

MEMOIRE

Architecture et Technologie des Ordinateurs

- **Volatile et Non-volatile**
- **Morte et Vive**
- **Adressable par octet**

Une mémoire peut être classée sous plusieurs critères comme la volatilité (volatile, non-volatile), l'accessibilité (accès aléatoire RAM ou séquentiel), l'adressage (par octet ou associatif), la capacité, les performances ... Intéressons-nous à certains de ces critères :

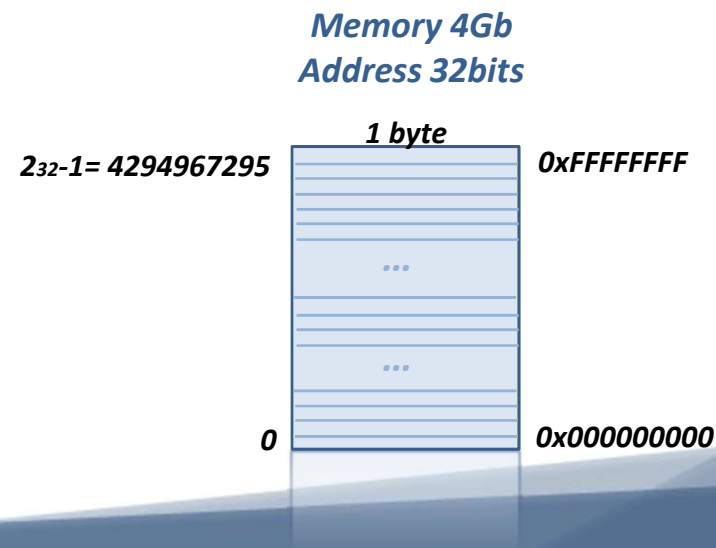
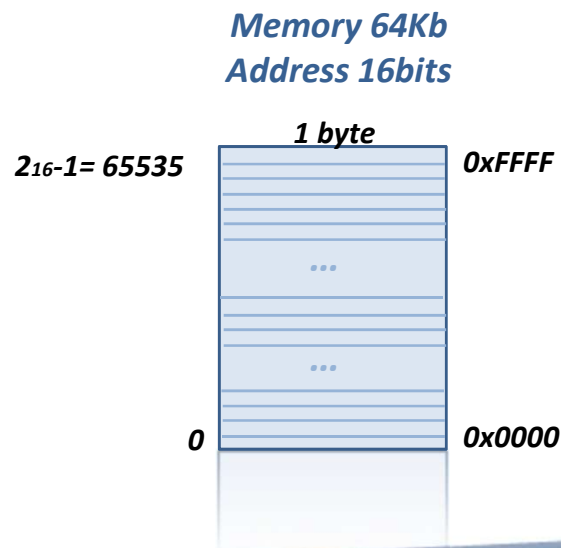
- **Mémoire Volatile** : se dit d'une mémoire ne gardant pas les informations stockées (données ou code) en cas de perte de l'alimentation. Prenons l'exemple de la mémoire principale (SDRAM, DDRAM), mémoires cache, registres processeur.
- **Mémoire Non-volatile** : se dit donc d'une mémoire gardant les informations stockées (données ou code) en cas de perte de l'alimentation. Prenons l'exemple des mémoires de stockage de masse (disque dur, SD-card, DVD, CD, Blue-Ray...), du BIOS...

- Volatile et Non-volatile
- **Morte et Vive**
- Adressable par octet

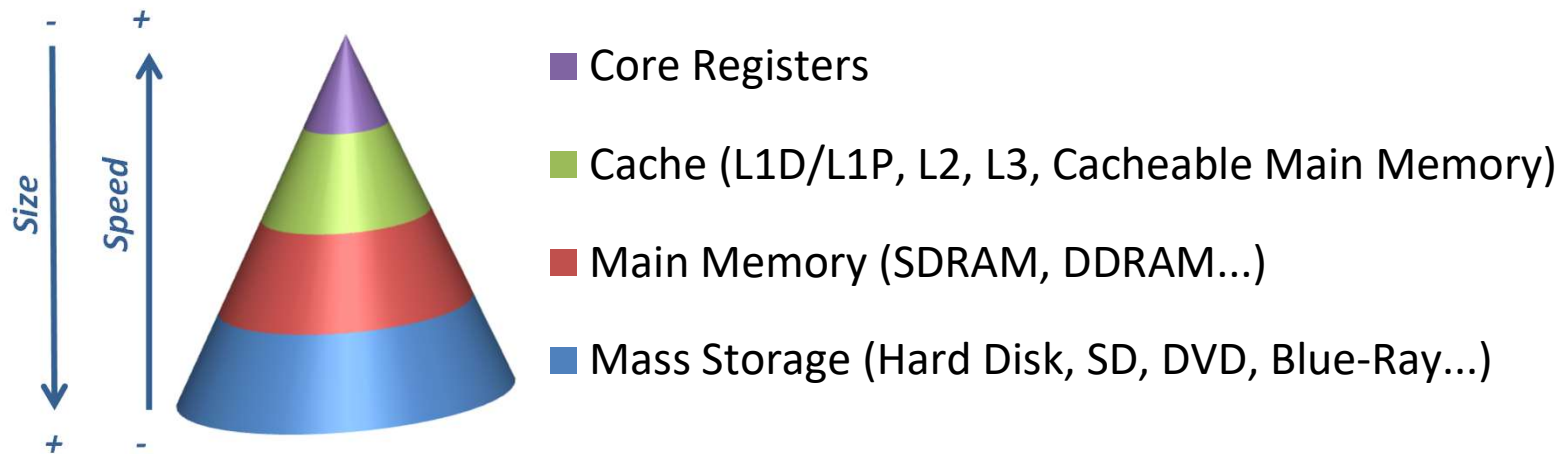
- **Mémoire Morte** : Mémoire morte ou ROM (Read Only Memory) se dit d'une mémoire non-volatile accessible qu'en lecture qui donc préprogrammée. Prenons l'exemple du BIOS, du séquenceur microcode MSROM dans les architectures Nehalem et Sandy Bridge, de nombreux systèmes embarqués...
- **Mémoire Vive** : Mémoire vive ou RAM (Random Access Memory – accès n'importe quelle case sur un temps quasiment fixe) se dit généralement d'une mémoire volatile accessible en lecture et écriture. Par exemple la mémoire principale (SDRAM, DDRAM) d'un ordinateur. Néanmoins, beaucoup d'ambiguïté et d'abus de langage existent autour des termes RAM, vive, ROM ... amenant très souvent à des maladresses et incompréhensions.

- Volatile et Non-volatile
- Morte et Vive
- Adressable par octet

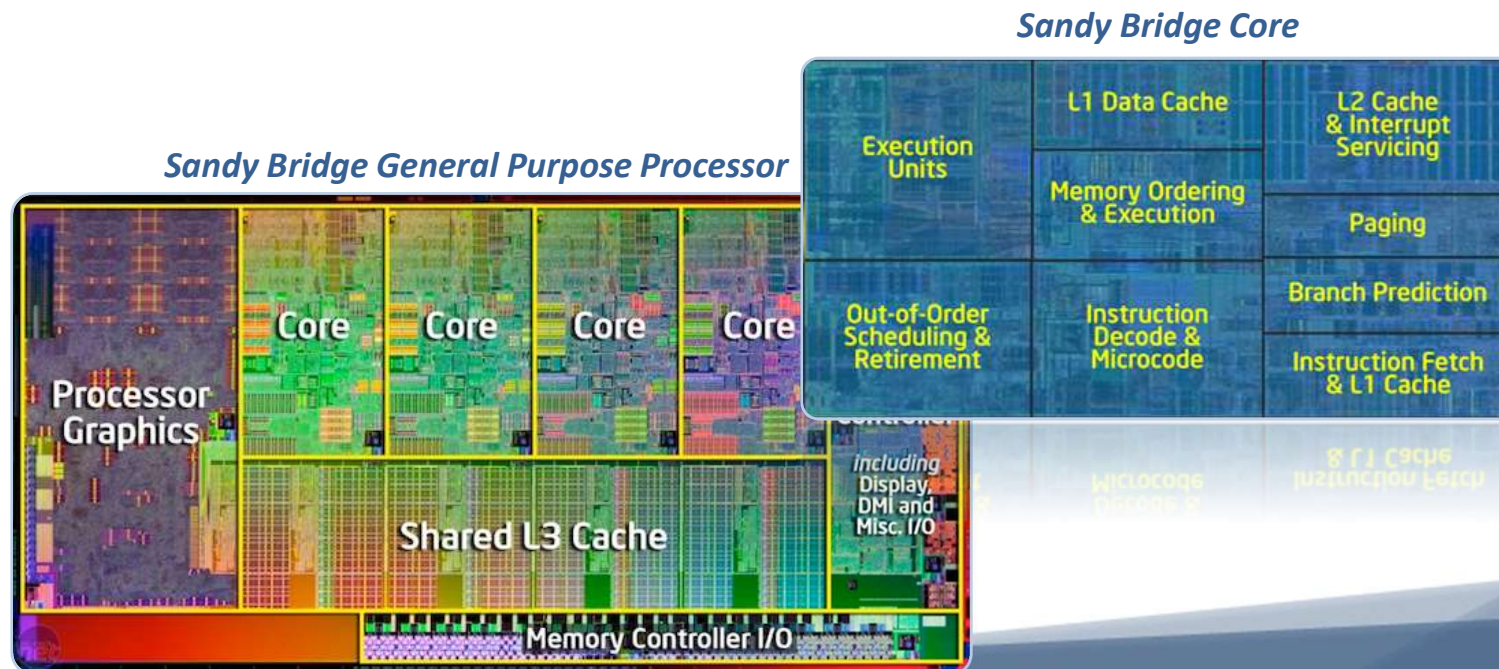
- **Mémoire Adressable par octet** : Beaucoup de familles mémoire sur ordinateur ou sur processeur embarqué (MCU, DSP, SoC) sont dîtes adressables par octet. Cela signifie qu'à chaque adresse mémoire correspond 1 octet. Prenons 2 exemples de mapping mémoire sur 16bits d'adresse et 32bits d'adresse :



Sur beaucoup de processeurs modernes (GPP, GPU, SoC, MCU, DSP ...), la mémoire est le plus souvent hiérarchisée et peut représenter par un modèle en couche :



Rappelons également que les technologies d'intégration des différents niveaux mémoire sont différentes. L'empreinte silicium pour 32Ko d'une mémoire cache L1 n'est pas du tout la même que pour 32Ko de cache L2. L'exemple ci-dessous illustre l'intégration de 32Ko de L1P/L1D, 256Ko de L2 et 6Mo de L3.



Observons à titre indicatif les latences d'accès aux différentes couches mémoire d'un core i7 de la famille Nehalem :

Memory Level	Capacity (bytes)	Line Size (bytes)	Access Latency (clocks)	Access Throughput (clocks)
L1 Data	32kb	64	4	1
L1 Instruction	32kb	-	-	-
L2 unified	256kb	64	10	varie
L3 shared (multi-core)	8Mb	64	35-40+	varie

*Rappelons qu'un cache est une copie d'une information présente dans un autre emplacement mémoire. Ceci peut entraîner des **problèmes de cohérences** des informations pouvant admettre une existence dans plusieurs emplacement physiques de l'architecture.*

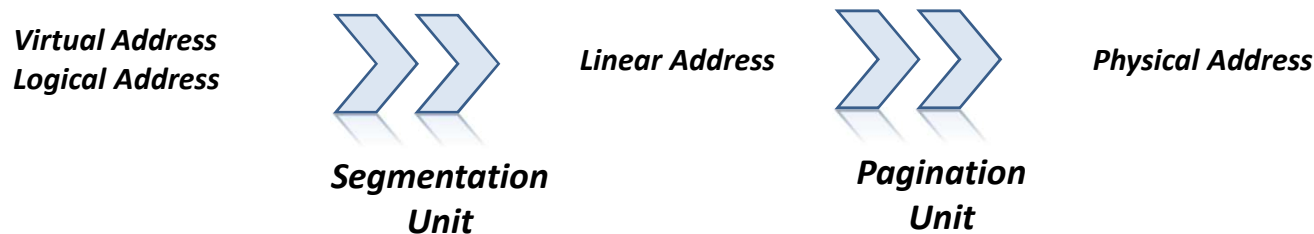
- Segmentation
- Pile et Tas
- Pagination
- exceptions et signaux

L'unité de gestion mémoire (MMU ou Memory Managment Unit) est intégrée dans la plupart des CPU's modernes et est exploitée par tout système d'exploitation évolué actuel (Windows, iOS, MAC OS, GNU/Linux...). Elle fut intégré sur processeur Intel sous ce nom depuis la famille 80286. Observons les principaux services offerts par cette unité :

- **Unité de Segmentation** (translation adresses logiques en adresses linéaires)
- **Unité de Pagination** (translation adresses linéaires en adresses physiques)
- **Unité de Protection** (MPU ou Memory Protection Unit).
Génération exceptions en cas d'accès mémoire illégaux.
- Arbitrage des bus...

- Segmentation
- Pile et Tas
- Pagination
- exceptions et signaux

- *Observons succinctement les mécanismes de translation d'une adresse virtuelle vers une adresse physique :*

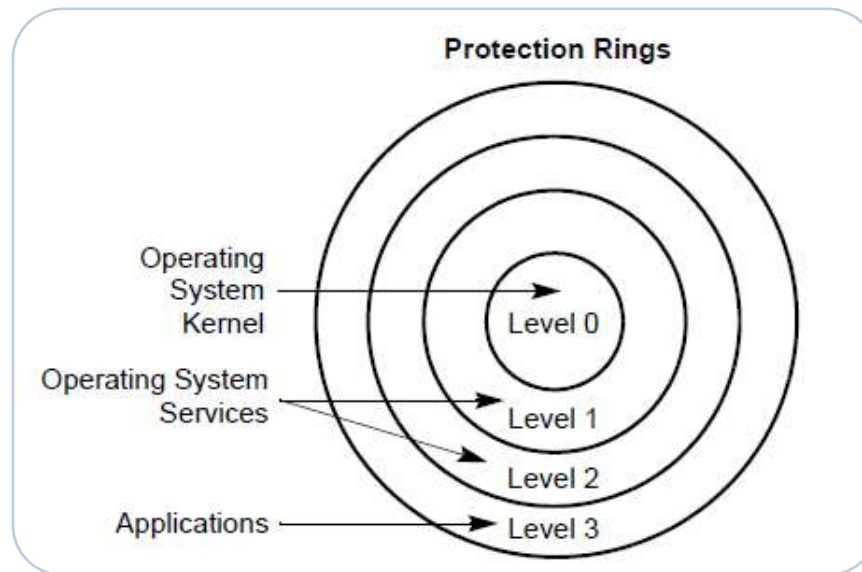


- *Nous verrons que dans les grandes lignes, pour un système comme Linux, ces mécanismes de translation peuvent être réduit à la pagination :*



- **Segmentation**
- Pile et Tas
- Pagination
- exceptions et signaux

Depuis la famille 80286 de Intel, un mode protégé avec privilèges a été introduit. La notion de privilège est utilisée par l'unité de segmentation en cas d'accès mémoire illégaux. Il s'agit d'un mécanisme de protection mémoire. Les CPU Intel proposent jusqu'à 4 niveaux de privilèges :



Vu dans la suite du cours !

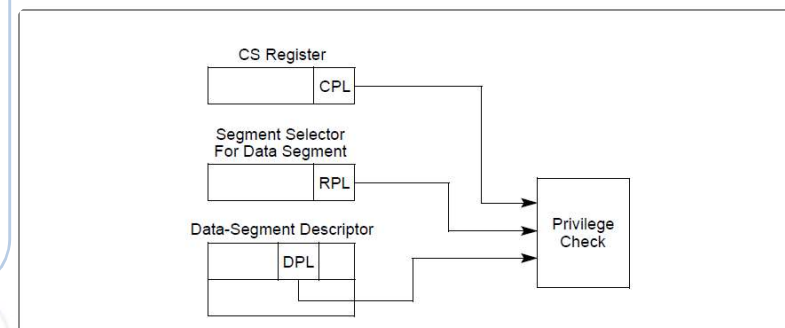
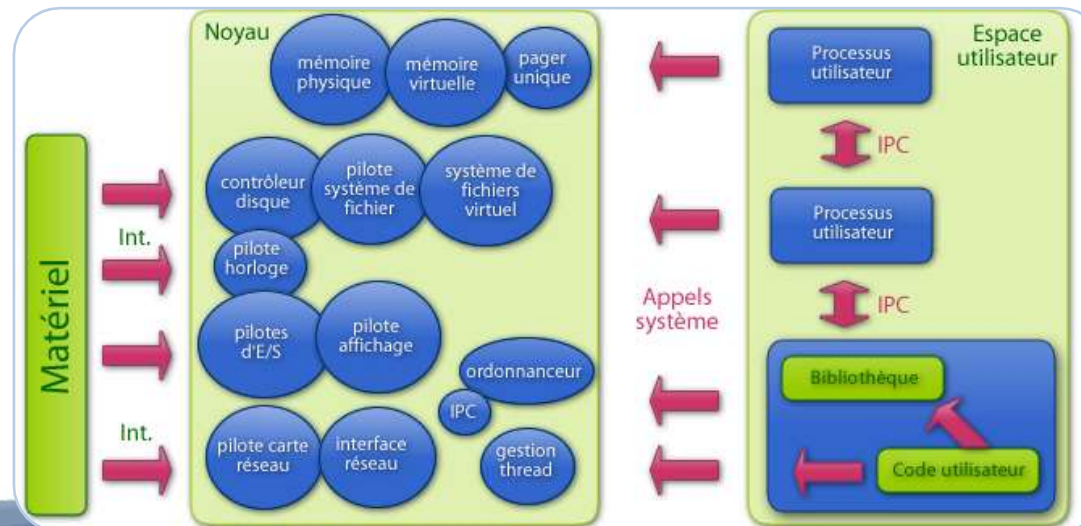


Figure 5-4. Privilege Check for Data Access

- **Segmentation**
- **Pile et Tas**
- **Pagination**
- **exceptions et signaux**

A titre indicatif, sur machine réelle linux n'utilise que 2 de ces 4 niveaux de privilèges. Ces deux niveaux sont alors appelés :

- **Mode kernel** : espace kernel ou noyau (privilège n°0 du CPU).
Privilège processeur n°1 ou supérieur sur machine virtuelle (XEN, KVM ...). La machine de virtualisation s'approprie le privilège n°0.
- **Mode User** : espace utilisateur (privilège n°3 du CPU)

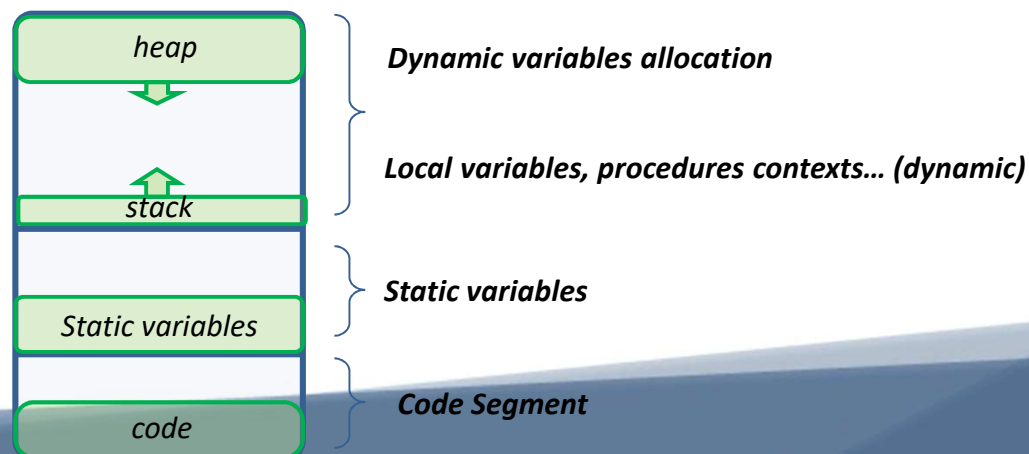


- **Segmentation**
- Pile et Tas
- Pagination
- exceptions et signaux

Intéressons-nous à l'unité de segmentation matérielle faisant partie intégrante de la MMU. Son but est de séparer en espaces logiquement indépendants le code, les variables statiques, les variables dynamiques ... Ce mécanisme de virtualisation mémoire assure historiquement une meilleure structuration des développements logiciels, facilite l'édition des liens et peut aider au partage de ressources dans le cadre d'application multithread.

