Quand sera utilisée l'assignation `load_esp0(tss,next)`? Indiquez ce qui ce passe si une interruption arrive juste avant et après cette assignation.
5. Pourquoi la commutation de tâche du processeur Ix86 n'est pas utilisée.

```
#define switch_to(prev, next, last) do { \
    asm volatile(                                          \
       "pushfl\n\t"                 /* save flags */   \
       "pushl %%ebp\n\t"            /* save EBP   */   \
       "movl  %%esp,%[prev_sp]\n\t" /* save ESP   */   \
       "movl  %[next_sp],%%esp\n\t" /* rest.ESP   */   \
       "movl  $1f,%[prev_ip]\n\t"   /* save EIP   */   \
       "pushl %[next_ip]\n\t"       /* rest.EIP   */   \
       "jmp __switch_to\n"                               \
       "1:\t"                                            \
       "popl %%ebp\n\t"            /* rest. EBP   */   \
       "popfl\n"                   /* rest. flags */   \
       : /* output parameters */                         \
         [prev_sp] "=m" (prev->thread.sp),               \
         [prev_ip] "=m" (prev->thread.ip),               \
         "=a" (last),                                    \
       : /* input parameters: */                         \
         [next_sp]  "m" (next->thread.sp),               \
         [next_ip]  "m" (next->thread.ip),               \
       : "ebx", "esi", "edi")                            \
   } while (0)
```

a) commutation au niveau registre

```
/* switch_to(x,yn) should switch tasks from x to y.  */
struct task_struct* fastcall __switch_to(
    struct task_struct *prev_p, struct task_struct *next_p) {
    struct thread_struct *prev = &prev_p->thread,
    struct thread_struct *next = &next_p->thread;
    int                   cpu = smp_processor_id();
    struct tss_struct    *tss = &per_cpu(init_tss, cpu);

    __unlazy_fpu(prev_p);
    ...
    load_sp0(tss, next);
    ...
    lazy_save_gs(prev->gs);
    ...
    lazy_load_gs(next->gs);

    return prev_p;
}
```

b) commutation au niveau tâche

FIGURE 28: Commutation de processus

D'où vins-je? Qui suis-je? Où vais-je?

Qui suis-je? Si je me pose cette question, c'est que je vis, je suis donc le processus "`*current`".

D'où vins-je? A l'origine je suis né d'un "*fork*" de mon père, qui lui même était né d'un "*fork*" de mon grand-père et ainsi de suite jusqu'à notre père universel. La création de notre père universel est assez floue, certains parlent de dieu d'autres de la grande illumination.
Ma vie a été une succession de périodes d'éveil et d'hibernation. J'ai quitté ma dernière hibernation grâce à un parent (on est tous des frères) qui est entré en hibernation pour me réveiller.

1. Qui est le père universel?
2. Combien ai-je de parents éveillés en période d'éveil?.
3. Mon père, mon grand-père sont-ils encore en vie?
4. Comment mon parent m'a-t-il réveillé?

Où vais-je? Mes possibilités sont: rester éveillé, entrer en hibernation et réveiller un parent, me suicider et réveiller un parent.

1. Quand opte-je pour le suicide?
2. Quand suis-je obligé d'opter pour une hibernation? Quelle instruction dois-je exécuter? Quand et à quelle instruction me réveillerai-je?
3. J'opte socialement pour une hibernation par
   `if (need_schedule) schedule();`
   Choisissez parmi les places suivantes, où il faut la placer:
   - Régulièrement, mise à la main ou ajoutée par le compilateur.
   - Au passage du mode système au mode utilisateur.
   - Au passage du mode utilisateur au mode système.
   - Dans les 2 derniers cas que se passe-t-il un processus fait "while (1);" en mode utisateur, en mode system.

4.4 Travaux dirigés: Traitement d'un appel système et des signaux

4.4.1 Changement de mode

Rappelez les circonstances qui amènent à la fonction "*system_call*" qui se trouve dans le fichier "*entry_32.S*". Donnez son algorithme général puis répondez aux questions suivantes:

1. Faites le schéma de la pile et donnez la valeurs des registres EAX, EBX, EDX à l'instruction "int 0x80" pour l'appel système "*open*". Faites le schéma de la pile à la ligne 531 du fichier "*entry_32.S*", en utilisant les macros "`SAVE_ALL`" et "`RESTORE_ALL`" donées ci-dessous. Quel est le prototype de la fonction "*sys_open*".

```
 1 .macro SAVE_ALL
 2     cld
 3     pushl %gs;  pushl %fs
 4     pushl %es;  pushl %ds
 5     pushl %eax; push %ebp
 6     push  %edi; push %esi
 7     pushl %edx; pushl %ecx
 8     pushl %ebx
 9     movl $$(__USER_DS), %edx
10     movl %edx, %ds
11     movl %edx, %es
12     movl $$(__KER_PERCPU), %edx
13     movl %edx, %fs
14 .endm
```

```
 1 .macro RESTORE_REGS
 2     popl %ebx
 3     popl %ecx
 4     popl %edx
 5     popl %esi
 6     popl %edi
 7     popl %ebp
 8     popl %eax
 9     popl %ds
10     popl %es
11     popl %fs
12     popl %gs
13 .endm
```

2. Expliquez le cheminement du code d'erreur des appels système.

```
519 ENTRY(system_call)
522     pushl %eax              # save orig_eax
524     SAVE_ALL
525     GET_THREAD_INFO(%ebp)
529     cmpl $(nr_syscalls), %eax
530     jae syscall_badsys
532     call *sys_call_table(,%eax,4)
533     movl %eax,PT_EAX(%esp)        # store the return value
534 syscall_exit:
540     movl TI_flags(%ebp), %ecx
541     testl $_TIF_ALLWORK_MASK, %ecx    # current->work
542     jne resume_userspace

545 restore_all:
546     RESTORE_ALL
547     iret

1189 #include "syscall_table_32.S"
1191 syscall_table_size=(.-sys_call_table)
```

3. Indiquez ce que fait la fonction "*do_notify_resume*" dans les cas simples suivants:
   - Le signal SIGKILL est positionné.
   - Le signal SIGQUIT est positionné et ignoré.
4. Le processus $P1$ émet un signal à un processus $P2$. Quand $P2$ le prendra-t-il en compte dans les cas suivants:
   - $P2$ est en mode user.
   - $P2$ est en mode système.
   - $P2$ est bloqué sur une IO.