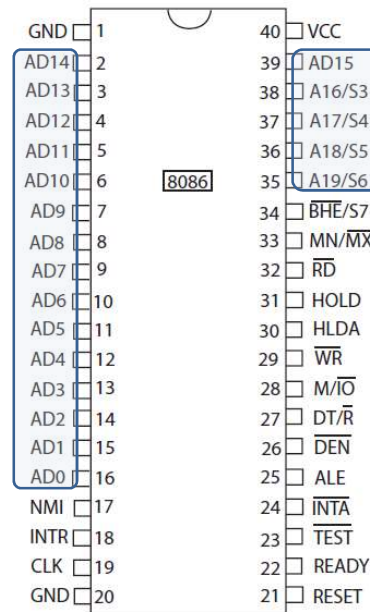
Exemple de segmentation mémoire :

Segmented Memory



- Segmentation
- Pile et Tas
- Pagination
- exceptions et signaux

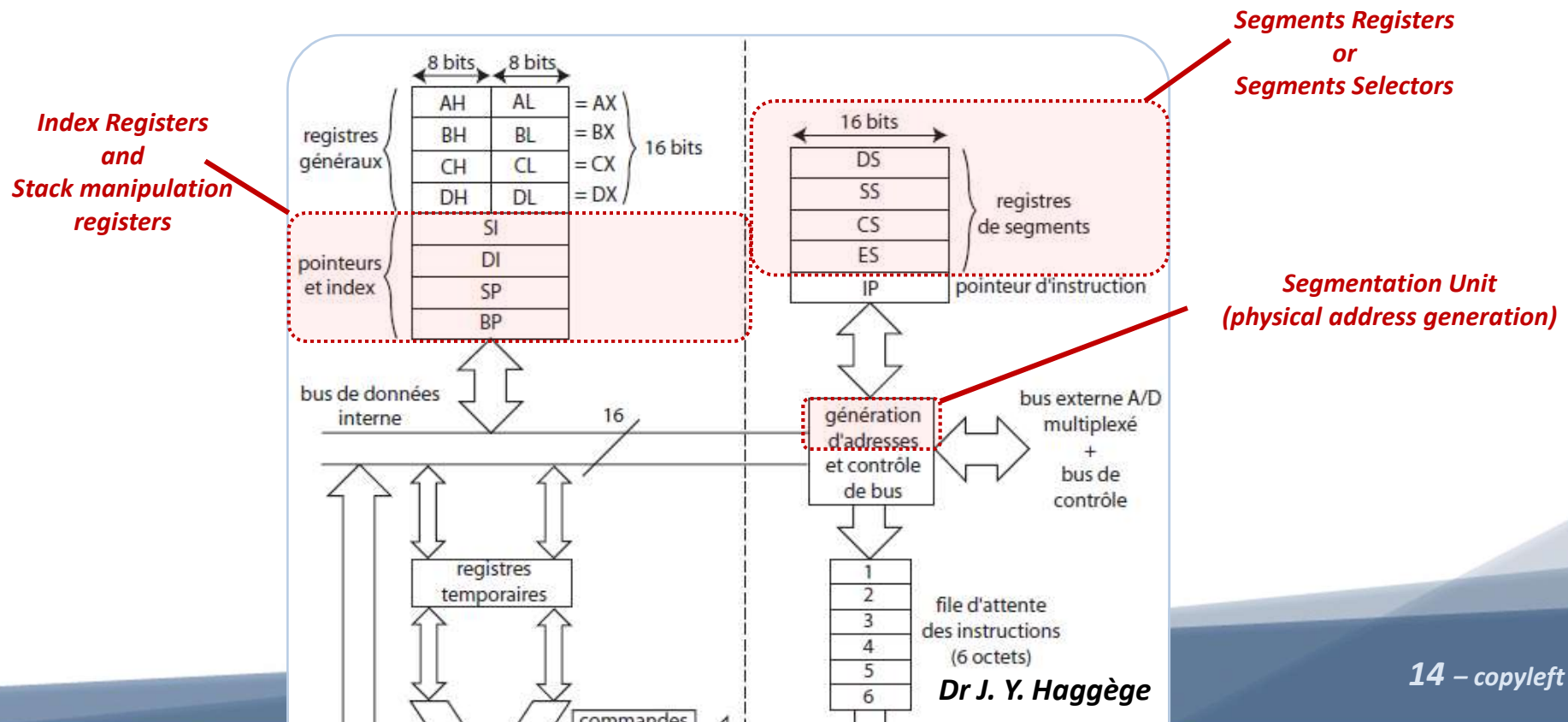
Etudions la segmentation mémoire implémentée sur CPU 8086 de Intel. Le 8086 possède un espace mémoire adressable de 1Mo (20bits d'adresse physique).



 : bus d'adresse 20 bits
(adresse physique)

- **Segmentation**
- Pile et Tas
- Pagination
- exceptions et signaux

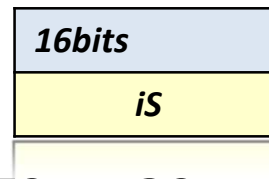
Le 8086 autorise la manipulation de 4 segments mémoire de 16bits d'adresse (DS=Data Segment, SS=Stack Segment, CS=Code Segment et ES=Extra Segment) :



- **Segmentation**
- Pile et Tas
- Pagination
- exceptions et signaux

Sur processeurs x86-64 modernes, les sélecteurs de segment font toujours 16bits, néanmoins de nouveaux sélecteurs ont été ajoutés (FS et GS).

Segment Selector (i = C, D, S, E, F and G)



Les segments DS, ES, FS et GS sont 4 segments de données. Ils permettent notamment d'assurer des accès plus efficaces et sécurisés en fonction des structures de données manipulées. Exemple d'utilisation de ces segments :

- *Contenir les données statiques du module courant*
- *Contenir les données dynamiques du module courant*
- *Partager des données avec un autre programme*
- ...

- **Segmentation**
- Pile et Tas
- Pagination
- exceptions et signaux

L'espace adressable est divisé en segments de 64Ko dont les 4 bits de poids faibles d'adresse sont multiples de 16. Une case mémoire est repérée par :

- **Adresse de base du segment** (16bits)
- **Offset** ou adresse effective dans le segment (16bits)

L'unité de segmentation est chargée de réaliser la translation d'une adresse logique (couple segment:offset) vers une adresse physique sur 20 bits (broches en sortie du CPU). Etudions la correspondance adresse logique adresse physique :

$$\text{Adresse physique} = \text{segment} \ll 4 + \text{offset}$$

- **Segmentation**
- Pile et Tas
- Pagination
- exceptions et signaux

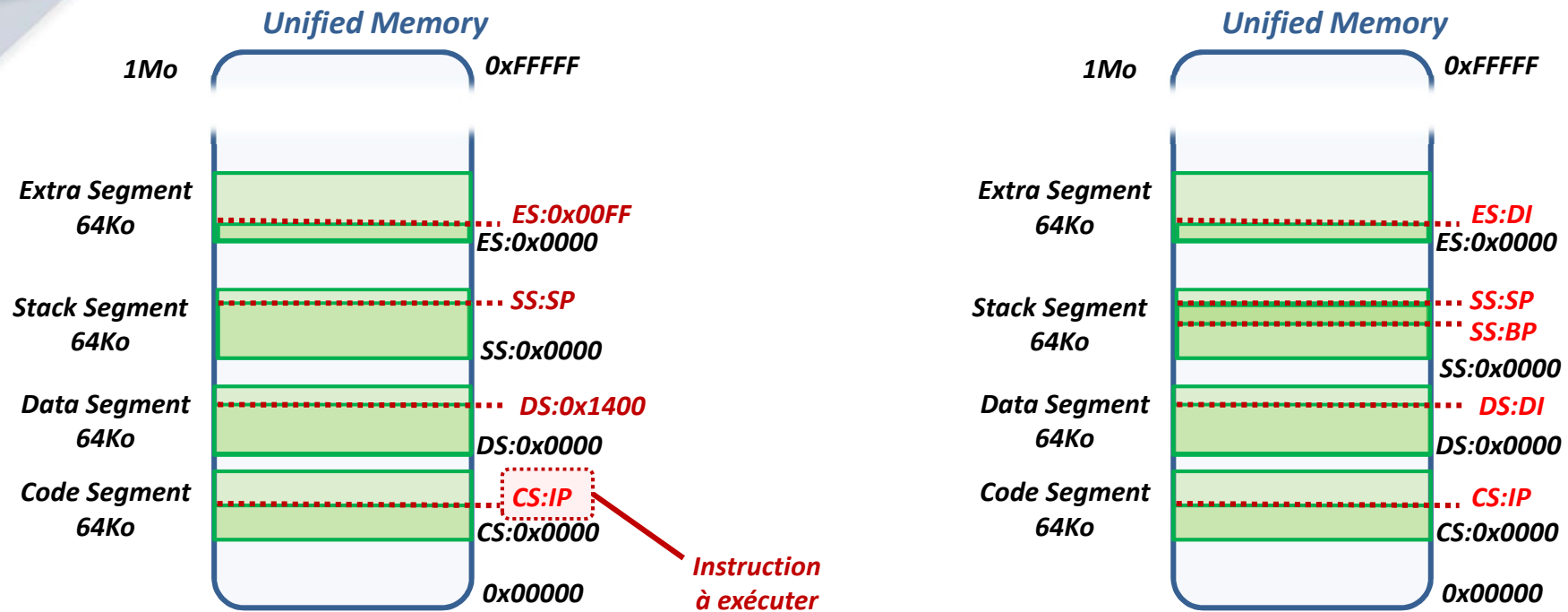
Etudions quelques couples segment:offset générant les adresses physiques :

Adresse physique = segment << 4 + offset

<i>DS (Data Segment)</i>	<i>Segment:offset</i>	<i>Physical Address</i>
<i>0xFB0A</i>	<i>DS:0x0005</i>	<i>0xFB0A5</i>
<i>0xF000</i>	<i>DS:0xB0A5</i>	<i>0xFB0A5</i>
<i>0xF203</i>	<i>DS:0x9075</i>	<i>0xFB0A5</i>
<i>...</i>	<i>...</i>	<i>...</i>

- Segmentation
- Pile et Tas
- Pagination
- exceptions et signaux

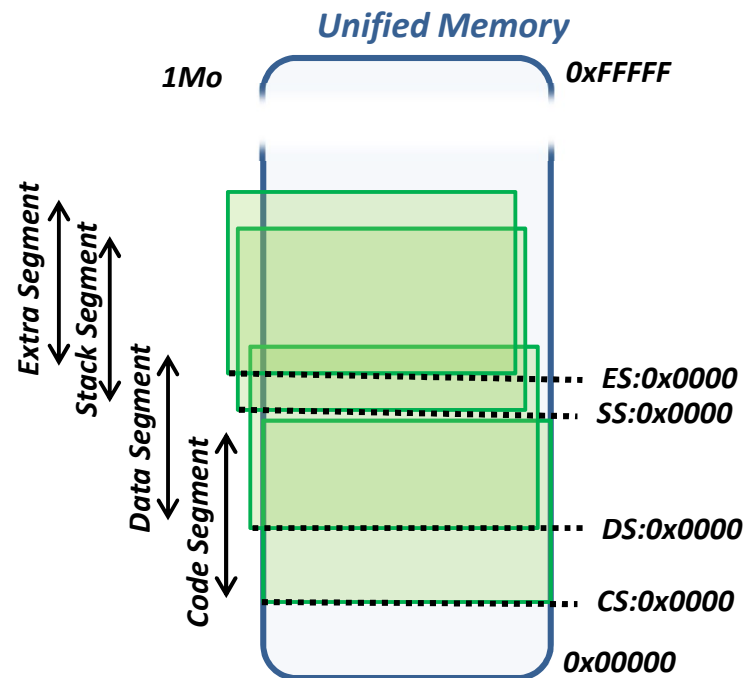
Observons un exemple de mapping mémoire :



Les registres de segments peuvent être associés à des registres de d'index (SI, DI, BP et SP). Observons les principaux couples : SS:BP, SS:SP, DS:SI, DS:DI, ES:SI et ES:DI

- Segmentation
- Pile et Tas
- Pagination
- exceptions et signaux

Les segments peuvent très bien se chevaucher voir même se recouvrir :

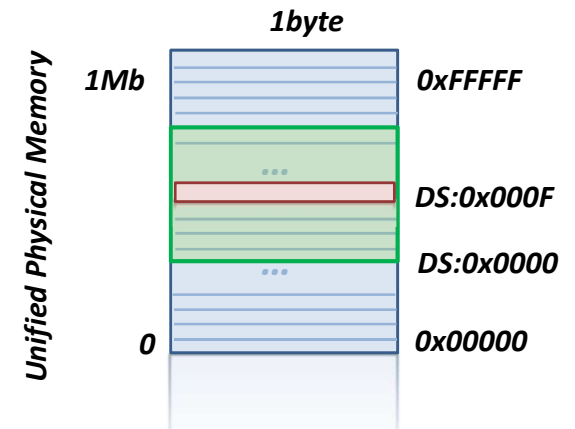


Pour information, observons les valeurs par défaut au reset du CPU : IP=0x0000, CS=0xFFFF, DS=ES=SS=0x0000. Le CPU boot donc à l'adresse CS:IP=0xFFFF0 (bootstrap).

- **Segmentation**
- Pile et Tas
- Pagination
- exceptions et signaux

- **Adressage direct** : déplacement de données du CPU vers la mémoire ou vice versa. L'adresse de la case mémoire à manipuler est directement passée avec l'opcode de l'instruction.

```
movb    (0x000F), %bl    ; utilise par défaut segment DS
movb    %ds:0x000F, %bl  ; forçage segment
```



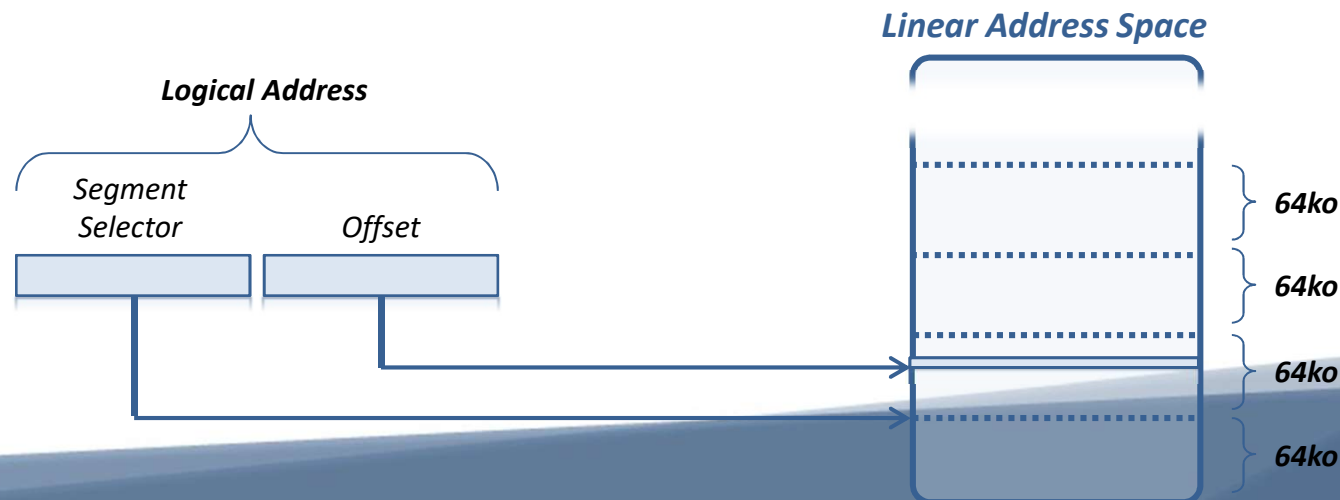
- **Adressage indirect** : déplacement de données du CPU vers la mémoire ou vice versa. L'adresse de la case mémoire à manipuler est passée indirectement par un registre.

```
movb    $0x000F, %bx
movb    %ds:(%bx), %bl    ; forçage segment
```


- **Segmentation**
- Pile et Tas
- Pagination
- exceptions et signaux

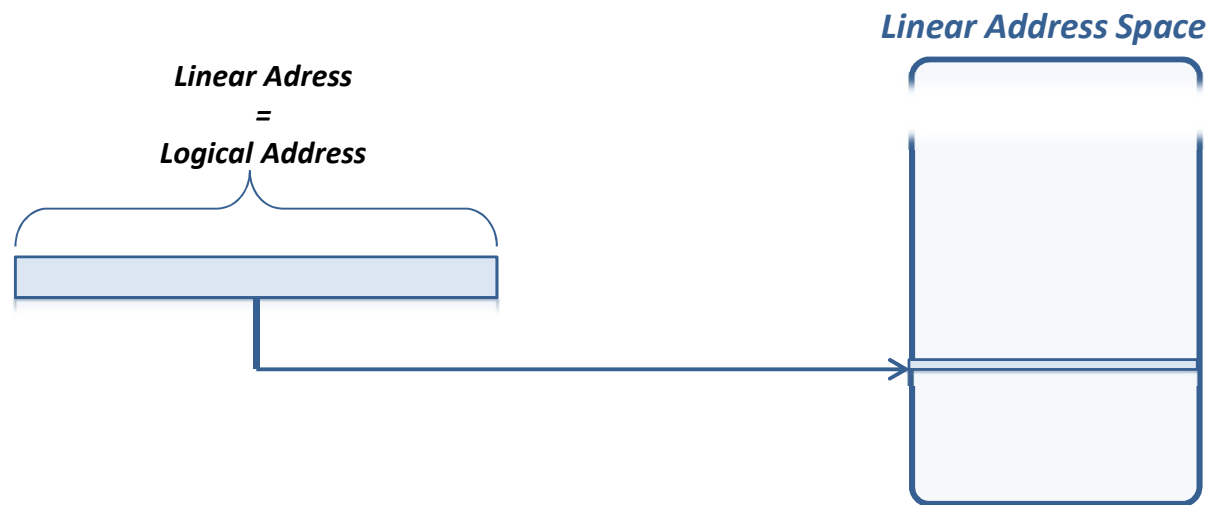
Les CPU's Intel modernes sont capables de gérer trois modèles mémoire distincts :

- **Real addressing mode** : modèle mémoire du 8086, préservé pour des soucis rétrocompatibilité. Segments de taille fixe (64ko), plage d'adresses linéaires de 1Mo (20bits). Pas de pagination mémoire (adresse linéaire = adresse physique). Les modèles mémoire segmentés sont plus complexe à gérer par les chaînes de compilation, maximise les erreurs de programme ...



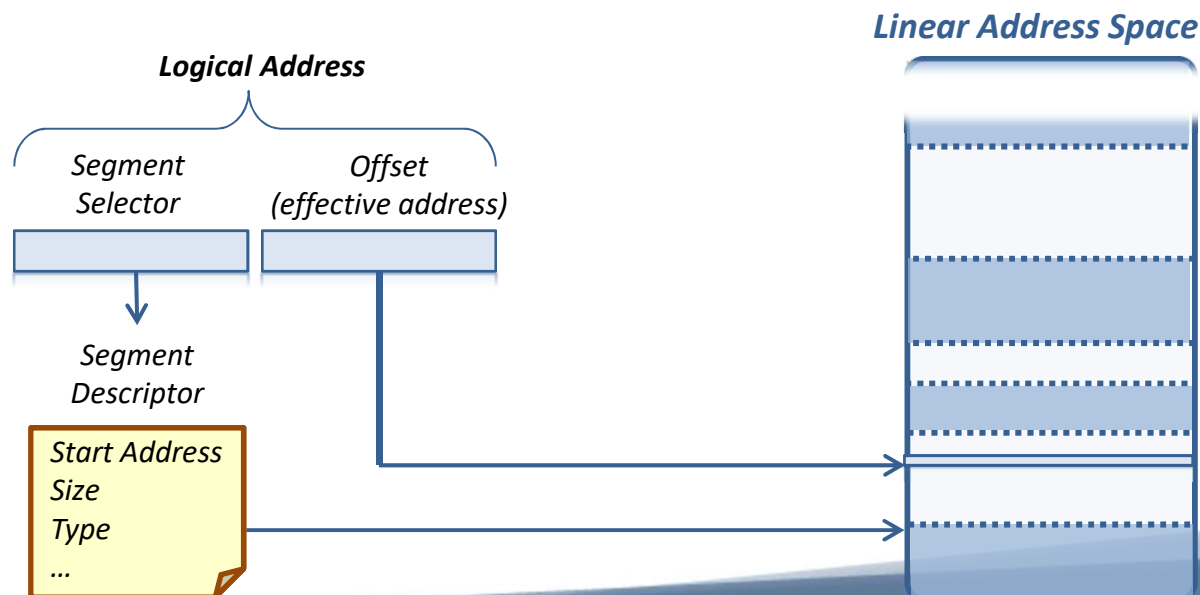
- **Segmentation**
- Pile et Tas
- Pagination
- exceptions et signaux

- **Flat addressing mode** : Espace mémoire adressable par octet contigu et non segmenté. Ce modèle inhibe la génération d'exceptions via l'unité de segmentation, néanmoins il offre une meilleure flexibilité et minimise l'utilisation de ressources hardware côté MMU.



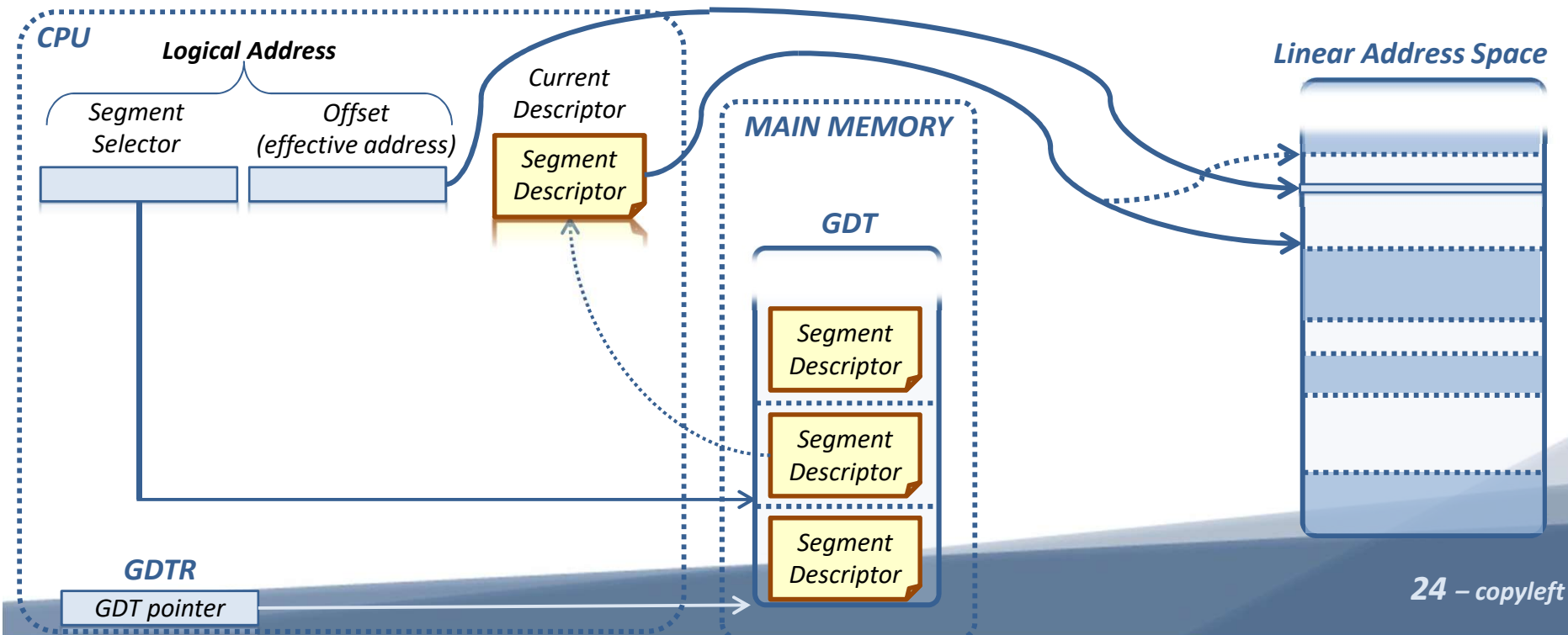
- Segmentation
- Pile et Tas
- Pagination
- exceptions et signaux

- **Segmented addressing mode** : modèle mémoire segmenté dont les segments peuvent être de taille variable. A chaque segment est associé un descripteur de segment (adresse de base du segment, taille, type code/data/stack...). Pagination possible. Par exemple, une architecture IA-32 peut supporter jusqu'à 16383 segments différents et chaque segment peut occuper jusqu'à 4Go.



- Segmentation
- Pile et Tas
- Pagination
- exceptions et signaux

Etudions plus en détail de mode segmenté arrivé avec le 80286 de Intel. Ce modèle mémoire utilise deux tables, la GDT (Global Descriptor Table) ou la LDT (Local Descriptor Table). La GDT est chargée de garder à portée de main les descripteurs de segments des processus les plus couramment utilisés :



- Segmentation
- Pile et Tas
- Pagination
- exceptions et signaux

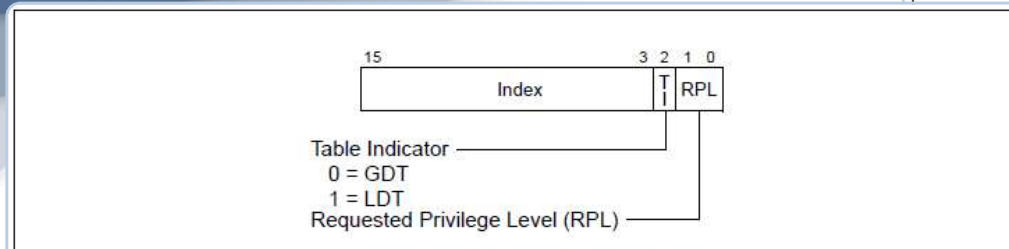


Figure 3-6. Segment Selector

Figure 3-6. Segment Selector

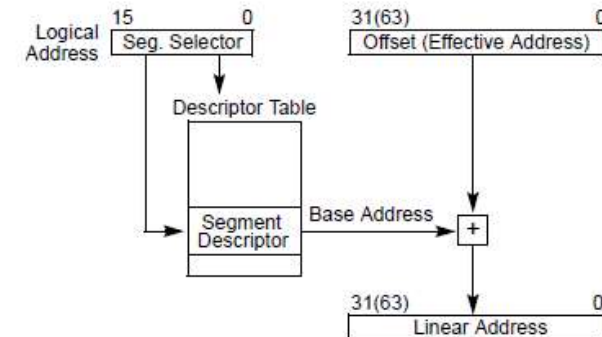


Figure 3-5. Logical Address to Linear Address Translation

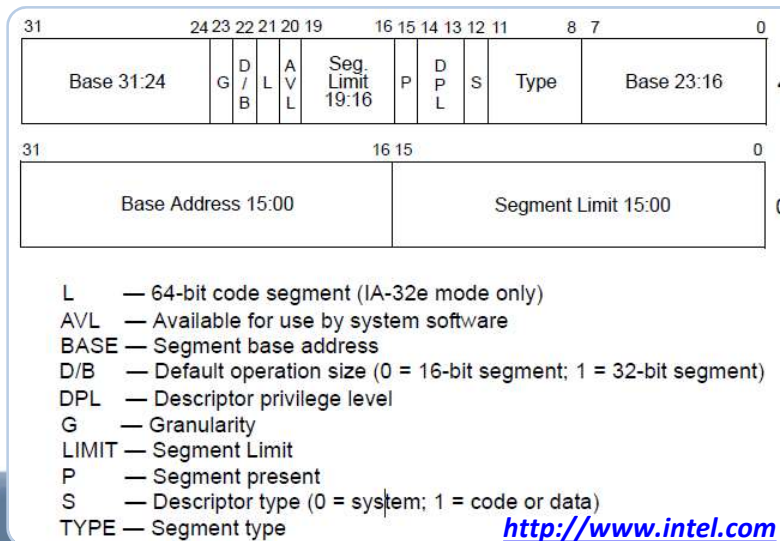
Figure 3-5. Logical Address to Linear Address Translation

Observons le contenu d'un sélecteur de segment (16bits). Index sur 13bits permettant de fixer la position relative du descripteur dans la table. Table Indicator permettant de savoir si le descripteur se situe dans la GDT ou la LDT, suivi de 2 bits de niveau de privilège associé au segment référencé (Data Segment). Les Code Segment et Stack Segment possède à la place de RPL le champ CPL (current Privilège Level) qui est associé au niveau de privilège de la tâche courante.

- **Segmentation**
- **Pile et Tas**
- **Pagination**
- **exceptions et signaux**

Observons le contenu d'un descripteur de segment (8 octets) générés ou mis à jour à la compilation (linker/loader) ou par l'OS :

- **BASE (32bits)** : adresse de base du segment sur un espace adressable de 4Go d'adresses linéaires
- **G (1bit)** : granularité du segment, soit 1o (si à 0) soit 4Ko (si à 1).
- **LIMIT (20bits)** : fixe la taille du segment en fonction de la granularité (entre 1Mo et 4Go).
- **P (1bit)** : segment présent en mémoire principale (si à 1) ou non.
- **S (1bit) et TYPE (4bits)** : fixe le type de segment system, code, data...
- **DPL (2bits)** : fixe le niveau de privilège associé au segment.
- **D/B, L ...**



- **Segmentation**
- Pile et Tas
- Pagination
- exceptions et signaux

Observons maintenant plus en détail la gestion de la protection mémoire par la MMU. Celle-ci observe et compare en temps réel pour chaque accès mémoire les champs CPL (Current Priority Level, associé à la tâche courante), RPL (Request Priority Level, associé au programme courant, peut-être modifié par l'instruction APRL pour du partage de données) et DPL (Descriptor Priority Level, associé au segment ciblé). En cas de violation de privilège, une exception matérielle sera relevée et préemptée par l'OS.

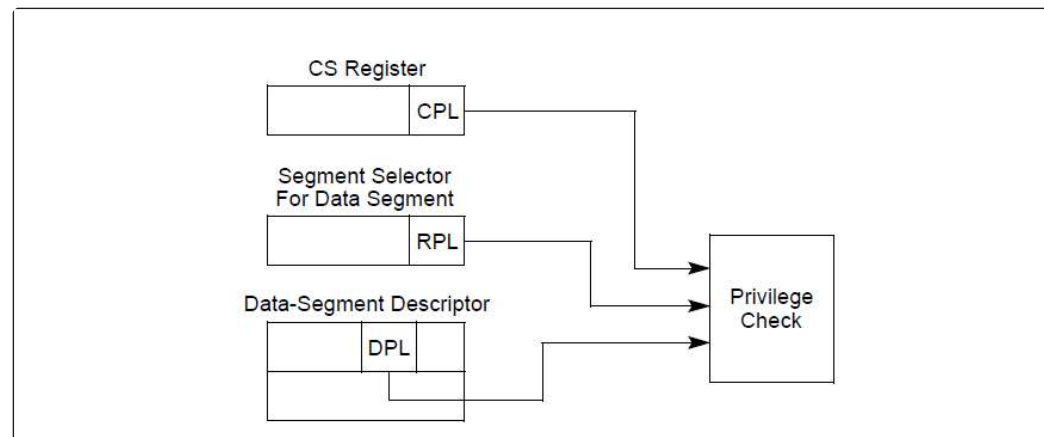
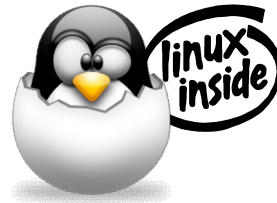


Figure 5-4. Privilege Check for Data Access

- **Segmentation**
- Pile et Tas
- Pagination
- exceptions et signaux

Il faut savoir que Linux utilise très peu la segmentation mémoire contrairement à la pagination. Voici les principales raisons :



- ***Le travail de la MMU est grandement simplifié si les processus partagent les mêmes segments mémoire (même espace d'adressage linéaire)***
- ***Linux a vocation à être multiplateformes, or de nombreuses architectures RISC n'ont qu'un usage très limité de la segmentation. A titre indicatif, le kernel Linux 2.6 n'utilise la segmentation mémoire que sur architecture x86.***

- **Segmentation**
- Pile et Tas
- Pagination
- exceptions et signaux

Par exemple sous Linux, tous les processus évoluant dans l'espace user partagent les mêmes segments de code et de data (Flat memory model). Idem pour les processus noyau évoluant en mode kernel. Observons les 4 principaux descripteurs de segment de linux.

Segment	Base	G	Limit	S	Type Code/data	P in memory	DPL privilège	D/B 32bits code/data
		4Gb size						
User code	0x00000000	1	0xFFFFF	1	10	1	3	1
User data	0x00000000	1	0xFFFFF	1	2	1	3	1
Kernel code	0x00000000	1	0xFFFFF	1	10	1	0	1
Kernel data	0x00000000	1	0xFFFFF	1	2	1	0	1



- **Segmentation**
- Pile et Tas
- Pagination
- exceptions et signaux

Observons également une partie du contenu des GDT's de Linux ainsi que les valeurs des sélecteurs associés (une GDT par cœur). Chaque GDT Linux possède 18 descripteurs utiles et 14 descripteurs non utilisés. Observons également sous gdb le contenu des sélecteurs de segment pour une application user :

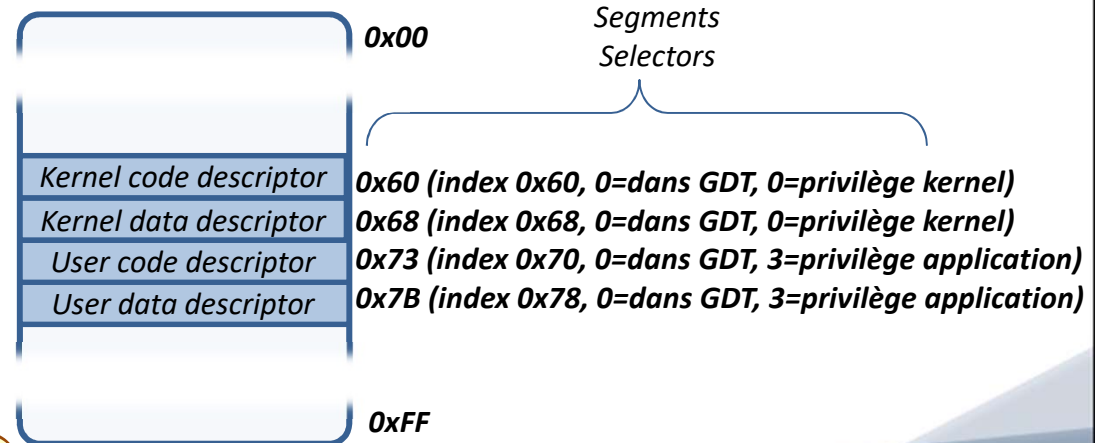
```

root@vmlinux: /home/vmlinux/x86-64/stack
(gdb) info all-registers
eax            0x1          1
ecx            0xbffff6b4    -1073744204
edx            0xbffff644    -1073744316
ebx            0xb7fc5ff4    -1208197132
esp            0xbffff608    0xbffff608
ebp            0xbffff618    0xbffff618
esi            0x0           0
edi            0x0           0
eip            0x80483bd      0x80483bd <main+9>
eflags         0x200282 [ SF IF ID ]
cs             0x73          115
ss             0x7b          123
ds             0x7b          123
es             0x7b          123
fs             0x0           0
gs             0x33          51
02            0x33          51
12            0x0           0
62            0x10          16
14            0x10          16
16            0x10          16

```

Toujours les mêmes !