5. Traitement des callback des signaux. Indiquez ce qui ce passe dans les étapes suivantes.
   - le processus $P1$ fait "`signal(SIGUSR1,(int(*)(int))handler)`"
   - un processus $P2$ lance l'appel système "`kill(pid_P1,SIGUSR1)`".
   - $P1$ a été interrompu entre ces deux instructions:
     `move $addr_var,%eax; move $5,(%eax);`
     Quand $P1$ remonte en mode user, il exécute "*do_notify_resume*".
     - Donnez une modification simple des piles user et système permettant à $P1$ de reprendre dans la fonction "*handler*" puis d'enchaîner sur l'instruction "`move $5,(%eax);`".
     - Expliquez pourquoi cette solution simple décale le pointeur de pile.
     - Expliquez pourquoi l'instruction "`move $5,(%eax);`" a peu de chance de fonctionner correctement.

     Comment résoudre ces 2 problèmes?

6. Donnez la check-list pour ajouter un appel système et le tester.

4.4.2 Appels système dits rapides

L'instruction "`int $0x80`" est jugée lente. Pour les exécutables dynamiques, les appels système sont implantés par les instructions `sysenter` et `sysexit` décrites à la page 9.
Le noyau contient une page contenant le code ci-contre. Elle est ajoutée à une adresse viruelle fixe (ADDR_SYSENTER) au processus utilisateur. Les routines d'appel système appelle la fonction "*kernel_vsyscall*" au lieu d'exécuter l'instruction "`int $80`". L'instruction `sysenter` branche sur la routine système "*sysenter_entry*" présentée sur la figure 29.
Commentez ces 2 routines.

```
 1 kernel_vsyscall:
 2   push    %ecx
 3   push    %edx
 4   push    %ebp
 5   mov     %esp,%ebp
 6   sysenter
 7 sysenter_return:
 8   pop     %ebp
 9   pop     %edx
10   pop     %ecx
11   ret
```

4.5 Travaux dirigés: Création de processus

Sous Unix la seule façon de créer un processus est l'appel système "*fork*".

1. l'appel système "*fork*" n'a pas de paramètre, son vis-à-vis noyau est "*ptregs_fork*" qui est défini dans le fichier "*entry_32.S*" (page 46) et rappellé ci-dessous:

```
 1 ENTRY(sysenter_entry)
 2   movl TSS_sysenter_esp0(%esp),%esp
 3   pushl $(__USER_DS)
 4   pushl %ebp    [userland %esp]
 5   pushfl
 6   pushl $(__USER_CS)
 7   pushl $SYSENTER_RETURN [%userland return addr]
 8   ...
 9   pushl %eax
10   SAVE_ALL
11   cmpl $(nr_syscalls), %eax
12   jae syscall_badsys
13   call *sys_call_table(,%eax,4)
14   movl %eax,EAX(%esp)
15   ....
16   /* if something modifies registers it
17    * must also disable sysexit */
18   movl EIP(%esp), %edx
19   movl OLDESP(%esp), %ecx
20   xorl %ebp,%ebp
21   sysexit
```

FIGURE 29: Les grandes lignes de la routine "*sysenter_entry*"

```
713 ptregs_fork:
714     leal 4(%esp),%eax
715     jmp sys_fork
```

Son implantation du coté noyau est donc la fonction "*sys_fork*" ci-dessous, elle a un paramètre de type "`pt_regs`". Donnez la structure de ce type.

```
int sys_fork(struct pt_regs *regs)
{
  return do_fork(SIGCHLD, regs->sp, regs, 0, NULL, NULL);
}
```

2. Etudiez les fonctions "`do_fork`" et "`copy_process`". Qui du fils ou du père démarrera en premier en mode utilisateur.

3. Détaillez précisément les lignes 256-264 du fichier "*process.c*".

   (a) Comment $\emptyset$ est renvoyé dans le fils?

   (b) Où nait le fils?

   (c) Quelle est l'utilité de la mise à jour de esp0?

4.6 Travaux dirigés: Programmation système.

Les questions suivantes permettent d'établir les paradigmes de la programmation système en général et de Linux en particulier. On y répondra en envisageant d'abord un noyau pour une machine mono processeur, puis un noyau pour une machine multiprocesseur.

1. Quand est-on en programmation système?
2. Quel est le mode du processeur?
3. Quand ont lieu les commutations de processus?
4. Dans quelle condition les instructions de la séquence "`I1; I2;`" ne seront pas exécutées séquentiellement?
5. Comment les rendre séquentielles?

arch/x86/include/asm/processor.h

```
416 struct thread_struct {
    ... ... ...
419     unsigned long       sp0;
420     unsigned long       sp;
    ... ... ...
430     unsigned long       ip;
431     unsigned long       fs;
432     unsigned long       gs;
    ... ... ...
471 };
```

include/linux/cred.h

```
115 struct cred {
116     atomic_t    usage;
117     uid_t       uid;        /* real UID of the task */
118     gid_t       gid;        /* real GID of the task */
119     uid_t       suid;       /* saved UID of the task */
120     gid_t       sgid;       /* saved GID of the task */
121     uid_t       euid;       /* effective UID of the task */
122     gid_t       egid;       /* effective GID of the task */
    ... ... ...
143 };
```

include/linux/mm_types.h

```
194 struct mm_struct {
195     struct vm_area_struct * mmap;       /* list of VMAs */
    ... ... ...
206     pgd_t * pgd;
208     atomic_t mm_count;  /* How many references to "struct mm_struct" */
    ... ... ...
227     unsigned long total_vm, locked_vm, shared_vm, exec_vm;
228     unsigned long stack_vm, reserved_vm, def_flags, nr_ptes;
229     unsigned long start_code, end_code, start_data, end_data;
230     unsigned long start_brk, brk, start_stack;
231     unsigned long arg_start, arg_end, env_start, env_end;
    ... ... ...
281 };
```

include/linux/sched.h

```
640 struct user_struct {
641     atomic_t __count;   /* reference count */
642     atomic_t processes; /* How many processes does this user have? */
643     atomic_t files;     /* How many open files does this user have? */
644     atomic_t sigpending;    /* How many pending signals does this user have? */
    ... ... ...
665     uid_t uid;
    ... ... ...
675 };
    ... ... ...
1117 struct task_struct {
1118     volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    ... ... ...
1132     int prio, static_prio, normal_prio;
    ... ... ...
1172     struct mm_struct *mm, *active_mm;
    ... ... ...
1184     pid_t pid;
    ... ... ...
1190     /*
1191      * pointers to (original) parent process, youngest child, younger sibling,
1192      * older sibling, respectively.  (p->father can be replaced with
1193      * p->real_parent->pid)
1194      */
1195     struct task_struct *real_parent; /* real parent process */
1196     struct task_struct *parent; /* recipient of SIGCHLD, wait4() reports */
1197     /*
1198      * children/sibling forms the list of my natural children
1199      */
1200     struct list_head children;  /* list of my children */
1201     struct list_head sibling;   /* linkage in my parent's children list */
1202     struct task_struct *group_leader;   /* threadgroup leader */
    ... ... ...
1233     cputime_t utime, stime, utimescaled, stimescaled;
    ... ... ...
1248     const struct cred *cred;    /* effective (overridable) subjective task
    ... ... ...
1266 /* CPU-specific state of this task */
1267     struct thread_struct thread;
```

```
1268 /* filesystem information */
1269     struct fs_struct *fs;
1270 /* open file information */
1271     struct files_struct *files;
     ... ... ...
1275     struct signal_struct *signal;
1276     struct sighand_struct *sighand;
     ... ... ...
1432 };
     ... ... ...
1831 extern struct task_struct init_task;
1833 extern struct   mm_struct init_mm;
```