Linux's GDT



- Segmentation
- **Pile et Tas**
- Pagination
- exceptions et signaux

Nous allons maintenant nous attarder sur des segments historiques très important, ceux relatifs à la gestion dynamique de données. Le contexte d'appel et d'exécution d'une procédure (allouées en entrant dans la procédure) et les autres variables alloués dynamiquement durant l'exécution d'un programme.

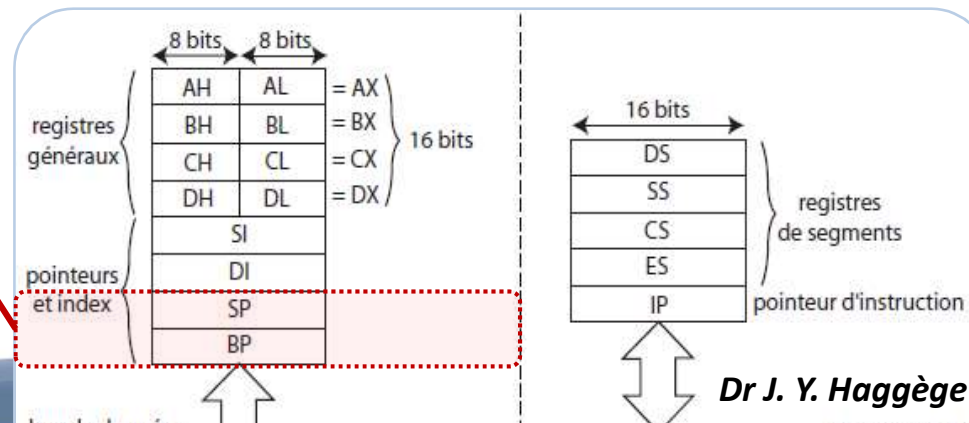
- **Stack ou Pile** : zone mémoire permettant l'allocation dynamique de ressources mémoire associées au contexte d'appel et d'exécution de procédures (paramètres et valeur de retour de fonction, variables locales, adresse de retour et contexte d'exécution de la procédure appelante).
- **Heap ou Tas** : zone mémoire permettant l'allocation dynamique de ressources mémoire durant l'exécution d'un programme.

- **Segmentation**
- *Pile et Tas*
- *Pagination*
- *exceptions et signaux*

Retrouvons le 8086 de Intel et observons les deux registres contenant les pointeurs associés aux mécanismes de gestion des piles :

- **Stack Pointer (SP)** : pointe le sommet courant de la pile
- **Base Pointer (BP) ou Frame Pointer** : pointe la base courante de la pile qui est associée au contexte d'exécution de la procédure en cours de traitement.

**Stack
manipulation
registers**

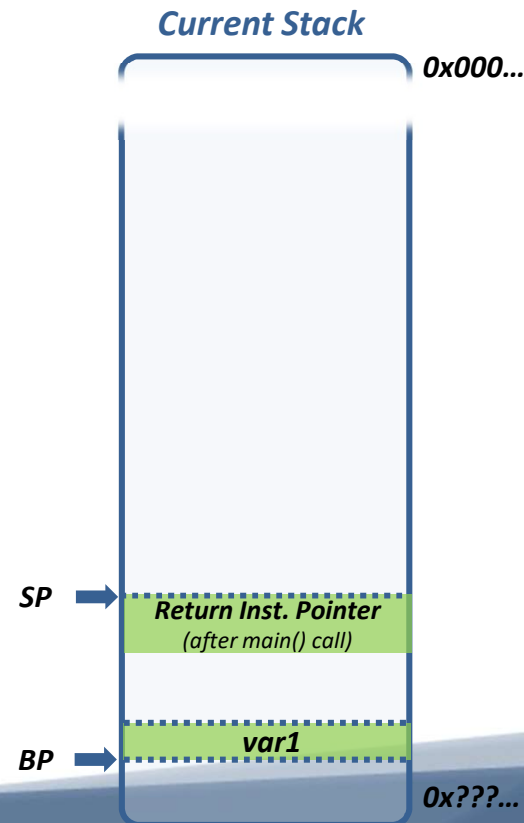


Dr J. Y. Haggège

- Segmentation
- **Pile et Tas**
- Pagination
- exceptions et signaux

Une pile est une famille de file d'attente gérée comme une "LIFO queue" (Last In First Out). Observons un programme en langage C et étudions un exemple de gestion de pile :

```
void fctStack1( float var3) {  
    short var4;  
    // user Code  
    var4 = fctStack2( 1 );  
}  
  
short fctStack2( char var5) {  
    short var6=2;  
    // user Code  
    return var6;  
}  
  
int main (void) {  
    char var1;  
    float var2=2.0;  
  
    fctStack1(var2 );  
    return 0;  
}
```



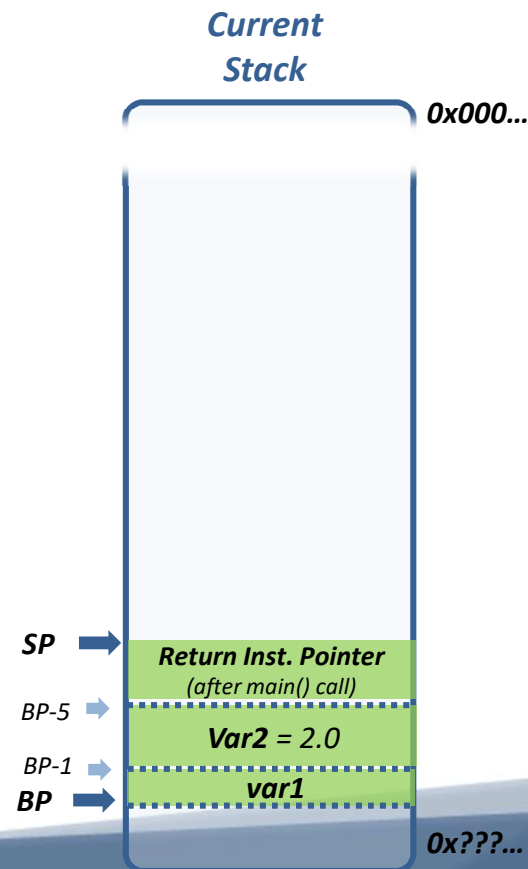
- Segmentation
- **Pile et Tas**
- Pagination
- exceptions et signaux

Les variables locales sont manipulées via des adresses relatives à BP (Base Pointer ou Frame Pointer). Par exemple var1 se situe à l'adresse BP, var2 à l'adresse BP+1 ... :

```
void fctStack1( float var3) {  
    short var4;  
    // user Code  
    var4 = fctStack2( 1 );  
}
```

```
short fctStack2( char var5) {  
    short var6=2;  
    // user Code  
    return var6;  
}
```

```
int main (void) {  
    char var1;  
    float var2=2.0;  
  
    fctStack1(var2 );  
    return 0;  
}
```



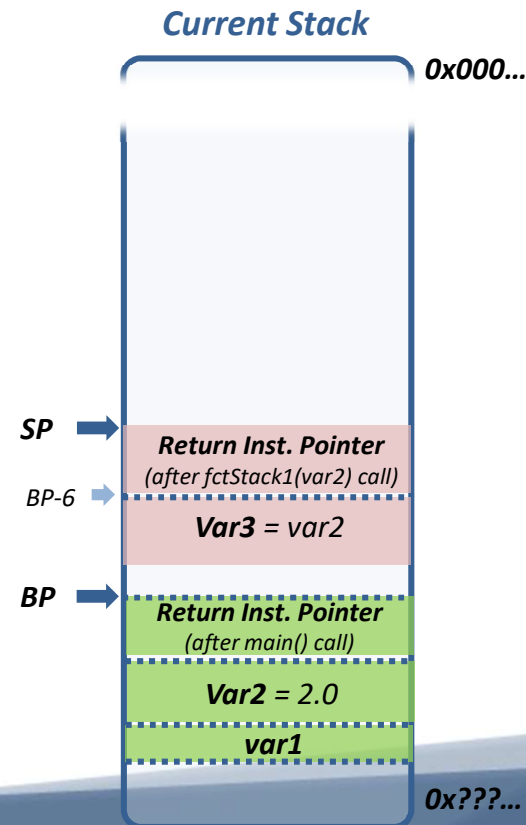
- Segmentation
- **Pile et Tas**
- Pagination
- exceptions et signaux

Les pointeurs BP (Base Pointer) et SP (Stack Pointer) sont associés au contexte de la procédure courante. Deux pointeurs suffisent pour manipuler la totalité des procédures du Thread courant :

```
void fctStack1( float var3) {  
    short var4;  
    // user Code  
    var4 = fctStack2( 1 );  
}
```

```
short fctStack2( char var5) {  
    short var6=2;  
    // user Code  
    return var6;  
}
```

```
int main (void) {  
    char var1;  
    float var2=2.0;  
  
    fctStack1(var2 );  
    return 0;  
}
```



- Segmentation
- **Pile et Tas**
- Pagination
- exceptions et signaux

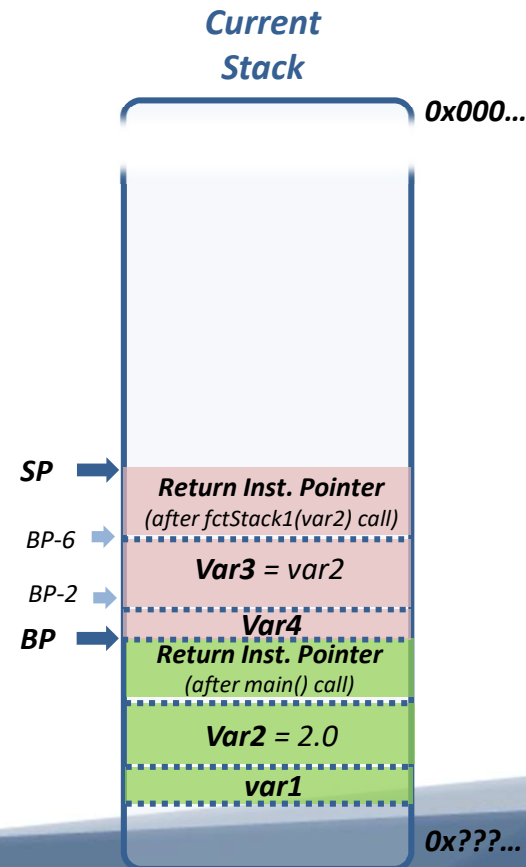
A chaque appel de procédure, l'adresse de l'instruction de retour est également empilée sur la pile :

```
void fctStack1( float var3) {
    short var4;
    // user Code
    var4 = fctStack2( 1 );
}
```

```
short fctStack2( char var5) {
    short var6=2;
    // user Code
    return var6;
}
```

```
int main (void) {
    char var1;
    float var2=2.0;

    fctStack1(var2 );
    return 0;
}
```



- Segmentation
- **Pile et Tas**
- Pagination
- exceptions et signaux

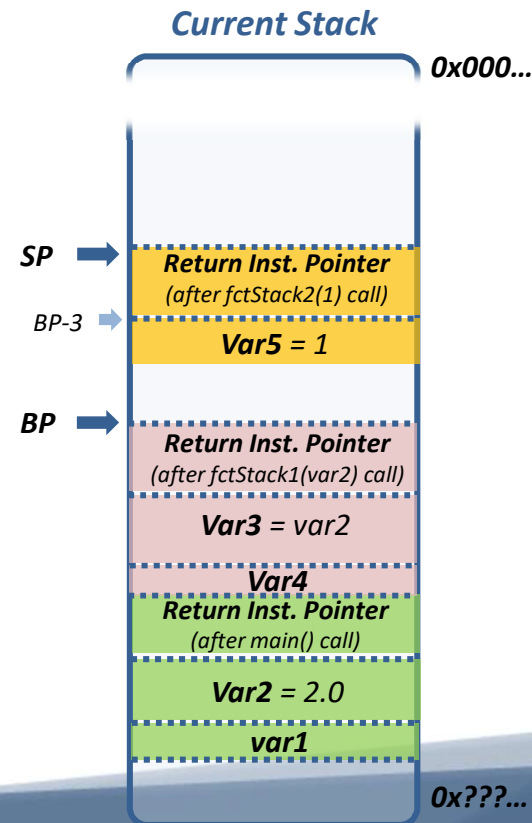
La pile possède une taille fixe (pouvant être modifiée). Il faut donc être prudent à d'éventuels débordement de pile (stack overflow). Prenons l'exemple des fonctions récursives qui sont à manipuler avec précaution :

```
void fctStack1( float var3) {
    short var4;
    // user Code
    var4 = fctStack2( 1 );
}
```

```
short fctStack2( char var5) {
    short var6=2;
    // user Code
    return var6;
}
```

```
int main (void) {
    char var1;
    float var2=2.0;

    fctStack1(var2 );
    return 0;
}
```



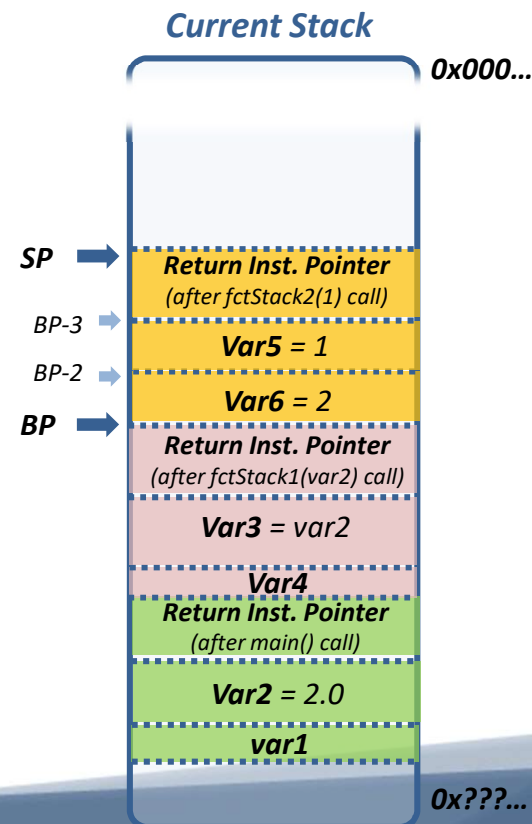
- Segmentation
- **Pile et Tas**
- Pagination
- exceptions et signaux

Le jeu d'instruction x86 définit des instructions dédiées à l'empilement et au dépilement de données sur la pile (**push** et **pop**). L'instruction **ret** est chargée de dépiler l'adresse de retour et de mettre à jour IP (Instruction Pointer) :

```
void fctStack1( float var3) {  
    short var4;  
    // user Code  
    var4 = fctStack2( 1 );  
}
```

```
short fctStack2( char var5) {  
    short var6=2;  
    // user Code  
    return var6;  
}
```

```
int main (void) {  
    char var1;  
    float var2=2.0;  
  
    fctStack1(var2 );  
    return 0;  
}
```



- Segmentation
- **Pile et Tas**
- Pagination
- exceptions et signaux

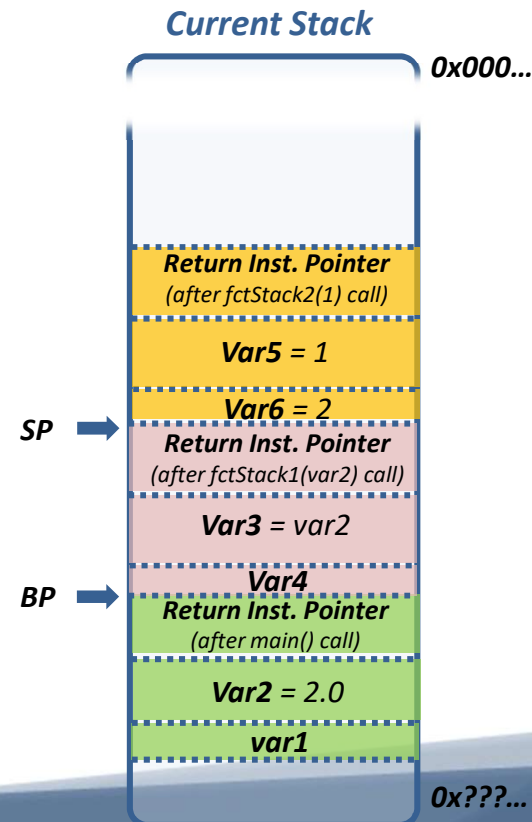
Le compilateur retournera le plus souvent la valeur de retour d'une fonction par registre (plus rapide). Il fera également de même pour les fonctions n'ayant qu'un paramètre (passage par registre) :

```
void fctStack1( float var3) {
    short var4;
    // user Code
    var4 = fctStack2( 1 );
}
```

```
short fctStack2( char var5) {
    short var6=2;
    // user Code
    return var6;
}
```

```
int main (void) {
    char var1;
    float var2=2.0;

    fctStack1(var2 );
    return 0;
}
```



- Segmentation
- **Pile et Tas**
- Pagination
- exceptions et signaux

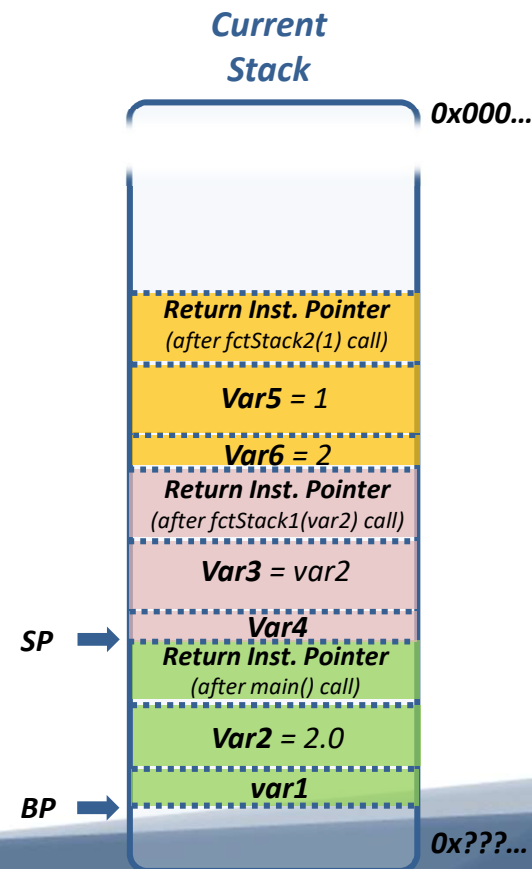
Nous pouvons constater que les variables précédemment allouées dans la pile ne sont pas “effacées”, seul les contextes d’exécution ont été perdus :

```
void fctStack1( float var3) {
    short var4;
    // user Code
    var4 = fctStack2( 1 );
}
```

```
short fctStack2( char var5) {
    short var6=2;
    // user Code
    return var6;
}
```

```
int main (void) {
    char var1;
    float var2=2.0;

    fctStack1(var2 );
    return 0;
}
```



- Segmentation
- **Pile et Tas**
- Pagination
- exceptions et signaux

Observons un exemple de code C ainsi que l'assembleur équivalent compilé sous gcc (syntaxe AT&T) sur architecture Intel 64. Attention, sur architecture Intel, le stack pointer doit être aligné modulo 20 ou 40 en fonction du flag D présent dans le descripteur de segment :

```
int main(void){
    // pushq    %rbp          ; empile BP contexte fonction appelante (pour future restauration)
    //                               ; pointeurs sur 64bits (architecture Intel 64)
    // movq     %rsp, %rbp    ; mise à jour BP procédure courante

    float lclFloat=8.0;
    // movl     $0x41000000, %eax    ; affectation 8.0 à eax (registre 32bits)
    //                               ; Norme IEEE754 Single Precision → Value=(-1)e(S) x 1.M x 2e(E-127)
    //                               ; 32bits → S EEEEEEEEE MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
    //                               ; 0x41000000 → S=+ E=130 M=1.0 → value = +1.0x2e3 = 8.0
    // movl     %eax, -4(%rbp)    ; sauvegarde 8.0 dans la pile à l'adresse relative bp-4

    // code utilisateur ...

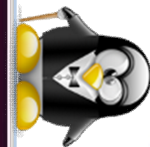
    return 0;
    // movl     $0, %eax         ; valeur de retour passée par eax (registre 32bits)
    // popq     %rbp             ; dépile BP de la procédure appelante et MAJ de bp (restauration ancien contexte)
    // ret                               ; dépile l'adresse de retour et mise à jour du registre d'instruction (IP)
}
```

- Segmentation
- **Pile et Tas**
- Pagination
- exceptions et signaux

*Il est possible de lire et modifier la taille de la pile du processus courant via les fonctions **getrlimit** et **setrlimit**. Appelons depuis le shell la commande **ulimit** (obsolète) et observons puis modifions la taille de la pile associée au shell courant. Linux utilise par défaut des piles de **8Mo** et de taille minimale **128Ko** depuis la version 2.6.34 du kernel (modulo la taille d'une page sur de nombreux systèmes) :*

```
vmlinux@vmlinux: ~  
vmlinux@vmlinux:~$ ulimit -a  
core file size          (blocks, -c) 0  
data seg size           (kbytes, -d) unlimited  
scheduling priority     (-e) 0  
file size               (blocks, -f) unlimited  
pending signals         (-i) 15990  
max locked memory       (kbytes, -l) 64  
max memory size         (kbytes, -m) unlimited  
open files              (-n) 1024  
pipe size               (512 bytes, -p) 8  
POSIX message queues    (bytes, -q) 819200  
real-time priority      (-r) 0  
stack size              (kbytes, -s) 8192  
cpu time                (seconds, -t) unlimited  
max user processes      (-u) 15990  
virtual memory          (kbytes, -v) unlimited  
file locks              (-x) unlimited  
vmlinux@vmlinux:~$ ulimit -s  
8192  
vmlinux@vmlinux:~$ ulimit -s 16384  
vmlinux@vmlinux:~$ ulimit -s  
16384  
vmlinux@vmlinux:~$
```

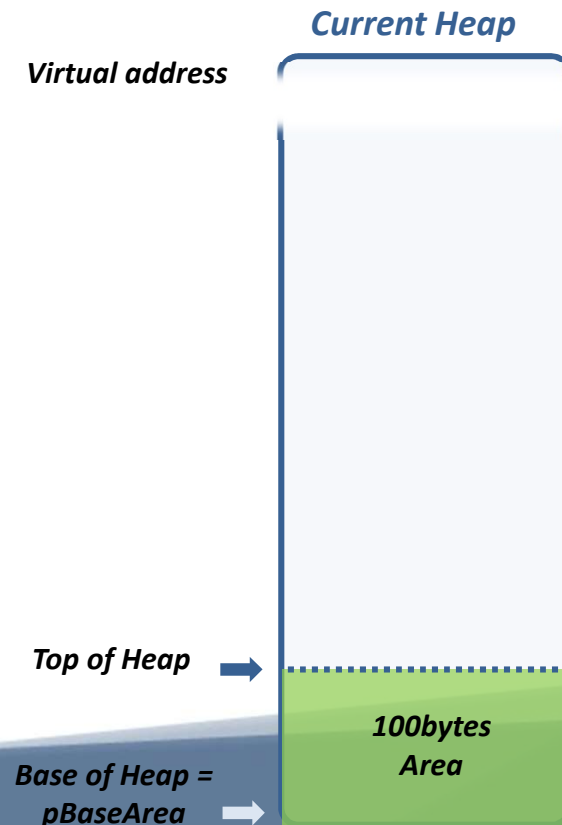
```
vmlinux@vmlinux: ~  
vmlinux@vmlinux:~$ ulimit -s  
8192  
vmlinux@vmlinux:~$
```



- Segmentation
- **Pile et Tas**
- Pagination
- exceptions et signaux

Intéressons nous maintenant aux mécanismes d'allocation dynamique de données sur le tas ou heap. Le Tas est propre à chaque processus et est une zone mémoire dont les ressources sont librement allouables en cours d'exécution du programme :

```
int main (void) {  
    char* pBaseArea;  
  
    // allocate 100bytes in heap  
    pBaseArea = (char*) malloc (100*sizeof(char));  
    if (pBaseArea == NULL )  
        printf ("\nheap allocation failure\n");  
  
    // user application ...  
  
    // free last 100bytes allocated  
    free (pBaseArea );  
  
    return 0;  
}
```



- Segmentation
- **Pile et Tas**
- Pagination
- exceptions et signaux

Etudions un programme C très simple réalisant des allocations dynamiques et observons les mécanismes de gestion du tas :

```
int main (void) {
    float* pBaseArea1;
    char* pBaseArea2;
    double* pBaseArea3;

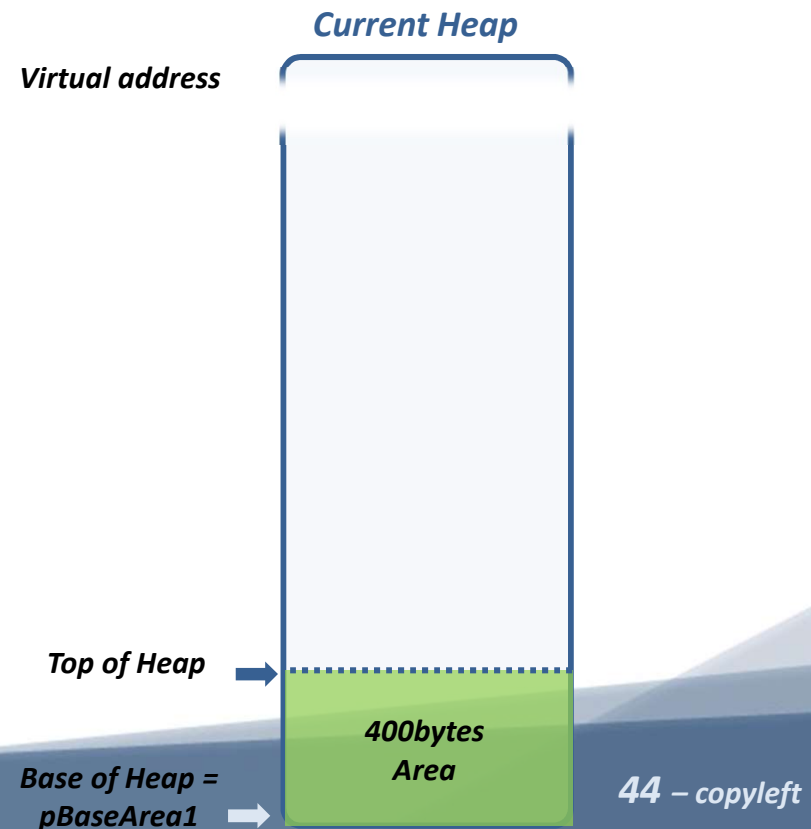
    pBaseArea1 = (float*) malloc (100*sizeof(float));
    pBaseArea2 = (char*) malloc (100*sizeof(char));
    pBaseArea3 = (double*) malloc (100*sizeof(double));

    // user application ...

    free ( pBaseArea2 );
    pBaseArea2 = (char*) malloc (200*sizeof(char));

    // user application ...

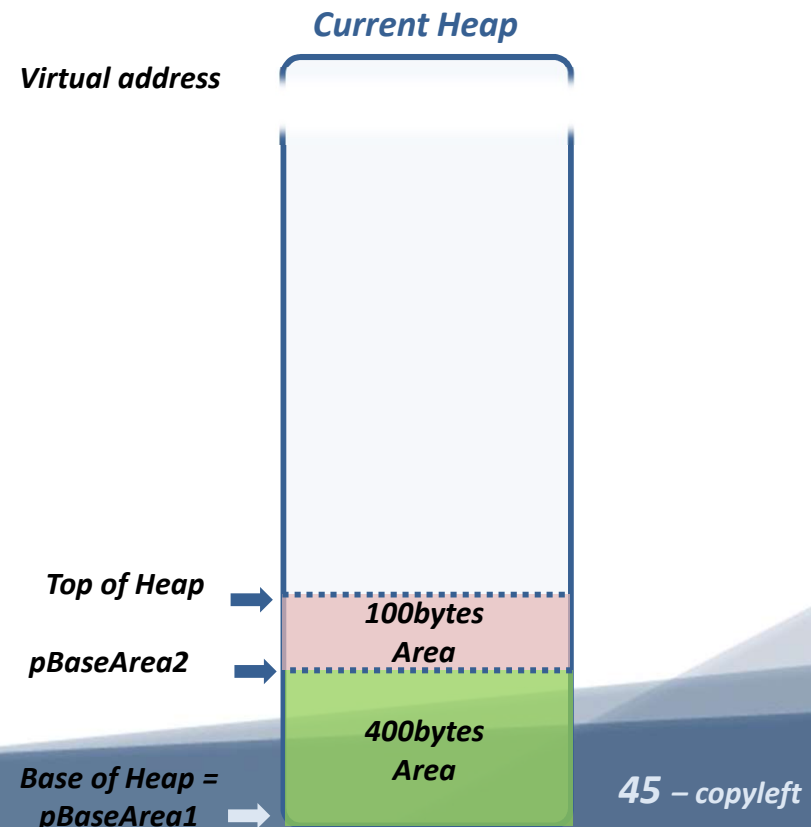
    free ( pBaseArea1 );
    free ( pBaseArea2 );
    free ( pBaseArea3 );
    return 0;
}
```



- Segmentation
- **Pile et Tas**
- Pagination
- exceptions et signaux

Comme pour la gestion de la pile, des débordements de tas sont possibles (heap overflow). Nous verrons par la suite que nous avons néanmoins accès à de très larges ressources mémoire sous Linux.

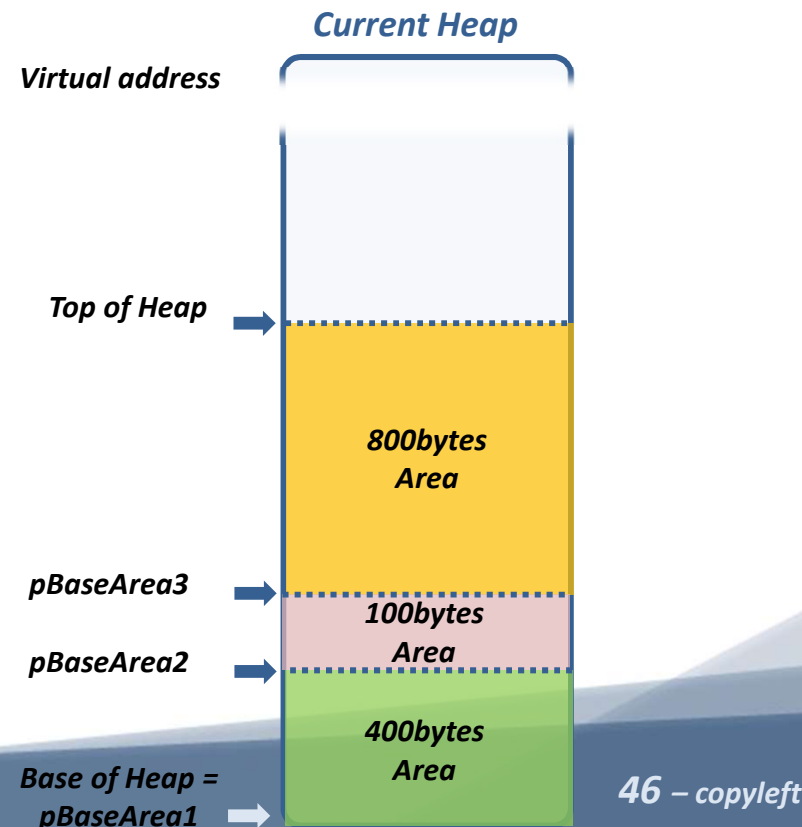
```
int main (void) {  
    float* pBaseArea1;  
    char* pBaseArea2;  
    double* pBaseArea3;  
  
    pBaseArea1 = (float*) malloc (100*sizeof(float));  
    pBaseArea2 = (char*) malloc (100*sizeof(char));  
    pBaseArea3 = (double*) malloc (100*sizeof(double));  
  
    // user application ...  
  
    free ( pBaseArea2 );  
    pBaseArea2 = (char*) malloc (200*sizeof(char));  
  
    // user application ...  
  
    free ( pBaseArea1 );  
    free ( pBaseArea2 );  
    free ( pBaseArea3 );  
    return 0;  
}
```



- Segmentation
- **Pile et Tas**
- Pagination
- exceptions et signaux

Ne surtout pas oublier de libérer les différentes ressources mémoires allouées dans le tas après utilisation. Toute ressource mémoire allouée (surtout large) durant l'exécution de votre application ne pourra pas être manipulée par d'autres processus.

```
int main (void) {  
    float* pBaseArea1;  
    char* pBaseArea2;  
    double* pBaseArea3;  
  
    pBaseArea1 = (float*) malloc (100*sizeof(float));  
    pBaseArea2 = (char*) malloc (100*sizeof(char));  
    pBaseArea3 = (double*) malloc (100*sizeof(double));  
  
    // user application ...  
  
    free ( pBaseArea2 );  
    pBaseArea2 = (char*) malloc (200*sizeof(char));  
  
    // user application ...  
  
    free ( pBaseArea1 );  
    free ( pBaseArea2 );  
    free ( pBaseArea3 );  
    return 0;  
}
```



- Segmentation
- **Pile et Tas**
- Pagination
- exceptions et signaux

```
int main (void) {
    float* pBaseArea1;
    char* pBaseArea2;
    double* pBaseArea3;

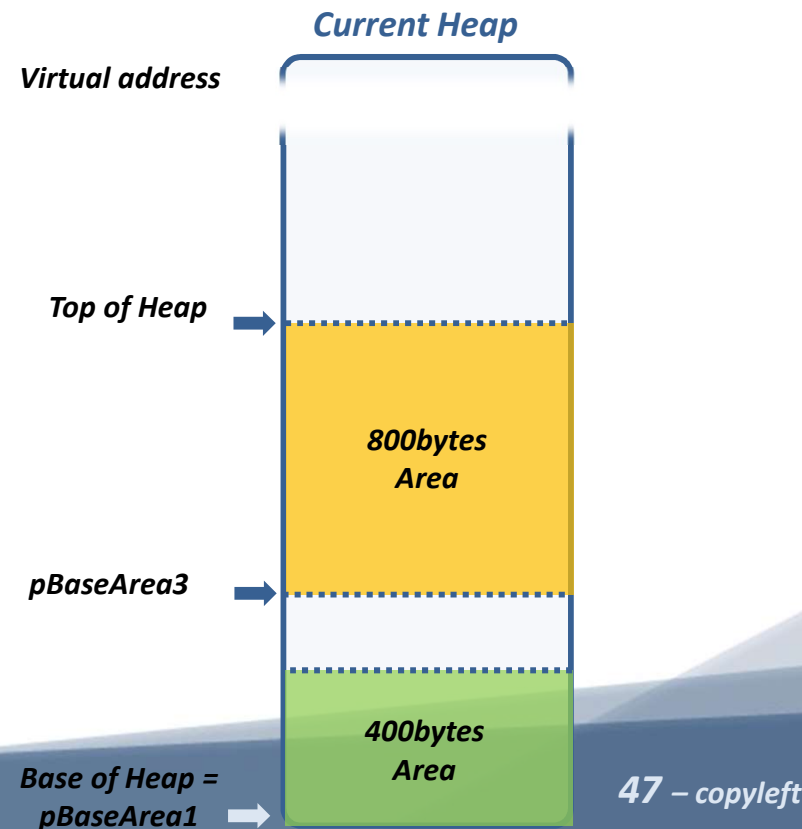
    pBaseArea1 = (float*) malloc (100*sizeof(float));
    pBaseArea2 = (char*) malloc (100*sizeof(char));
    pBaseArea3 = (double*) malloc (100*sizeof(double));

    // user application ...

    free ( pBaseArea2 );
    pBaseArea2 = (char*) malloc (200*sizeof(char));

    // user application ...

    free ( pBaseArea1 );
    free ( pBaseArea2 );
    free ( pBaseArea3 );
    return 0;
}
```



- Segmentation
- **Pile et Tas**
- Pagination
- exceptions et signaux

L'espace mémoire virtuel alloué sera toujours contigu (heureusement), néanmoins l'allocation dynamique de ressources amènera une fragmentation interne de la mémoire physique (géré par l'unité de pagination) :

```
int main (void) {
    float* pBaseArea1;
    char* pBaseArea2;
    double* pBaseArea3;

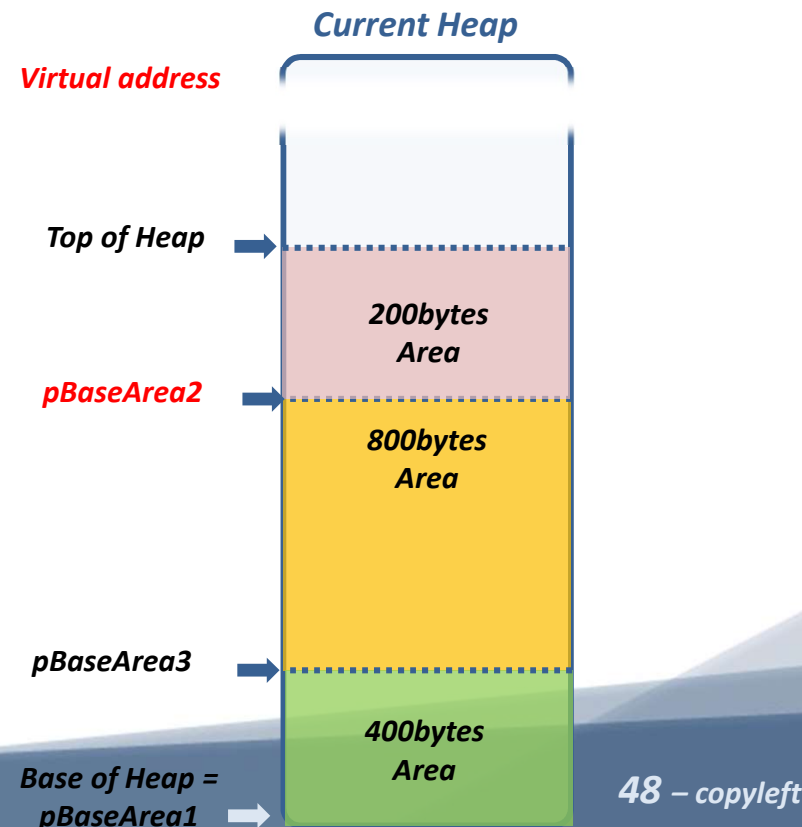
    pBaseArea1 = (float*) malloc (100*sizeof(float));
    pBaseArea2 = (char*) malloc (100*sizeof(char));
    pBaseArea3 = (double*) malloc (100*sizeof(double));

    // user application ...

    free ( pBaseArea2 );
    pBaseArea2 = (char*) malloc (200*sizeof(char));

    // user application ...

    free ( pBaseArea1 );
    free ( pBaseArea2 );
    free ( pBaseArea3 );
    return 0;
}
```