arch/x86/kernel/entry_32.S

```
  81 #define nr_syscalls ((syscall_table_size)/4)
 321 ENTRY(ret_from_fork)
 322     ...
 327     GET_THREAD_INFO(%ebp)
 330     pushl $0x0202           # Reset kernel eflags
 332     popfl
 334     jmp syscall_exit
 359 ENTRY(resume_userspace)
 365     movl TI_flags(%ebp), %ecx
 366     andl $_TIF_WORK_MASK, %ecx    # is there any work to be done on
 367                     # int/exception return?
 368     jne work_pending
 369     jmp restore_all
 519 ENTRY(system_call)
 522     pushl %eax            # save orig_eax
 524     SAVE_ALL
 525     GET_THREAD_INFO(%ebp)
 529     cmpl $(nr_syscalls), %eax
 530     jae syscall_badsys
 532     call *sys_call_table(,%eax,4)
 533     movl %eax,PT_EAX(%esp)        # store the return value
 534 syscall_exit:
 540     movl TI_flags(%ebp), %ecx
 541     testl $_TIF_ALLWORK_MASK, %ecx    # current->work
 542     jne resume_userspace
 545 restore_all:
 546     RESTORE_ALL
 547     iret
 620 work_pending:
 621     testb $_TIF_NEED_RESCHED, %cl
 622     jz work_notifysig
 624     call schedule
 630     movl TI_flags(%ebp), %ecx
 631     andl $_TIF_WORK_MASK, %ecx    # is there any work to be done other
 633     jz restore_all
 637 work_notifysig:                # deal with pending signals and
 645     call do_notify_resume
 646     jmp resume_userspace
 713 ptregs_fork:
 714     leal 4(%esp),%eax
 715     jmp sys_fork
1189 #include "syscall_table_32.S"
1191 syscall_table_size=(.-sys_call_table)
```

arch/x86/kernel/syscall_table_32.S

```
   1 ENTRY(sys_call_table)
   2     .long sys_restart_syscall
   3     .long sys_exit
   4     .long ptregs_fork
   5     .long sys_read
   6     ...
```

kernel/fork.c

```
 946 static struct task_struct *copy_process(unsigned long clone_flags,
 947                     unsigned long stack_start,
 948                     struct pt_regs *regs,
 949                     unsigned long stack_size,
 950                     int __user *child_tidptr,
 951                     struct pid *pid,
 952                     int trace)
 953 {
 954     int retval;
 955     struct task_struct *p;
     ... ... ...
 980     retval = -ENOMEM;
 981     p = dup_task_struct(current);
     ... ... ...
 999     retval = copy_creds(p, clone_flags);
     ... ... ...
1033     p->utime = cputime_zero;
1034     p->stime = cputime_zero;
1035     p->gtime = cputime_zero;
     ... ... ...
1103     if ((retval = copy_files(clone_flags, p)))
1104         goto bad_fork_cleanup_semundo;
1105     if ((retval = copy_fs(clone_flags, p)))
1106         goto bad_fork_cleanup_files;
1107     if ((retval = copy_sighand(clone_flags, p)))
1108         goto bad_fork_cleanup_fs;
     ... ... ...
1111     if ((retval = copy_mm(clone_flags, p)))
1112         goto bad_fork_cleanup_signal;
     ... ... ...
1117     retval = copy_thread(clone_flags, stack_start, stack_size, p, regs);
     ... ... ...
1123         pid = alloc_pid(p->nsproxy->pid_ns);
     ... ... ...
```

```
1136    p->pid = pid_nr(pid);
    ... ... ...
1269    return p;
    ... ... ...
1341 long do_fork(unsigned long clone_flags,
1342          unsigned long stack_start,
1343          struct pt_regs *regs,
1344          unsigned long stack_size,
1345          int __user *parent_tidptr,
1346          int __user *child_tidptr)
1347 {
1348    struct task_struct *p;
1349    int trace = 0;
1350    long nr;
    ... ... ...
1390    p = copy_process(clone_flags, stack_start, regs, stack_size,
1391             child_tidptr, NULL, trace);
    ... ... ...
1396    if (!IS_ERR(p)) {
    ... ... ...
1401        nr = task_pid_vnr(p);
    ... ... ...
1430            wake_up_new_task(p, clone_flags);
    ... ... ...
1442    } else {
1443        nr = PTR_ERR(p);
1444    }
1445    return nr;
1446 }
```

arch/x86/kernel/process_32.c

```
248 int copy_thread(unsigned long clone_flags, unsigned long sp,
249    unsigned long unused,
250    struct task_struct *p, struct pt_regs *regs)
251 {
252    struct pt_regs *childregs;
253    struct task_struct *tsk;
254    int err;

256    childregs = task_pt_regs(p);
257    *childregs = *regs;
258    childregs->ax = 0;
259    childregs->sp = sp;

261    p->thread.sp = (unsigned long) childregs;
262    p->thread.sp0 = (unsigned long) (childregs+1);

264    p->thread.ip = (unsigned long) ret_from_fork;
    ... ... ...
298    return err;
299 }
```

4.7 Travaux pratiques 1: Mode utilisateur

Attention: Les questions de ce TP sont à faire **séquentiellement**. De manière concrête, la question i ne sera pas notée si les questions 1 à $i-1$ n'ont pas été faites correctement.Pour ce TP sont fournis:

- "*my_ read_ test.c*" programme C pour tester la $1^{ère}$ question.
- "*my_ test.c*" programme C pour faire la $2^{ième}$ question.
- "*dummy_ 32.c*" programme C pour faire la $3^{ième}$ question.

ATTENTION: ce TP doit être fait sur une machine Linux 32 bits.