- Segmentation
- **Pile et Tas**
- Pagination
- exceptions et signaux

*Il faut néanmoins être très prudent quant au travail effectif des fonction malloc et free. **Beaucoup de processeurs (souvent sans MMU) et C toolchain associées dans l'embarqué ne gèrent pas ou mal les mécanismes de fragmentation de la mémoire physique.***

```
int main (void) {  
    float* pBaseArea1;  
    char* pBaseArea2;  
    double* pBaseArea3;  
  
    pBaseArea1 = (float*) malloc (100*sizeof(float));  
    pBaseArea2 = (char*) malloc (100*sizeof(char));  
    pBaseArea3 = (double*) malloc (100*sizeof(double));  
  
    // user application ...  
  
    free ( pBaseArea2 );  
    pBaseArea2 = (char*) malloc (200*sizeof(char));  
  
    // user application ...  
  
    free ( pBaseArea1 );  
    free ( pBaseArea2 );  
    free ( pBaseArea3 );  
    return 0;  
}
```

Virtual address

Current Heap

Top of Heap
Base of Heap →

49 – copyleft

- Segmentation
- **Pile et Tas**
- Pagination
- exceptions et signaux

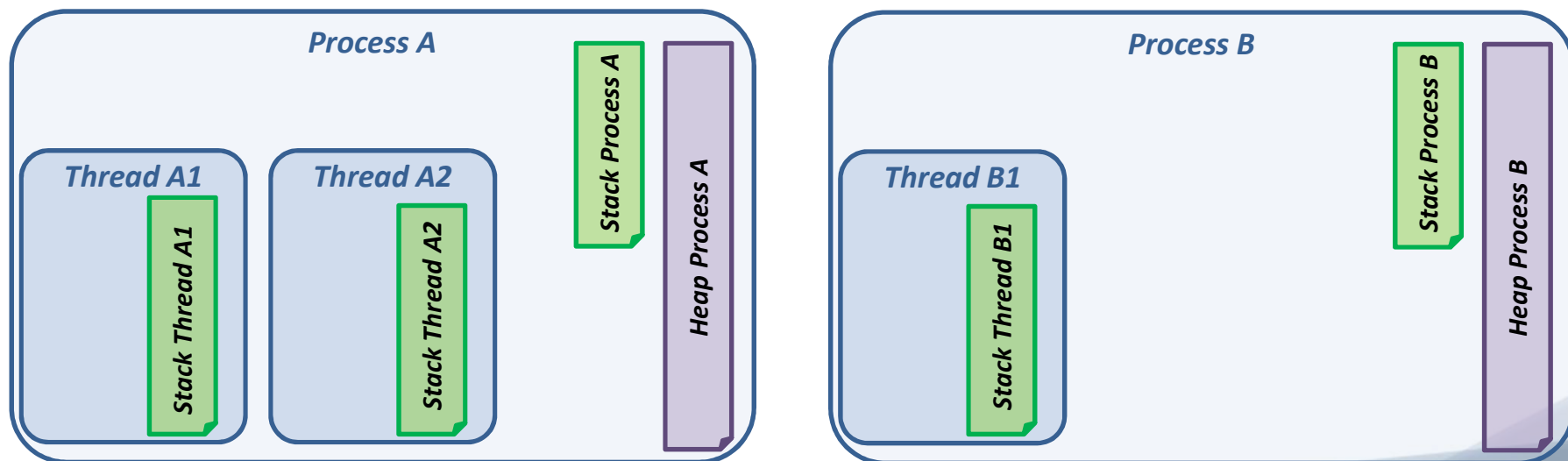
Sous Linux, un processus possède des accès mémoires “illimités” pour l’allocation dynamique de données sur le Tas. En réalité, les allocations dynamiques ne sont possibles que jusqu’aux limites physiques de la machine. Par exemple pour une application user, ~3Go max sur architecture IA-32 (le système se réservant ~1Go). Observons par exemple les ressources mémoire utilisables par le shell :



```
vmlinux@vmlinux: ~/x86-64/heap
vmlinux@vmlinux: ~/x86-64/heap
vmlinux@vmlinux:~/x86-64/heap$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority     (-e) 0
file size                (blocks, -f) unlimited
pending signals         (-i) 15990
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files              (-n) 1024
pipe size               (512 bytes, -p) 8
POSIX message queues    (bytes, -q) 819200
real-time priority      (-r) 0
stack size              (kbytes, -s) 8192
cpu time                (seconds, -t) unlimited
max user processes      (-u) 15990
virtual memory          (kbytes, -v) unlimited
file locks              (-x) unlimited
vmlinux@vmlinux:~/x86-64/heap$ ulimit -m
unlimited
vmlinux@vmlinux:~/x86-64/heap$
```

- Segmentation
- Pile et Tas
- Pagination
- exceptions et signaux

Pour un processus Linux typique exploitant plusieurs threads, le tas est propre au processus et peut être partagé par tous les threads fils. En revanche les piles sont propres à chaque processus et chaque thread. Cela assure une réelle indépendance des contextes d'exécution (variables spatialement séparées en mémoire) :



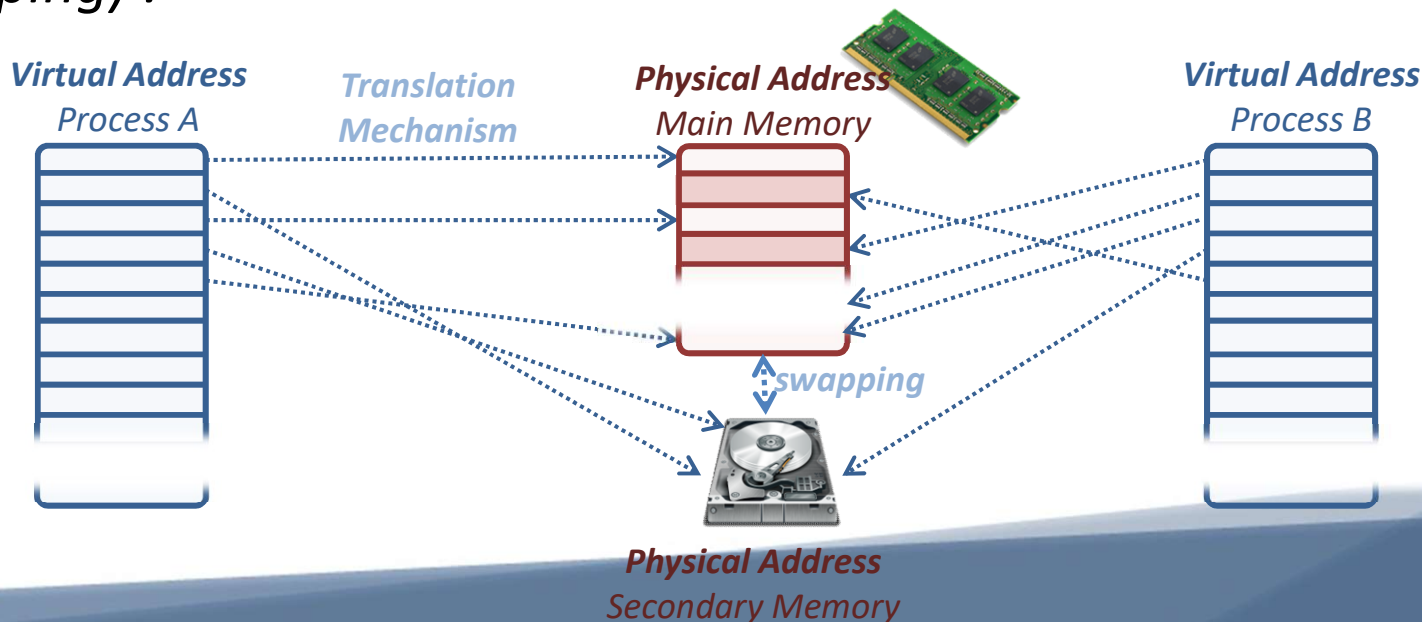
- Segmentation
- Pile et Tas
- **Pagination**
- exceptions et signaux

La pagination mémoire est un mécanisme de virtualisation mémoire extrêmement rencontré et utilisé sur grand nombre d'architectures modernes, architectures x86 comme processeurs pour l'embarqué (attention, pas tous). Les principaux intérêts à travailler avec une mémoire virtuelle paginée sont :

- *Offrir à un processus un **espace mémoire contigu masquant la fragmentation de la mémoire physique.***
- *Proposer un espace mémoire paginé virtuel pouvant dépasser l'espace physique réel de la mémoire principale.*
- *La pagination mémoire facilite également la **protection mémoire** et le partage de ressources mémoire entre processus.*

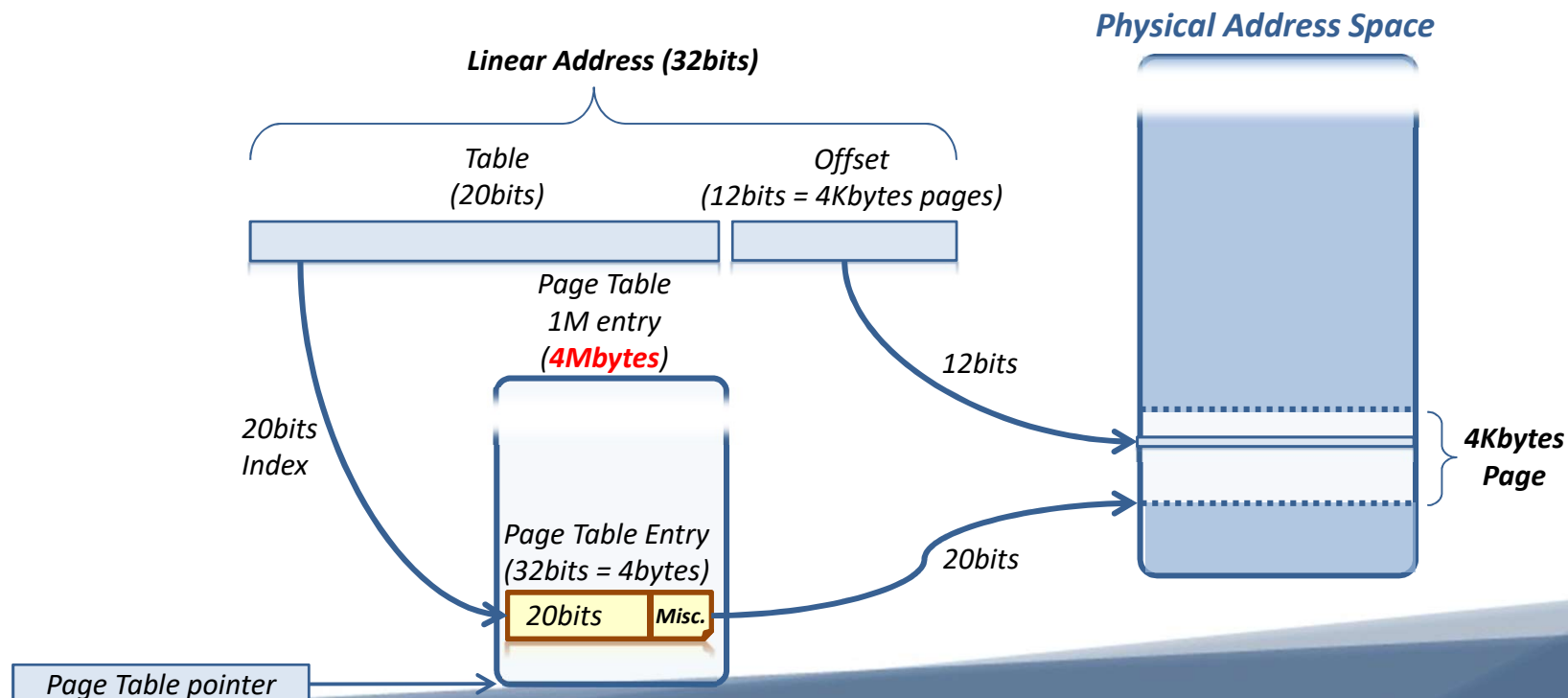
- Segmentation
- Pile et Tas
- **Pagination**
- exceptions et signaux

Une mémoire paginée est découpée en pages fixant ainsi une granularité de la mémoire. La mémoire principale est découpée en frames (cadres) de même taille, chaque cadre contenant une page. Il peut y avoir plus de pages que de cadres, d'où l'intérêt. Les pages ne se trouvant pas en mémoire principale sont généralement en mémoire secondaire et seront chargées en mémoire vive en cas de besoin (swapping) :



- Segmentation
- Pile et Tas
- **Pagination**
- exceptions et signaux

Le mécanisme de translation d'adresse (adresse linéaire vers adresse physique) de l'unité de pagination est extrêmement performant et consiste à une simple consultation de table (tables des pages) :



- Segmentation
- Pile et Tas
- **Pagination**
- exceptions et signaux

Lorsqu'une demande de translation d'adresse est demandée à l'unité de pagination, plusieurs cas peuvent se produire. Il faut savoir qu'à chaque entrée de la table des pages (Page Table Entry) sont associés des champs, notamment un champ de validité :

- **Valid Entry (main memory address)** : translate l'adresse linéaire en adresse physique (cas le plus courant).
- **Invalid Entry** : Si par exemple la page n'est pas mappée en mémoire physique, l'exception matérielle #PF (Page Fault) est levée par le CPU. Le kernel peut prendre l'initiative ou non de la charger en RAM.
- **Valid Entry (secondary memory address)** : déplacement de la page en mémoire de masse vers un cadre libre en mémoire principale.

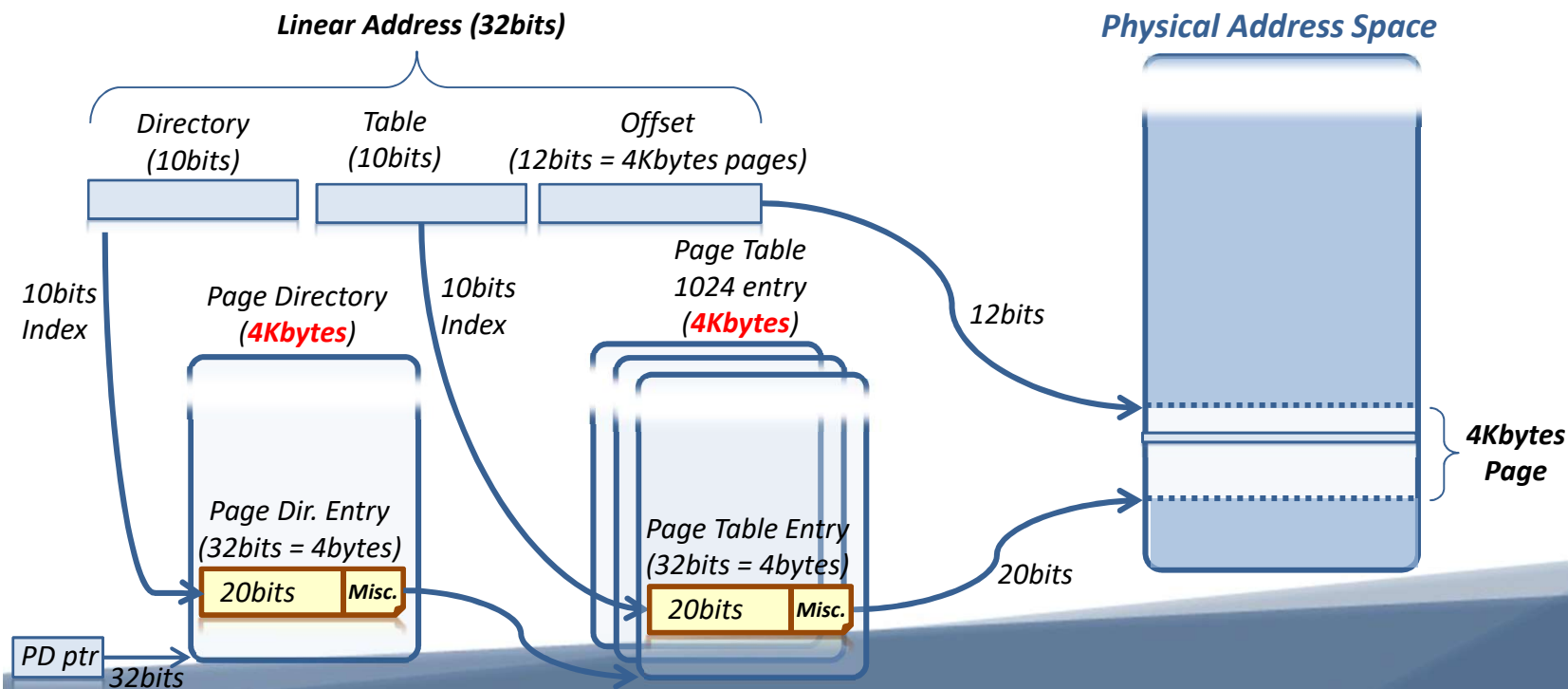
- Segmentation
- Pile et Tas
- **Pagination**
- exceptions et signaux

Une translation d'adresse linéaire invalide peut donc lever une exception matérielle côté CPU (exception #PF vu par la suite) dans deux cas de figure : **Translation d'adresse invalide** ou **Droits d'accès à la page cible invalides**. Observons le code d'erreur retourné par le CPU et donc la nature des défauts pouvant lever l'exception matérielle #PF (souvent à l'origine du célèbre **segmentation fault (core dumped)** de Linux) :

31	Reserved				4	3	2	1	0
					I/D	RSVD	U/S	W/R	P
P	0	The fault was caused by a non-present page.							
	1	The fault was caused by a page-level protection violation.							
W/R	0	The access causing the fault was a read.							
	1	The access causing the fault was a write.							
U/S	0	A supervisor-mode access caused the fault.							
	1	A user-mode access caused the fault.							
RSVD	0	The fault was not caused by reserved bit violation.							
	1	The fault was caused by a reserved bit set to 1 in some paging-structure entry.							
I/D	0	The fault was not caused by an instruction fetch.							
	1	The fault was caused by an instruction fetch.							