1. En vous inspirant de la fonction démacrofiée "***open()***" vue en TD, écrire dans un fichier "*read.S*", le code assembleur correspondant à une fonction "***my_read(fid,buffer,size)***". Celle-ci fait la même chose que "***read(int,char*,int)***" mais affiche en plus un point chaque fois qu'elle est appelé.
   Le tester avec le programme 'C' "*my_ read_ test.c*".
   ATTENTION: une fois validé, conservez ce fichier, il sera utilisé dans tous les autres TP.
2. Enlevez l'affichage du point dans votre appel système et générez un fichier de 200 mo dans le répertoire "*/tmp*".
   (a) Complètez la fonction "***my_read***" du fichier "*my_ test.c*".
   (b) Compilez "*my_ test.c*" en mettant la taille du buffer à 1 octet et donnez le temps que met le programme pour lire le fichier de 200 mo.
   (c) Compilez "*my_ test.c*" en mettant la taille du buffer à 64 ko et donnez le temps que met le programme pour lire le fichier de 200 mo.
3. Compilez sans le modifier le programme ci-dessous, il doit écrive "bonjour" puis "au revoir".

```
int main() {
  write(1,"bonjour\n",8);
  _start();
}
int _start() { write(1,"au revoir\n",10); }
```

   Corrigez le pour supprimer le "segfault" en modifiant le corps de la function "***main***".
4. Regardez le programme "*dummy_ 32.c*".
   (a) Compilez le et exécutez le.
   (b) Modifiez le pour qu'il écrive 512.

(c) Changez le "*main*" en "*dummy*", compilez le, exécutez le et expliquez la faute de segmentation.

(d) Ajoutez un "*exit*" À LA FIN POUR ÉVITER LA FAUTE DE SEGMENTATION.

4.8 Travaux pratiques 2: Mode système

Attention: Les questions de ce TP sont à faire **séquentiellement**. De manière concrête, la question i ne sera pas notée si les questions 1 à $i-1$ n'ont pas été faites correctement.

1. Implantez l'appel système "*essai*" qui a en paramètre un entier et qui renvoie cet entier plus 1. Pour le tester, le programme en mode utilisateur doit utiliser le programme assembleur fait au TP précédent et prendre en argument sur la ligne de commande l'entier à incrémenter.
2. Que se passe-t-il si on lui donne un nombre négatif? Modifiez le programme C de teste (pas l'assembleur) pour qu'il écrive un résultat correct également pour les nombres négatifs.
3. Implantez l'appel système "*essaib(int* a,int b)*" qui ajoute b à la valeur mémoire pointée par a ($*a+ = b$) si il n'y a pas de dépassement de capacité. On implantera tous les contrôles et on renverra des codes d'erreur significatifs.
4. Modifiez l'appel système "*fork_ise()*" pour que le fils recoive le pid du père au lieu de zéro.
5. Modifiez l'appel système "*fork_ise()*" pour faire démarrer le processus fils avant le processus père.
6. Modifiez l'appel système "*fork_ise()*" pour que le programme suivant fonctionne:

```
void fils() { printf("FILS\n"); _exit(0; }
int main()  { fork_ise(fils); printf("PERE\n"); return 0; }
```

5 Système de fichiers

5.1 Système de fichiers

Un système de fichiers est une structure de données implantée sur les secteurs d'un disque pour créer une organisation arborescente des données. Les

super-bloc	inodes	données

a) structure générale d'un volume

adr		taille
0	numéro du 1^{er} secteur data	2
4	nombre de secteurs du volume	4
8	nombre d'éléments dans la table des bloc libres	2
12	table de 50 blocs libres, le dernier est un bloc d'indirection	200
212	nombre d'éléments dans la table des inodes libres	2
216	table de 100 inodes libres	200
432	nombre de secteurs données libres	4
436	nombre d'inodes libres	4
492	état (clean)	4
496	version du sysème de fichier	4
500	taille cluster (1=512o, 2=1024o)	4

b) super-bloc

adr		taille
0	mode & type	2
2	nombres de lien	2
4	utilisateur	2
6	group	2
8	taille	4
12	données (3*13)	40
52	date du dernier accès	4
56	date de la dernière modificatition	4 4
60	date de création	4

c) inode

adr		taille
0	nombre d'éléments dans la table table des bloc libres	2
2	table de 50 blocs libres, le dernier est un nouveau bloc d'indirection	200

d) structure d'un bloc libre d'indirection

adr		taille
0	inode	2
2	nom	14

e) entrée de répertoire

- Dans la table d'inode, les inodes sont rangés tout les 64 octects.
- Le champ "données" de l'inode contient:
  - Pour un fichier régulier, 13 numéros (sur 3 octets) de bloc données. Les 10 premiers sont directs, le $11^{ième}$ est indirect (1 niveau), le $12^{ième}$ est indirect (2 niveaux), le $13^{ième}$ (3 niveaux).
  - Pour un fichier périphérique, il contient son numéro sur 4 octets.
  - Pour un lien symbolique, la valeur du lien si le lien n'est pas trop long.
- Un bloc d'indirection est un tableau de numéros (sur 4 octets) de bloc de la zone données.

FIGURE 30: Système de fichiers Unix système V.

créer répertoire	dir, name, att	ajoute un répertoire vide de nom "name" avec les attributs "att" dans le répertoire "dir"
détruire répertoire		
créer fichier		
détruire fichier		

TABLE 2: Opérations sur les répertoire d'un système de fichiers.

opérations de base sur un système de fichiers sont présentées sur la table 1. Les critères de qualité d'un système de fichiers sont:

1. L'efficacité des opérations de base c'est-à-dire le temps nécessaire pour les effectuer.
2. L'efficacité du stockage c'est-à-dire le rapport: le nombre d'octets stockés pour l'utilisateur sur le nombre d'octets stockés sur le disque.
3. La tolérance aux pannes. Comme toute bonne mécanique, un disque est sujet à de petites pannes qui se caractérisent par la perte de quelques secteurs. La tolérance est la mesure de l'impact de telles pannes sur l'ensemble du système de fichiers.

A titre d'exemple vous trouverez sur la figure 14 la description du système de fichiers MSDOS 16 bits et sur la figure 30 le système de fichiers Unix système V.

Les fichiers pour la structure des système de fichiers sont une suite séquentielle d'octets, les répertoires sont des fichiers comme les autres mais possédant une structure interne propre. Ce sont eux qui implantent l'organisation arborescente. Ces fichiers répertoires ne doivent pas être corrompus, c'est pourquoi les opérations de bases sont enrichies de primitives pour modifier les fichiers répertoires. Celles-ci sont présentées sur la table 2.

5.2 Exercices

1 Considérons un disque dont les secteurs physiques font 512 octets.

type	disque	cluster	taille max	lecture		efficacité	
				premier	dernier	512 o	512 ko
fat16	64 mo						
fat16	1 go						
fat16	2 go						
sys. V	T go	2 (1 ko)					

TABLE 3: Comparaison des systèmes de fichiers fat16 et SysV

1. Complétez la table 3.
   - Dans la colonne "cluster", on donnera le nombre de secteurs physiques par cluster et sa taille en octets.
   - Dans la colonne "taille max", on donnera la taille maximale d'un fichier.
   - Dans la colonne "lecture", on donnera le nombre d'accès disque nécessaire dans le pire cas pour lire le premier et le dernier octet d'un fichier.
     On supposer que la fat et l'entrée du répertoire sont en mémoire pour le système de fichiers fat16, et que l'inode du fichier est en mémoire pour le système de fichiers SysV.
   - Dans la colonne "efficacité", on donnera l'efficacité de stockage.
2. Etudiez la tolérance à la perte d'un secteur.
3. Etudiez l'efficacité d'accès.

2 Donnez l'algorithme de "*fsck*".

5.3 Travaux Dirigés: Etude des structures de données du noyau

La gestion des systèmes de fichiers est organisée autour de plusieurs structures de données dont les principales font l'objet des chapitres suivants.

5.3.1 La structure "super_bloc"

Cette structure est définie à la ligne 738 du fichier "*fs.h*" Elle décrit de manière générique le super bloc d'un système de fichiers. Etudiez cette structure puis répondez aux questions suivantes:

1. Comment est fait le lien entre cette structure et le super bloc physique?
2. Comment est fait le lien entre cette structure et le système de fichiers?
3. Où sont stockés les super blocs?

5.3.2 La structure "vfsmount"

La structure "vfsmount" est définie dans le fichier "*mount.h*" et elle décrit les points de montage. Etudiez cette structure puis répondez aux questions suivantes:

1. Quel est le principe du montage (rôle de "mnt_mountpoint" et "mnt_root")?
2. Précisez la signification des flags "MNT_...".

5.3.3 La structure "inode"

Cette structure est définie à la ligne 713 du fichier "*fs.h*". Elle correspond à un fichier de n'importe quel système de fichiers de n'importe quel type. Etudiez cette structure puis répondez aux questions suivantes:

1. Quelle est la clef d'un inode?
2. Donnez la signification des paramètres des fonctions: "create", "mknod", lignes 1514 et 1521. de "*fs.h*"
3. Quand est allouée une telle structure?

5.3.4 La structure "dentry"

Cette structure est définie à la ligne 89 du fichier "*dcache.h*". Elle correspond aussi à un fichier. Etudiez cette structure puis répondez aux questions suivantes:

1. Quelle est la clef d'un dentry?
2. Que "cache" cette structure de données?
3. Pour qelles opérations est-elle utilisée?

5.3.5 Les structures "page" et "address_space"

La structure "page" décrit un buffer mémoire d'un bloc ou plusieurs blocs disque. Elle est décrite dans le fichier "*mm_types.h*". La structure "address_space" décrit un ensemble de buffers. Elle est décrite à la ligne 613 du fichier "*fs.h*". Etudiez ces structures puis répondez aux questions suivantes:

1. Le champ "index" de la structure "page" est la clef, à quoi correspond-elle?
2. Le champ "flags" de la structure "page" peux prendre, entre autres, les valeurs ci-dessous, que signifient-elles?

   #define PG_error 0x01
   #define PG_uptodate 0x04
   #define PG_dirty 0x08
3. Les structures "*page*" sont stockées dans un tableau "mem_map". Expiquez la macro "*page_address*", ci-dessous:

   #define PAGE_OFFSET 0xC0000000
   #define __va(x) ((void *)((unsigned long)(x)\
   + PAGE_OFFSET))
   #define page_to_pfn(page) ((unsigned long)((page) - mem_map))
   #define page_address(page) __va(page_to_pfn(page) « PAGE_SHIFT)

   Précisez en particulier l'importance de la macro "*__va*".
4. A quoi correspond la structure "address_space" incluse dans un inode.
5. Donnez l'algorithme principal de la lecture du $N^{ième}$ caractère d'un fichier.
6. Donnez l'algorithme principal de la fonction "*mmap*".

5.3.6 La structure "file"

Cette structure est définie à la ligne 899 du fichier "*fs.h*". Elle correspond à un fichier ouvert d'un processus. Etudiez cette structure puis répondez aux questions suivantes:

1. A-t-elle un équivalent sur le disque?
2. Est-elle indépendante du système de fichiers?
3. Quel est son rôle?
4. Quel est l'identifiant d'un fichier en mode USER? A quoi correspond-il? (Regardez les structures "task_struct" et "files_struct".

include/linux/mount.h

```
23 #define MNT_NOSUID 0x01
24 #define MNT_NODEV  0x02
25 #define MNT_NOEXEC 0x04
26 #define MNT_NOATIME    0x08
```

```
   ... ... ...
29 #define MNT_READONLY   0x40    /* does the user want this to be r/o? */
   ... ... ...
39 struct vfsmount {
40    struct list_head mnt_hash;
41    struct vfsmount *mnt_parent;    /* fs we are mounted on */
42    struct dentry *mnt_mountpoint;  /* dentry of mountpoint */
43    struct dentry *mnt_root;        /* root of the mounted tree */
44    struct super_block *mnt_sb;     /* pointer to superblock */
45    struct list_head mnt_mounts;    /* list of children, anchored here */
47    int mnt_flags;
   ... ... ...
73 };
```

include/linux/dcache.h

```
33 struct qstr {
34    unsigned int hash;
35    unsigned int len;
36    const unsigned char *name;
37 };
   ... ... ...
89 struct dentry {
   ... ... ...
94    struct inode *d_inode;      /* Where the name belongs to - NULL is
95                                 * negative */
   ... ... ...
100   struct hlist_node d_hash;    /* lookup hash list */
101   struct dentry    *d_parent;  /* parent directory */
102   struct qstr       d_name;
   ... ... ...
112   struct list_head d_subdirs; /* our children */
   ... ... ...
119   unsigned char d_iname[DNAME_INLINE_LEN_MIN];    /* small names */
120 };
```

include/linux/mm_types.h

```
40 struct page {
41    unsigned long flags;    /* Atomic flags, some possibly
42                             * updated asynchronously */
   ... ... ...
56    unsigned long private;  /* Mapping-private opaque data:
57                             * usually used for buffer_heads
58                             * if PagePrivate set; used for
59                             * swp_entry_t if PageSwapCache;
60                             * indicates order in the buddy
61                             * system if PG_buddy is set.
62                             */
63    struct address_space *mapping; /* If low bit clear, points to
64                             * inode address_space, or NULL.
65                             * If page mapped as anonymous
66                             * memory, low bit is set, and
67                             * it points to anon_vma object:
68                             * see PAGE_MAPPING_ANON below.
69                             */
70        };
   ... ... ...
78        pgoff_t index;      /* Our offset within mapping. */
   ... ... ...
101 };
```

include/linux/fdtable.h

```
42 /* Open file table structure */
43 struct files_struct {
   ... ... ...
57    struct file * fd_array[NR_OPEN_DEFAULT];
58 };
```