- Segmentation
- Pile et Tas
- **Pagination**
- exceptions et signaux

L'un des gros problème amené par le mécanisme présenté précédemment est la taille occupée par la table des pages (4Mo). Il faut savoir que cette table est présente en mémoire principale. Une solution est d'utiliser une seconde table sauvent des pointeurs vers les tables des pages (tables de 4Ko granularité mémoire physique) :



- Segmentation
- Pile et Tas
- **Pagination**
- exceptions et signaux

Observons maintenant les mécanismes de gestion de la pagination des architectures Intel IA-32e ainsi que celle utilisée par Linux depuis la version 2.6.11. Pagination à 4 niveaux (pour des pages de 4Ko) :

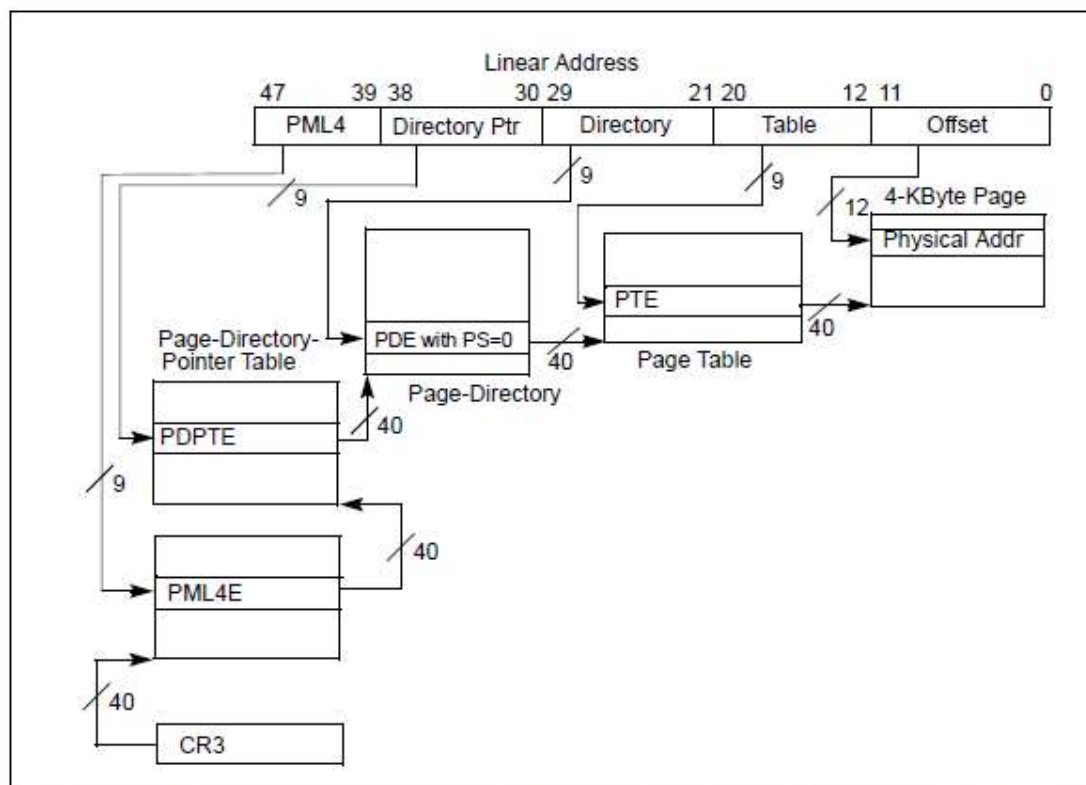


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

- Segmentation
- Pile et Tas
- **Pagination**
- exceptions et signaux

Il faut savoir que les architectures Intel modernes supportent des pages de différentes tailles : 4Ko, 2Mo, 4Mo ou 1Go (dépend du mode de pagination) :

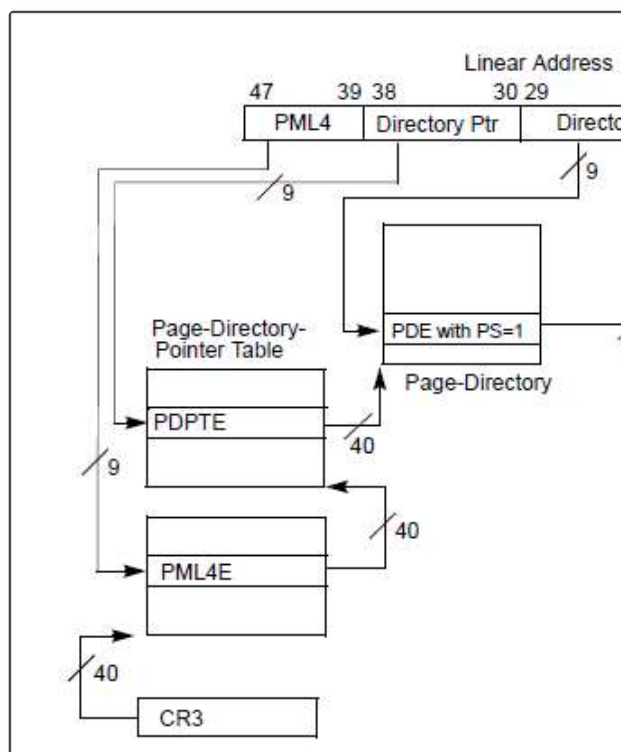


Figure 4-9. Linear-Address Translation to a 2-MB Page using IA-32e Paging

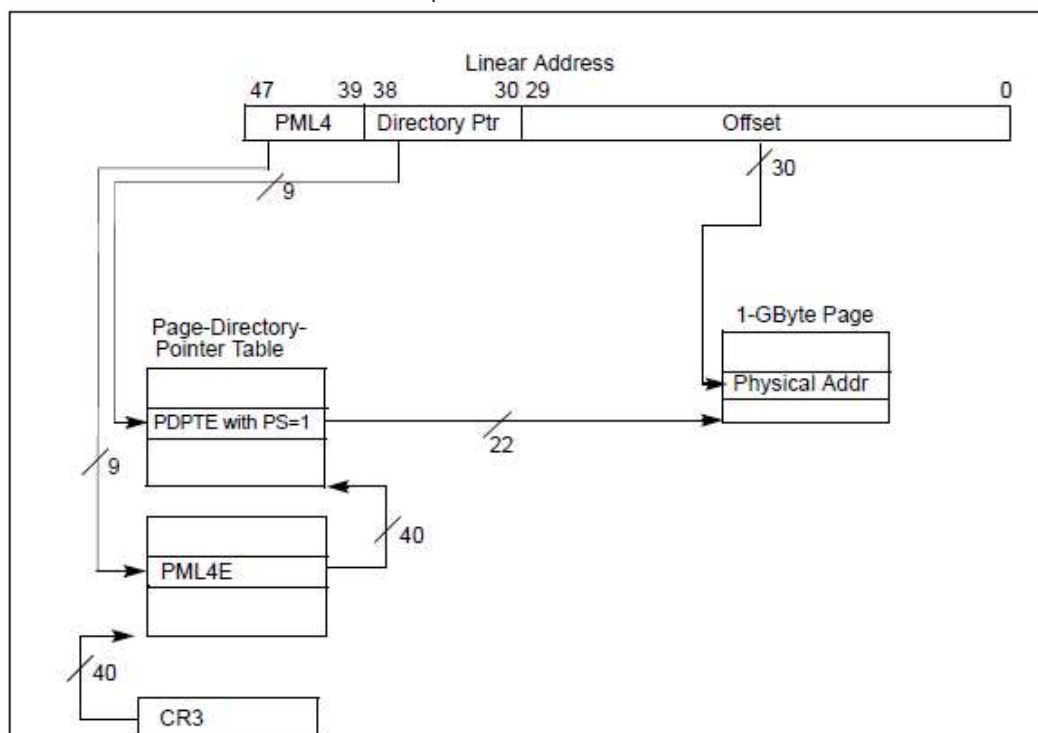
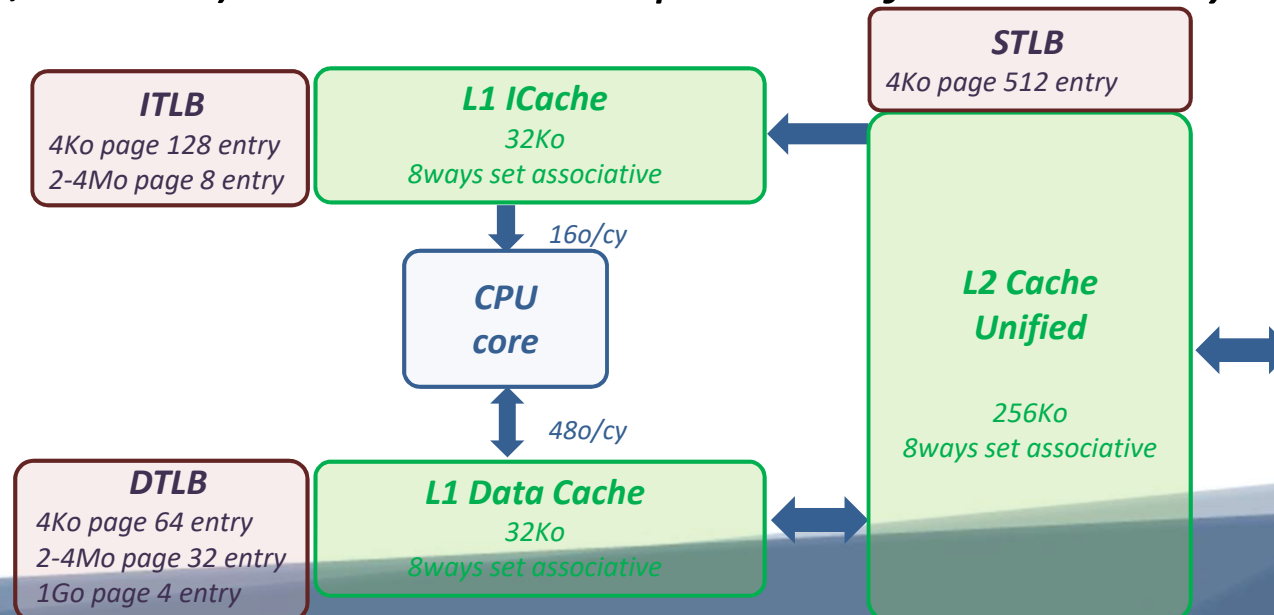


Figure 4-10. Linear-Address Translation to a 1-GB Page using IA-32e Paging

- Segmentation
- Pile et Tas
- **Pagination**
- exceptions et signaux

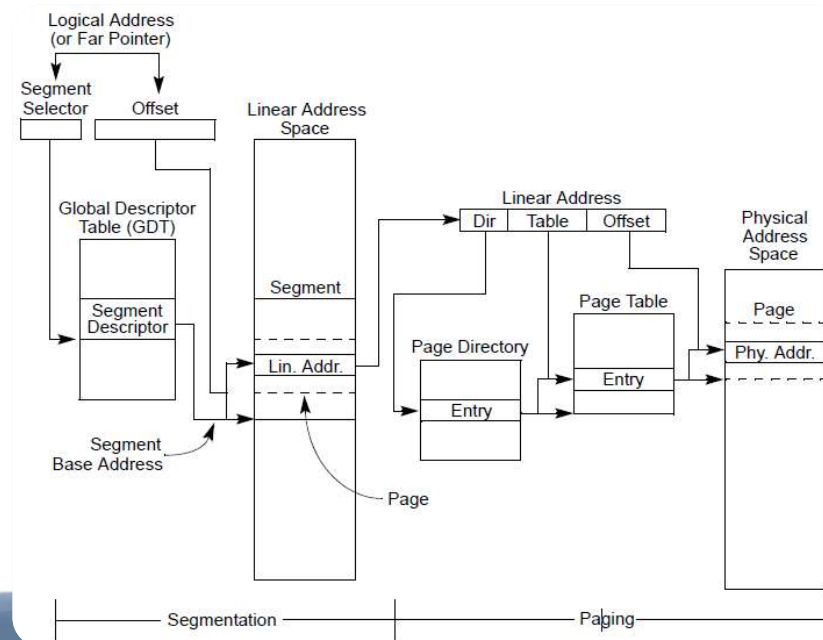
Un dernier mécanisme d'optimisation utilisé afin d'accélérer les mécanismes de translation d'adresse est d'utiliser de petites mémoires cache associatives (Table Lookaside Buffer ou TLB) chargées de sauver les entrées les plus couramment appelées. Les stratégies de remplacement des entrées des TLB's sont semblables aux techniques de gestion des caches processeur et seront vues par la suite (LRU, random, FIFO ...). Prenons l'exemple de la famille Sandy Bridge :



- Segmentation
- Pile et Tas
- **Pagination**
- exceptions et signaux



En résumant, sous Linux la segmentation matérielle est très peu utilisée et est essentiellement manipulée pour des soucis de protection mémoire (privilèges). En revanche la pagination est extrêmement usitée afin d'adresser un espace mémoire virtuel contigu pouvant être supérieur aux ressources réelles de la mémoire principale qui elle est fragmentée (transparent pour le développeur).



- Segmentation
- Pile et Tas
- **Pagination**
- exceptions et signaux



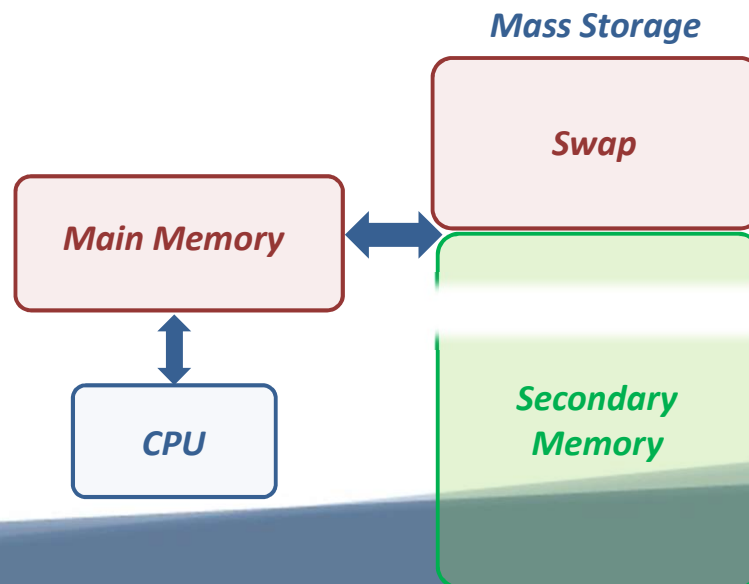
L'espace d'adressage virtuel pouvant être supérieur à l'espace physique en mémoire principale, Linux utilise une zone d'échange en mémoire secondaire nommée swap (swap signifie échange). Cette zone doit être vue comme une extension de la mémoire principale. Il s'agit sous Linux d'une partition du disque dur permettant l'échange de pages entre la mémoire vive ayant une taille restreinte et le disque. Attention, il ne s'agit pas d'un cache (copie), mais bien d'une zone d'échange. Prenons un exemple en langage C de swapping de variables :

```
int main(void){
    char data1 = 1, data2 = 2;
    swap (&data1, &data2);
    // after swapping, data1 = 2 and data2 = 1
    return 0;
}

Void swap ( char* pData1, char* pData2) {
    char tmp = *pData1;
    *pData1 = *pData2;
    *pData2 = tmp;
}
```

- Segmentation
- Pile et Tas
- **Pagination**
- exceptions et signaux

Afin d'obtenir de bonnes performances, cette zone doit si possible être exploitée le moins possible, car du swapping trop fréquent peut ralentir les performances du système. Le swap est typiquement utilisé par les applications gourmandes en ressources mémoire, durant la mise en veille (contenu mémoire vive sauvé en swap) ... La taille du swap Linux doit-être typiquement comprise entre 1x et 1,5x la taille de la mémoire vive.