include/linux/fs.h

```
613 struct address_space {
614    struct inode            *host;      /* owner: inode, block_device */
615    struct radix_tree_root  page_tree;  /* radix tree of all pages */
616    spinlock_t              tree_lock;  /* and lock protecting it */
   ... ... ...
630 } __attribute__((aligned(sizeof(long))));
   ... ... ...
713 struct inode {
   ... ... ...
718    unsigned long       i_ino;
719    atomic_t            i_count;
720    unsigned int        i_nlink;
721    uid_t               i_uid;
722    gid_t               i_gid;
723    dev_t               i_rdev;
   ... ... ...
729    struct timespec     i_atime;
730    struct timespec     i_mtime;
731    struct timespec     i_ctime;
   ... ... ...
735    umode_t             i_mode;
   ... ... ...
739    const struct inode_operations   *i_op;
740    const struct file_operations    *i_fop; /* former ->i_op->default_file_ops */
741    struct super_block  *i_sb;
743    struct address_space    *i_mapping;
   ... ... ...
778 };
   ... ... ...
899 struct file {
   ... ... ...
908    struct path     f_path;
```

```
 909 #define f_dentry   f_path.dentry
 910 #define f_vfsmnt   f_path.mnt
     ... ... ...
 913     atomic_long_t   f_count;
 915     fmode_t         f_mode;
 916     loff_t          f_pos;
     ... ... ...
 936 };
     ... ... ...
1301 extern struct list_head super_blocks;
     ... ... ...
1306 struct super_block {
1307     struct list_head    s_list;     /* Keep this first */
     ... ... ...
1311     unsigned char       s_dirt;
     ... ... ...
1314     const struct super_operations   *s_op;
     ... ... ...
1320     struct dentry       *s_root;
     ... ... ...
1331     struct list_head    s_inodes;   /* all inodes */
1332     struct list_head    s_dirty;    /* dirty inodes */
     ... ... ...
1351     void            *s_fs_info; /* Filesystem private info */
     ... ... ...
1380 };
     ... ... ...
1484 struct file_operations {
     ... ... ...
1487     ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
1488     ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
     ... ... ...
1491     int (*readdir) (struct file *, void *, filldir_t);
     ... ... ...
1496     int (*mmap) (struct file *, struct vm_area_struct *);
     ... ... ...
1511 };
1513 struct inode_operations {
1514     int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
1515     struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
1516     int (*link) (struct dentry *,struct inode *,struct dentry *);
1517     int (*unlink) (struct inode *,struct dentry *);
1518     int (*symlink) (struct inode *,struct dentry *,const char *);
1519     int (*mkdir) (struct inode *,struct dentry *,int);
1520     int (*rmdir) (struct inode *,struct dentry *);
1521     int (*mknod) (struct inode *,struct dentry *,int,dev_t);
     ... ... ...
1540 };
     ... ... ...
1556 struct super_operations {
1557     struct inode *(*alloc_inode)(struct super_block *sb);
1558     void (*destroy_inode)(struct inode *);
     ... ... ...
1565     void (*write_super) (struct super_block *);
     ... ... ...
1577     ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
1578     ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
     ... ... ...
1581 };
```

5.3.7 Synthèse

Le schéma sur la figure 31 représente une vue simplifiée de la structure de données liant les principaux types tels les "`task_struct`", les piles systèmes, les "`file`" à un instant donné.
Complètez cette figure dans les cas suivants:

1. le processus INIT ouvre le fichier I22.
2. le processus P1 "dup" le fichier standard de sortie.
3. le processus P1 ferme le fichier standard d'erreur.
4. le processus P2 "fork" et son fils est le processus X.
5. le processus P2 "thread" et la thread est le processus X.

5.4 Travaux Dirigés: Etude de l'appel système "*open(...)*".

5.4.1 l'appel système "*open(...)*"

Le point d'entrée de l'appel système "***open***" est la fonction système "***sys_open***" qui se trouve à la ligne 1050 du fichier "*fs/open.c*".

1. Donnez l'algorithme de "***do_sys_open***" commencant à la ligne 1028 du fichier "*fs/open.c*".
2. Indiquez les actions que fait la fonction "***do_filp_open***".

5.4.2 Partie indépendante du système de fichier

Une grande partie des traitements est indépendante du système de fichiers, ceux-ci constituent le Virtual File System. Les fonctions du VFS comme les points d'entrée des systèmes de fichiers utilisent les structures suivantes pour

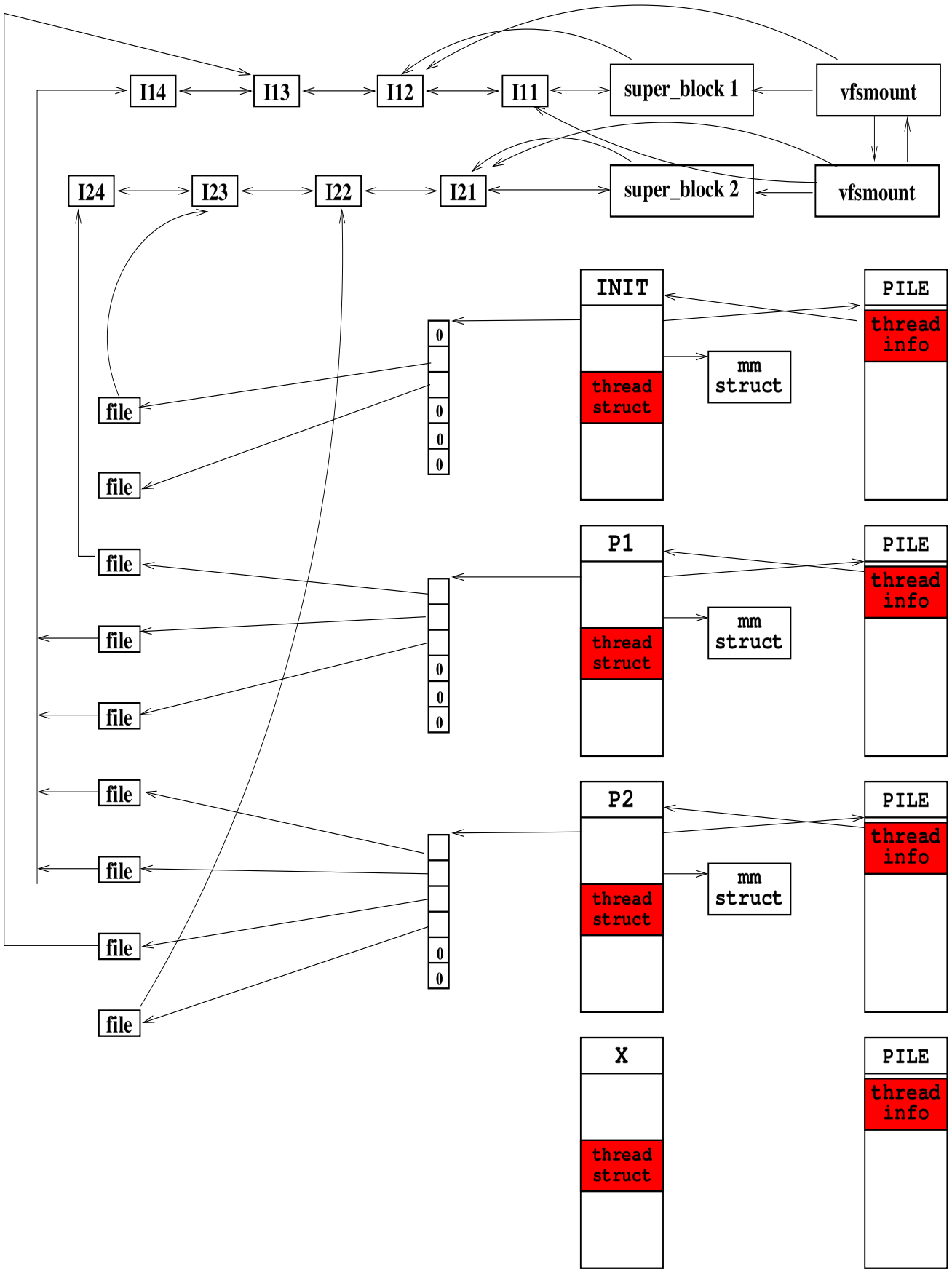


FIGURE 31: Structures de données

se passer les paramètres.

```
struct path {
  struct vfsmount *mnt;
  struct dentry *dentry;
};

struct nameidata {
  struct dentry   *dentry;    /* le répertoire */
  struct vfsmount *mnt;       /* les mounts autorisés */
  struct qstr      last;      /* le path à chercher dans dentry */
  unsigned int     flags;
  ...
};
```

1. Après quelques indirections, on arrive à la fonction "*do_path_lookup*" à la ligne 1020 du fichier "*fs/namei.c*". Quel est son rôle?
2. Après quelques indirections on arrive à la fonction "*__link_path_walk*" à la ligne 829 du fichier "*fs/namei.c*".
   (a) Expliquez ce qu'elle fait?
   (b) Où est fait l'avancée dans le chemin?
3. Donnez l'algorithme de la fonction "*do_lookup*" à la ligne 787 du fichier "*fs/namei.c*". A quelle ligne les spécificités du système de fichiers sont-elles prises en compte?

5.4.3 Partie "ext2"

Etudiez la fonction "*ext2_lookup*" sachant que la structure qui décrit le format interne d'un répertoire ext2 est donnée ci-dessous, puis répondez aux questions suivantes:

```
#define EXT2_NAME_LEN 255
struct ext2_dir_entry {
  __le32  inode;          /* Inode number */
  __le16  rec_len;        /* Directory entry length */
  __le16  name_len;       /* Name length */
  char    name[EXT2_NAME_LEN];    /* File name */
};
```

1. Par quel mécanisme est appelée la fonction "*ext2_lookup*"?
2. Dans quelles conditions sera appelée la fonction "*ext2_lookup*"?

fs/open.c

```
1028 long do_sys_open(int dfd, const char __user *filename, int flags, int mode)
1029 {
1030     char *tmp = getname(filename);
1031     int fd = PTR_ERR(tmp);
1033     if (!IS_ERR(tmp)) {
1034         fd = get_unused_fd_flags(flags);
1035         if (fd >= 0) {
1036             struct file *f = do_filp_open(dfd, tmp, flags, mode, 0);
1037             if (IS_ERR(f)) {
1038                 put_unused_fd(fd);
1039                 fd = PTR_ERR(f);
1040             } else {
1041                 fsnotify_open(f->f_path.dentry);
1042                 fd_install(fd, f);
1043             }
1044         }
1045         putname(tmp);
1046     }
1047     return fd;
1048 }
1050 SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, int, mode)
1051 {
1052     long ret;
    ... ... ...
1057     ret = do_sys_open(AT_FDCWD, filename, flags, mode);
    ... ... ...
1060     return ret;
1061 }
```

fs/namei.c

```
472 static struct dentry * real_lookup(struct dentry * parent,
473                         struct qstr * name, struct nameidata *nd) {
474     struct dentry * result;
475     struct inode *dir = parent->d_inode;
477     mutex_lock(&dir->i_mutex);
    ... ... ...
492     result = d_lookup(parent, name);
493     if (!result) {
494         struct dentry *dentry;
    ... ... ...
501         dentry = d_alloc(parent, name);
502         result = dir->i_op->lookup(dir, dentry, nd);
    ... ... ...
510         }
512         mutex_unlock(&dir->i_mutex);
513         return result;
514     }
    ... ... ...
520     mutex_unlock(&dir->i_mutex);
    ... ... ...
526     return result;
527 }
    ... ... ...
789 static int do_lookup(struct nameidata *nd, struct qstr *name,
```

```
790             struct path *path)
791 {
792     struct vfsmount *mnt = nd->path.mnt;
793     struct dentry *dentry = __d_lookup(nd->path.dentry, name);
795     if (!dentry)
796         goto need_lookup;
    ... ... ...
799 done:
800     path->mnt = mnt;
801     path->dentry = dentry;
802     __follow_mount(path);
803     return 0;
805 need_lookup:
806     dentry = real_lookup(nd->path.dentry, name, nd);
807     if (IS_ERR(dentry))
808         goto fail;
809     goto done;
    ... ... ...
819 fail:
820     return PTR_ERR(dentry);
821 }
823 /*
824  * Name resolution.
825  * This is the basic name resolution function, turning a pathname into
826  * the final dentry. We expect 'base' to be positive and a directory.
827  *
828  * Returns 0 and nd will have valid dentry and mnt on success.
829  * Returns error and drops reference to input namei data on failure.
830  */
831 static int __link_path_walk(const char *name, struct nameidata *nd)
832 {
833     struct path next;
834     struct inode *inode;
835     int err;
836     unsigned int lookup_flags = nd->flags;
    ... ... ...
843     inode = nd->path.dentry->d_inode;
    ... ... ...
848     for(;;) {
849         unsigned long hash;
850         struct qstr this;
851         unsigned int c;
    ... ... ...
863         this.name = name;
864         c = *(const unsigned char *)name;
866         hash = init_name_hash();
867         do {
868             name++;
869             hash = partial_name_hash(c, hash);
870             c = *(const unsigned char *)name;
871         } while (c && (c != '/'));
872         this.len = name - (const char *) this.name;
873         this.hash = end_name_hash(hash);
    ... ... ...
887         if (this.name[0] == '.') switch (this.len) {
888             default:
889                 break;
890             case 2:
891                 if (this.name[1] != '.')
892                     break;
893                 follow_dotdot(nd);
894                 inode = nd->path.dentry->d_inode;
896             case 1:
897                 continue;
898         }
    ... ... ...
910         err = do_lookup(nd, &this, &next);
    ... ... ...
915         inode = next.dentry->d_inode;
    ... ... ...
919         if (inode->i_op->follow_link) {
920             err = do_follow_link(&next, nd);
    ... ... ...
923             err = -ENOENT;
924             inode = nd->path.dentry->d_inode;
925             if (!inode)
926                 break;
927         }
    ... ... ...
930         err = -ENOTDIR;
931         if (!inode->i_op->lookup)
932             break;
    ... ... ...
1011    }
    ... ... ...
1014    return err;
1015 }
    ... ... ...
1024 static int do_path_lookup(int dfd, const char *name,
1025                 unsigned int flags, struct nameidata *nd)
1026 {
    ... ... ...
1036    if (*name=='/') {
1038        nd->path = fs->root;
    ... ... ...
1042    } else {
1044        nd->path = fs->pwd;
    ... ... ...
1048    }
    ... ... ...
1072    retval = path_walk(name, nd);
    ... ... ...
1077    return retval;
    ... ... ...
1082 }
```