- Segmentation
- Pile et Tas
- **Pagination**
- exceptions et signaux

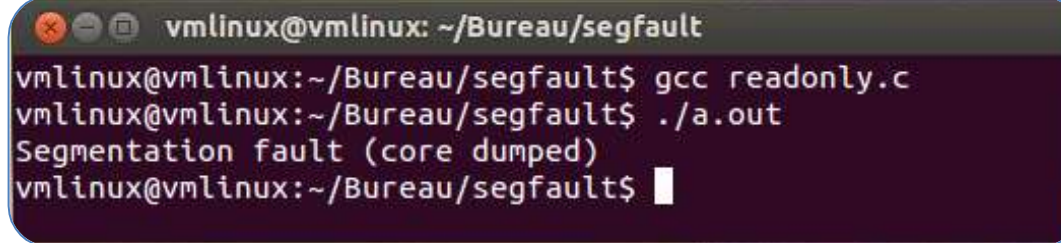
*Il est possible de jouer sur la politique de remplacement des pages en forçant le système à swapper le moins possible de ressources et exploiter au mieux les ressources en mémoire principale. Pour ce faire il faut modifier le paramètre système **/proc/sys/vm/swappiness** valant par défaut 60 (après modification, ~10 sur mes machines).*

- **swappiness = 0** : si possible, usage massif de la mémoire vive
- **swappiness = 100** : usage massif du swap en mémoire secondaire

```
vmlinux@vmlinux: ~  
vmlinux@vmlinux:~$ cat /proc/sys/vm/swappiness  
60  
vmlinux@vmlinux:~$ sudo sysctl vm.swappiness=10  
[sudo] password for vmlinux:  
vm.swappiness = 10  
vmlinux@vmlinux:~$ sudo swapoff -av  
swapoff sur /dev/sda5  
vmlinux@vmlinux:~$ sudo swapon -av  
swapon sur /dev/sda5  
swapon: /dev/sda5 : signature de l'espace d'échange t  
de page 4, ordre des octets identique  
swapon: /dev/sda5: pagesize=4096, swapspace=2145386496  
vmlinux@vmlinux:~$ cat /proc/sys/vm/swappiness  
10
```


- Segmentation
- Pile et Tas
- Pagination
- **exceptions et signaux**

*Intéressons-nous au célèbre **segmentation fault (core dumped)** de linux. Pour bien assimiler cette partie, nous nous intéresserons aux exceptions matérielles relevées par le processeur, aux mécanismes de préemption par le kernel et de communication avec les tâches applicatives à la source du défaut.*

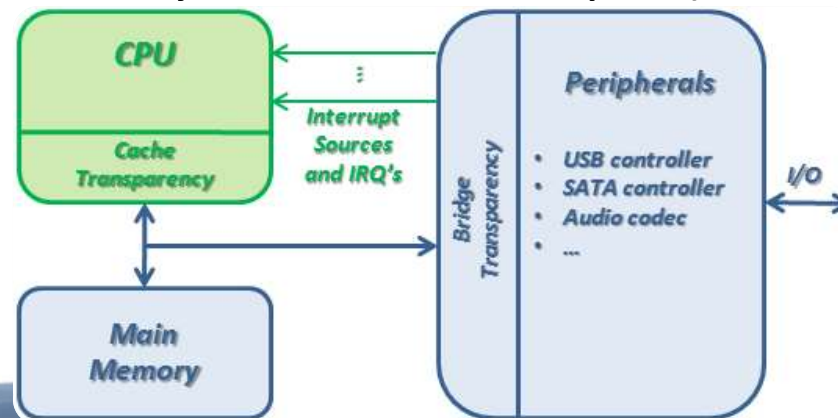
A terminal window with a dark background and light text. The window title is 'vmlinux@vmlinux: ~/Bureau/segfault'. The terminal shows the following commands and output:

```
vmlinux@vmlinux:~/Bureau/segfault$ gcc readonly.c
vmlinux@vmlinux:~/Bureau/segfault$ ./a.out
Segmentation fault (core dumped)
vmlinux@vmlinux:~/Bureau/segfault$
```

- Segmentation
- Pile et Tas
- Pagination
- **exceptions et signaux**

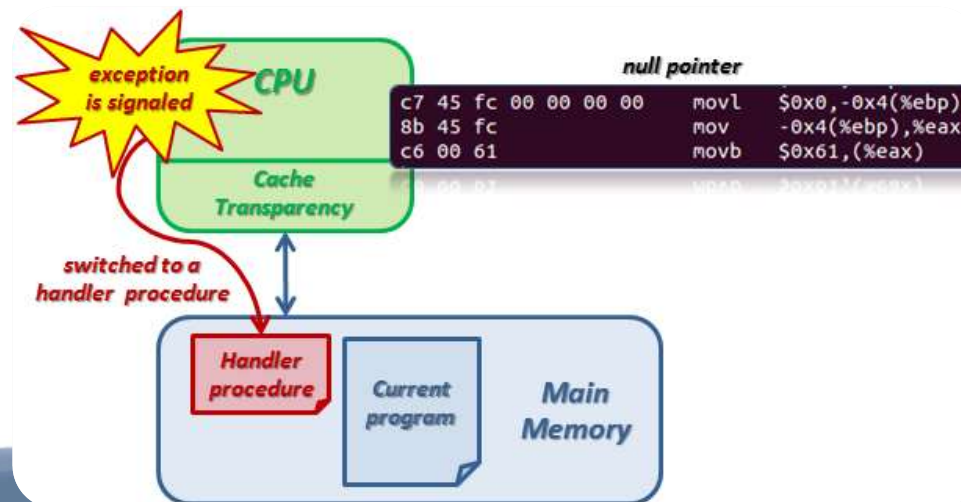
Avant d'appréhender la partie propre au système d'exploitation, attardons nous sur les mécanismes d'interruption d'un programme en cours d'exécution. Comme pour une grande majorité des processeurs à architecture CPU, deux mécanismes cohabitent :

- **Interrupt** : Evènement matériel asynchrone de communication typiquement utilisé par les périphériques (vu durant les enseignements de systèmes embarqués).



- Segmentation
- Pile et Tas
- Pagination
- **exceptions et signaux**

- **Exception** : Evènement matériel synchrone généré par le CPU (synchrone au regard du fonctionnement d'un CPU dont les traitements restent synchronisés sur une référence d'horloge, pas au regard de la probabilité d'occurrence). Ces événements sont relevés par le CPU lorsque celui-ci détecte une voire plusieurs conditions prédéfinies durant l'exécution d'une instruction (violation de privilège, division flottante par zéro, accès illégal en mémoire ...).



- Segmentation
- Pile et Tas
- Pagination
- **exceptions et signaux**

*Lorsqu'une interruption ou une exception se produit, le CPU stoppe l'exécution du programme en cours et donne la main à une procédure spécifiquement écrite pour traiter l'évènement matériel venant de se produire (fonction `do_page_fault` du kernel Linux dans le cas d'exception, présent dans `/arch/<cpu>/mm/fault.c`). Ces fonctions sont appelées *ISR's* (Interrupt Software/Service Routine) dans le cadre des interruptions. Observons les familles d'exceptions supportées sur architecture Intel IA-32 :*

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode. ¹
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction. ²
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory. ³
18	#MC	Machine Check	Error codes (if any) and source are model dependent. ⁴
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction ⁵
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT <i>n</i> instruction.

- Segmentation
- Pile et Tas
- Pagination
- exceptions et signaux

Chacune de ces familles d'exception englobe différents types de défauts. Prenons en une à titre d'exemple, celle associée aux opérations arithmétiques flottantes hors instructions vectorielles (#MF ou Math Fault) :

Interrupt 16—x87 FPU Floating-Point Error (#MF)

Exception Class Fault.

Description

Indicates that the x87 FPU has detected a floating-point error. The NE flag in the register CR0 must be set for an interrupt 16 (floating-point error exception) to be generated. (See Section 2.5, "Control Registers," for a detailed description of the NE flag.)

NOTE

SIMD floating-point exceptions (#XM) are signaled through interrupt 19.

While executing x87 FPU instructions, the x87 FPU detects and reports six types of floating-point error conditions:

- Invalid operation (#I)
 - Stack overflow or underflow (#IS)
 - Invalid arithmetic operation (#IA)
- Divide-by-zero (#Z)
- Denormalized operand (#D)
- Numeric overflow (#O)
- Numeric underflow (#U)
- Inexact result (precision) (#P)

- Segmentation
- Pile et Tas
- Pagination
- **exceptions et signaux**

Sur architecture Intel, les exceptions matérielles sont répertoriées en 3 grandes classes :

- ***Fault*** : *Lorsqu'une exception de ce type arrive, elle peut en général être corrigée et peut potentiellement autoriser la continuité d'exécution du programme ayant causé le défaut (dépend de la stratégie de l'OS). Sous Linux, si un défaut de ce type est détecté, le kernel prend l'initiative d'envoyer un signal (SIGSEGV, SIGFPE, SIGILL, SIGBUS) au processus à la cause du défaut. Par défaut, le processus est alors mis à mort.*
- ***Abort*** : *Défaut critique pour le système, le processus en cause n'est pas autorisé à reprendre la main*

- Segmentation
- Pile et Tas
- Pagination
- **exceptions et signaux**

- **Trap:** Ce type d'exception n'est pas un défaut matériel, prenons l'exemple de l'exception #BP ou Break Point. Il s'agit, dans le cas présent, d'un opcode de 1 octet (instruction breakpoint = INT3) remplaçant le premier octet de l'opcode de chaque instruction du programme sous test. Ce type d'exception peut être utilisé comme alternative par les outils de debuggage (signal SIGTRAP). En effet, le debugger sera alors appelé à l'exécution de chaque instruction.

```
1 /**
2 * @file sigtrap.c
3 * @brief exception #BP et signal UNIX SIGTRAP
4 * @author
5 * @date novembre 2013
6 */
7 #include <stdio.h>
8
9 /**
10 * @fn void main (void)
11 * @brief program entry point
12 */
13 int main(int argc, char **argv) {
14     __asm__("int3");
15     return 0;
16 }
```

```
080483b4 <main>:
80483b4: 55          push    %ebp
80483b5: 89 e5       mov     %esp,%ebp
80483b7: cc         int3
80483b8: b8 00 00 00 00 mov     $0x0,%eax
80483bd: 5d         pop     %ebp
80483be: c3         ret
```

```
vmlinux@vmlinux: ~/Bureau/segfault
vmlinux@vmlinux:~/Bureau/segfault$ gcc sigtrap.c
vmlinux@vmlinux:~/Bureau/segfault$ ./a.out
Trace/breakpoint trap (core dumped)
vmlinux@vmlinux:~/Bureau/segfault$
```

- Segmentation
- Pile et Tas
- Pagination
- **exceptions et signaux**

Intéressons nous maintenant aux signaux UNIX et à l'implémentation sur système UNIX-like comme Linux. Attention, ne pas confondre :

- **interruptions et exceptions** : événements matériels
- **signaux** : notifications logicielles asynchrones envoyées par le kernel à un processus ou thread cible suite à un événement matériel ou logiciel. Les appels système **kill** et **signal** permettent respectivement d'envoyer un signal (événement logiciel) et de capturer un signal (événement matériel ou logiciel) depuis un processus.

- Segmentation
- Pile et Tas
- Pagination
- **exceptions et signaux**

En cas d'occurrence, le kernel stoppe l'exécution du processus cible (fait au niveau hardware par le CPU si exception matérielle), celui-ci exécute alors une procédure spécifiquement écrite par le développeur afin de traiter le signal. Sinon, une procédure par défaut est appliquée. Observons les signaux supportés par Linux :

```
vmlinux@vmlinux: ~/Bureau/segfault
vmlinux@vmlinux:~/Bureau/segfault$ kill -l
1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP
6) SIGABRT     7) SIGBUS     8) SIGFPE     9) SIGKILL     10) SIGUSR1
11) SIGSEGV    12) SIGUSR2    13) SIGPIPE    14) SIGALRM     15) SIGTERM
16) SIGSTKFLT  17) SIGCHLD    18) SIGCONT    19) SIGSTOP     20) SIGTSTP
21) SIGTTIN    22) SIGTTOU    23) SIGURG     24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF    28) SIGWINCH   29) SIGIO       30) SIGPWR
31) SIGSYS     34) SIGRTMIN   35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
vmlinux@vmlinux:~/Bureau/segfault$
```

- Segmentation
- Pile et Tas
- Pagination
- **exceptions et signaux**

Observons les signaux système associés à des exceptions matérielles :

- **SIGSEGV** : le plus célèbre, signal générant le fameux **segmentation fault (core dumped)**. Plusieurs exceptions matérielles peuvent en être à la source : #PF page fault (page non présente en mémoire physique, exécution d'une page non-executable...), #GP General Protection (nombreux défauts principalement associés à des accès mémoire illégaux : écriture sur segment read-only, lecture d'un segment execute-only, dépassement taille limite de segment, violation de privilège, null segment selector ...), ...
- **SIGILL** : exécution d'un opcode invalide (exception #UD Invalid Opcode Exception)

- Segmentation
- Pile et Tas
- Pagination
- **exceptions et signaux**

- **SIGBUS** : *détection d'erreur sur bus physique. Par exemple détection de défauts d'alignement (exception #AC Alignement Check Exception) ou d'adresses physiques invalides (exception #MC Machine-Check, architecture CPU dépendant).*
- **SIGFPE** : *détection d'opérations arithmétiques erronées, par exemple division par zéro, valeur dé-normalisée, overflow ou underflow arithmétique ... (exceptions #XM SIMD floating point, #MF x87 floating point ...).*
- **SIGTRAP** : *vu précédemment, principalement utilisé par les outils de debug. Il ne s'agit pas de défaut mais d'exception matérielles voulues.*

- Segmentation
- Pile et Tas
- Pagination
- **exceptions et signaux**

Prenons quelques exemples permettant d'illustrer quelques unes des exceptions précédemment présentées. :

- **SIGBUS** : défaut d'alignement (activation matérielle nécessaire côté processeur)

```
/**
 * @fn void main (void)
 * @brief program entry point
 */
int main(int argc, char **argv) {
    int *pInt;
    char *pArea;

    // enable alignment checking
    #if defined(__GNUC__)
    # if defined(__i386__)
        // x86 architecture
        __asm__("pushf\norl $0x40000, (%esp)\npopf");
    # elif defined(__x86_64__)
        // x86_64 architecture
        __asm__("pushf\norl $0x40000, (%rsp)\npopf");
    # endif
    #endif

    // malloc() always provides aligned memory
    pArea = (char *) malloc(sizeof(int)+1);

    // increment the pointer by value different of modulo sizeof(*pInt)
    // making it misaligned
    pInt = (int *) (pArea+=3);           // nok
    //pInt = (int *) (pArea+=sizeof(*pInt)); // ok

    // unaligned access (dereference pointer)
    *pInt = 51;

    return 0;
}
```

```
vmlinux@vmlinux: ~/Bureau/segfault
vmlinux@vmlinux:~/Bureau/segfault$ gcc buserror.c
vmlinux@vmlinux:~/Bureau/segfault$ ./a.out
Bus error (core dumped)
vmlinux@vmlinux:~/Bureau/segfault$
```

- Segmentation
- Pile et Tas
- Pagination
- **exceptions et signaux**

Prenons quelques exemples permettant d'illustrer quelques unes des exceptions précédemment présentées. :

- **SIGBUS** : défaut d'alignement (activation matérielle nécessaire côté processeur)

```
**
* @fn void main (void)
* @brief program entry point
*/
int main(int argc, char **argv) {
    int *pInt;
    char *pArea;

    // enable alignment checking
    #if defined(__GNUC__)
    # if defined(__i386__)
        // x86 architecture
        __asm__("pushf\norl $0x40000, (%esp)\npopf");
    # elif defined(__x86_64__)
        // x86_64 architecture
        __asm__("pushf\norl $0x40000, (%rsp)\npopf");
    # endif
    #endif

    // malloc() always provides aligned memory
    pArea = (char *) malloc(sizeof(int)+1);

    // increment the pointer by value different of modulo sizeof(*pInt)
    // making it misaligned
    pInt = (int *) (pArea+=3);           // nok
    //pInt = (int *) (pArea+=sizeof(*pInt)); // ok

    // unaligned access (dereference pointer)
    *pInt = 51;

    return 0;
}
```

```
vmlinux@vmlinux: ~/Bureau/segfault
vmlinux@vmlinux:~/Bureau/segfault$ gcc buserror.c
vmlinux@vmlinux:~/Bureau/segfault$ ./a.out
Bus error (core dumped)
vmlinux@vmlinux:~/Bureau/segfault$
```


- Segmentation
- Pile et Tas
- Pagination
- **exceptions et signaux**

- **SIGFPE** : erreur arithmétique, division par zéro (exception #MF)

```
/**
 * @fn void main (void)
 * @brief program entry point
 */
int main(int argc, char **argv) {
    volatile int value = 3, zero = 0.0;
    value /= zero;

    return 0;
}
```

```
vmlinux@vmlinux: ~/Bureau/segfault
vmlinux@vmlinux:~/Bureau/segfault$ gcc divzero.c
vmlinux@vmlinux:~/Bureau/segfault$ ./a.out
Floating point exception (core dumped)
vmlinux@vmlinux:~/Bureau/segfault$
```

- **SIGSEGV** : erreur de segmentation (probablement exception #PF)

```
/**
 * @fn void main (void)
 * @brief program entry point
 */
int main(int argc, char **argv) {
    volatile char* ptr; // uninitialized pointer

    printf("current stack area = 0x%x\n", (unsigned int) &ptr);
    printf("ptr point anywhere = 0x%x\n", (unsigned int) ptr);

    *ptr = 'H';

    return 0;
}
```

```
vmlinux@vmlinux: ~/Bureau/segfault
vmlinux@vmlinux:~/Bureau/segfault$ gcc anywhere.c
vmlinux@vmlinux:~/Bureau/segfault$ ./a.out
current stack area = 0xbf985dbc
ptr point anywhere = 0xb77b6ff4
Segmentation fault (core dumped)
vmlinux@vmlinux:~/Bureau/segfault$
```

- Segmentation
- Pile et Tas
- Pagination
- *exceptions et signaux*

- ***SIGSEGV*** : écriture en zone read-only (exception #GP)

```
/**
 * @fn void main (void)
 * @brief program entry point
 */
int main(int argc, char **argv) {
    char *str = "Hello World";

    *str = 'a';

    return 0;
}
LEGEND 0:
```

```
vmlinux@vmlinux: ~/Bureau/segfault
vmlinux@vmlinux:~/Bureau/segfault$ gcc readonly.c
vmlinux@vmlinux:~/Bureau/segfault$ ./a.out
Segmentation fault (core dumped)
vmlinux@vmlinux:~/Bureau/segfault$
```

```
Contents of section .rodata:
8048498 03000000 01000200 48656c6c 6f20576f .....Hello Wo
80484a8 726c6400                                rld.
```


- Segmentation
- Pile et Tas
- Pagination
- **exceptions et signaux**

- **SIGSEGV** : intéressons-nous au célèbre **stack overflow**. En mode réel, celui-ci est notamment détecté par l'exécution d'instructions des familles PUSH et POP capables de lever l'exception #SS (Stack Fault) en cas de dépassement de limite du Stack Segment. Dans les autres modes mémoire, il est en général détecté par exception matérielle #PF (Page Fault), la pile étant de taille multiple de la taille d'une page mémoire. L'instruction PUSH est également capable de lever l'exception #PF (hors mode réel).

```
/**  
 * @fn void main (void)  
 * @brief program entry point  
 */  
int main(void) {  
    main();  
    return 0;  
}
```

```
080483b4 <main>:  
80483b4: 55          push    %ebp  
80483b5: 89 e5       mov     %esp,%ebp  
80483b7: 83 e4 f0    and     $0xffffffff0,%esp  
80483ba: e8 f5 ff ff call    80483b4 <main>  
80483bf: b8 00 00 00 mov     $0x0,%eax  
80483c4: c9         leave   %ebp  
80483c5: c3         ret     4  
80483c2: c3  
80483c4: c3
```

```
vmlinux@vmlinux: ~/Bureau/segfault  
vmlinux@vmlinux:~/Bureau/segfault$ gcc stackoverflow.c  
vmlinux@vmlinux:~/Bureau/segfault$ ./a.out  
Segmentation fault (core dumped)  
vmlinux@vmlinux:~/Bureau/segfault$
```

En cours de création



- **Définition d'un cache**
- **Caches processeurs**
- **Politiques de remplacement des lignes de cache**
 - ✓ LRU
 - ✓ Random
 - ✓ FIFO
- **Méthodes d'accès**
 - ✓ Fully associative
 - ✓ Direct mapped
 - ✓ N-way set associative

En cours de création



- **Définition**
- **Technologies**

En cours de création



- **Définition**
- **Disque dur**
- ✓ **Marchés**
- ✓ **Structure mécanique et géométrie**
- ✓ **Technologies**
- **Autres supports**
- ✓ **Clés usb**
- ✓ **SDcards**
- ✓ **DVD's**
- ✓ **...**

Merci de votre attention !