fs/ext2/namei.c

```
 57 static struct dentry *ext2_lookup(struct inode * dir, struct dentry *dentry, struct nameidata *nd)
 58 {
 59     struct inode * inode;
 60     ino_t ino;
    ... ... ...
 65     ino = ext2_inode_by_name(dir, &dentry->d_name);
    ... ... ...
 68         inode = ext2_iget(dir->i_sb, ino);
    ... ... ...
 72     return d_splice_alias(inode, dentry);
 73 }
    ... ... ...
373 const struct inode_operations ext2_dir_inode_operations = {
374     .create     = ext2_create,
375     .lookup     = ext2_lookup,
376     .link       = ext2_link,
    ... ... ...
391 };
```

fs/ext2/dir.c

```
193 static struct page * ext2_get_page(struct inode *dir, unsigned long n,
194                    int quiet)
195 {
196     struct address_space *mapping = dir->i_mapping;
197     struct page *page = read_mapping_page(mapping, n, NULL);
    ... ... ...
205     return page;
    ... ... ...
210 }
    ... ... ...
352 /*
353  *  ext2_find_entry()
354  *
355  * finds an entry in the specified directory with the wanted name. It
356  * returns the page in which the entry was found, and the entry itself
357  * (as a parameter - res_dir). Page is returned mapped and unlocked.
358  * Entry is guaranteed to be valid.
359  */
360 struct ext2_dir_entry_2 *ext2_find_entry (struct inode * dir,
361             struct qstr *child, struct page ** res_page)
362 {
363     const char *name = child->name;
364     int namelen = child->len;
365     unsigned reclen = EXT2_DIR_REC_LEN(namelen);
366     unsigned long start, n;
367     unsigned long npages = dir_pages(dir);
368     struct page *page = NULL;
369     struct ext2_inode_info *ei = EXT2_I(dir);
370     ext2_dirent * de;
371     int dir_has_error = 0;
    ... ... ...
376     /* OFFSET_CACHE */
377     *res_page = NULL;
379     start = ei->i_dir_start_lookup;
380     if (start >= npages)
381         start = 0;
383     n = start;
384     do {
385         char *kaddr;
386         page = ext2_get_page(dir, n, dir_has_error);
388             kaddr = page_address(page);
389             de = (ext2_dirent *) kaddr;
390             kaddr += ext2_last_byte(dir, n) - reclen;
391             while ((char *) de <= kaddr) {
    ... ... ...
398                 if (ext2_match (namelen, name, de))
399                     goto found;
400                 de = ext2_next_entry(de);
401             }
402             ext2_put_page(page);
    ... ... ...
406         if (++n >= npages)
407             n = 0;
    ... ... ...
417     } while (n != start);
418 out:
419     return NULL;
421 found:
422     *res_page = page;
423     ei->i_dir_start_lookup = n;
424     return de;
425 }
    ... ... ...
439 ino_t ext2_inode_by_name(struct inode *dir, struct qstr *child)
440 {
441     ino_t res = 0;
442     struct ext2_dir_entry_2 *de;
443     struct page *page;
445     de = ext2_find_entry (dir, child, &page);
446     if (de) {
447         res = le32_to_cpu(de->inode);
448         ext2_put_page(page);
449     }
450     return res;
451 }
```

5.5 Travaux pratiques

Attention: Les questions de ce TP sont à faire **séquentiellement**. De manière concrête, la question i ne sera pas notée si les questions 1 à $i-1$ n'ont pas été faites correctement.

1. Ajoutez au noyau un appel système "`my_stat(int fl)`" qui gère 4 compteurs $mycnt_i$, i appartenant à $\{1,2,3,4\}$. S'inspirez de ce qui a été fait pendant les TPs précédents pour l'appel en assembleur et le traitement des valeurs de retour. Son fonctionnement sera:
   - fl=0: renvoie 0 et réinitialise les compteurs à 0.
   - fl=1: renvoie $mycnt_1$
   - fl=2: renvoie $mycnt_2$
   - ...
2. Faites que $mycnt_1$ compte le nombre d'inodes recherchés par "*open_namei*". On fera un programme de test qui prend en argument un nom de fichier.
3. Faites que $mycnt_2$ compte le nombre d'inodes recherchés par "*open_namei*" qu'il a trouvé dans le cache.
4. Faites que $mycnt_3$ compte le nombre d'inodes recherchés par "*open_namei*" sur le disque par le système de fichier ext2.
5. Faites que $mycnt_4$ compte le nombre de pages lues par "*open_namei*" sur le disque. Pour tester ce compteur, on créera:
   - une partition ext2 que l'on formatera.
   - sur cette partition on créera un répertoire "*files*"
   - ce répertoire contiendra les fichiers "*fxxxx*" avec "xxxx" variant de 0000 à 5000.
   - le contenu des fichiers "f_i" sera "i".

   Puis on préparera les questions/démonstrations suivantes:
   (a) Le démontage d'un système de fichiers efface-t-il le dans le cache d'inodes les entrées relatives à ce système de fichiers.
   (b) Préparez une démonstration de l'algorithme de "*ext2_find_entry*".
6. Montrez l'efficacité du cache disque.
7. Montrez l'efficacité du cache des inodes.
8. Créez l'appel système "*myopen[int* i_count, int* d_count]*" qui crée le fichier "*gnat*" dans le répertoire courant, avec les protections "read" et "write" pour tout le monde(0666), seulement si il n'existe pas. Il renverra en valeur de retour un code qui indiquera si la création a eu lieu (0) ou pas (1). Il renverra dans tous les cas, dans "`i_count`", "`d_count`" les compteurs de références de l'inode et de son entrée dans le cache.
   Note: ***On n'utilisera pas les fonctions "sys_open" et "do_sys_open"***