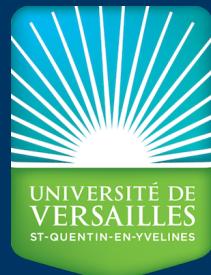


ELF & Mémoire d'un programme

M1 - CHPS

Architecture Interne des Systèmes d'exploitations (AISE)

Jean-Baptiste Besnard
<jean-
baptiste.besnard@paratools.com>



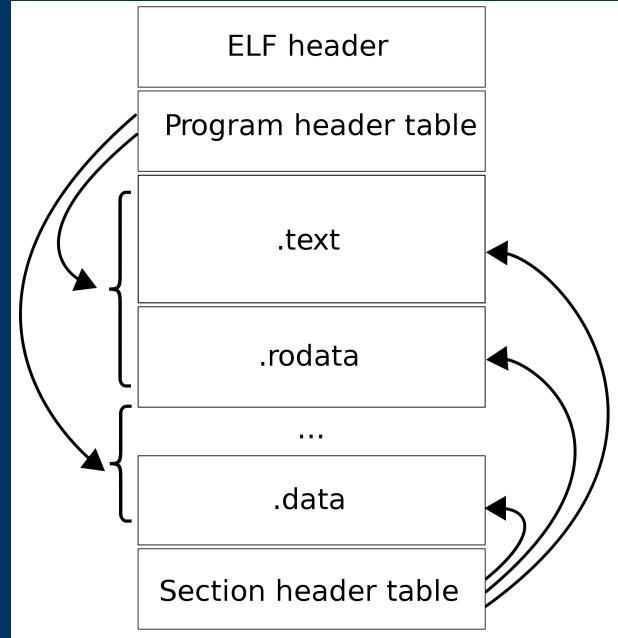
Outils utiles

- Compilateurs (C): **gcc**, **icc**, **xlc**, **clang**, **pgcc**...
 - Dont intermédiaires : **cpp**, **as**, **ld/gold**
- Debuggers : **gdb**, **ddt**, **lldb**, **adb**
- Analyse binaire (disassembling) :
 - ELF : **readelf**, **hte**, **elfedit**, **nm**
 - Objets : **objdump**, **objcopy**
 - Conversion : **xxd**, **hexdump**, **base64**
- Bonus : radare2, peda
- Opcodes x86_64 : <http://ref.x86asm.net/coder64.html>

Format ELF

- ELF = *Executable and Linkable Format*
- Décrit comment un binaire doit être représenté pour être compris par le lanceur de processus
- Un programme contient beaucoup d'informations. Pour rester cohérent, il est découpé en plusieurs sections
- Séquence magique : « **7F 45 4C 46** » = 7F« ELF »
- L'en-tête du ELF contient toutes les informations nécessaires à l'architecture (32/64 bits, endianness, ABI, type de fichier, jeu d'instruction...)

man elf



En-tête ELF:
Magique: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Classe: ELF64
Données: complément à 2, système à octets
Version: 1 (current)
OS/ABI: UNIX - System V
Version ABI: 0
Type: EXEC (fichier exécutable)
Machine: Advanced Micro Devices X86-64
Version: 0x1
Adresse du point d'entrée: 0x4003b0
Début des en-têtes de programme : 64 (octets dans le fichier)
Début des en-têtes de section : 6360 (octets dans le fichier)
Fanions: 0x0
Taille de cet en-tête: 64 (octets)
Taille de l'en-tête du programme: 56 (octets)
Nombre d'en-tête du programme: 9
Taille des en-têtes de section: 64 (octets)
Nombre d'en-têtes de section: 27
Table d'index des chaînes d'en-tête de section: 26

Format ELF

ELF header (Ehdr)

The ELF header is described by the type `Elf32_Ehdr` or `Elf64_Ehdr`:

```
#define EI_NIDENT 16

typedef struct {
    unsigned char e_ident[EI_NIDENT];
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    ElfN_Addr    e_entry;
    ElfN_Off     e_phoff;
    ElfN_Off     e_shoff;
    uint32_t      e_flags;
    uint16_t      e_ehsize;
    uint16_t      e_phentsize;
    uint16_t      e_phnum;
    uint16_t      e_shentsize;
    uint16_t      e_shnum;
    uint16_t      e_shstrndx;
} ElfN_Ehdr;
```

Format ELF

- **Program header** : Stocke les informations nécessaires à la création de l'image du processus. Structure le programme d'un point de vue mémoire

- **Section header** : Regroupe les informations nécessaires au bon fonctionnement du programme. Structure le programme d'un point de vue fonctionnel

- Le reste du ELF est composé de blocs d'instructions, indexées dans l'une et/ou l'autre de ces tables

En-têtes de section :										
[Nr]	Nom	Type	Adr	Décalage	Taille	ES	Fan	LN	Inf	Al
[0]	NULL		0000000000000000	00000000	00000000	00		0	0	0
[1]	.interp	PROGBITS	0000000000400238	000238	00001c	00	A	0	0	1
[2]	.note.ABI-tag	NOTE	0000000000400254	000254	000020	00	A	0	0	4
[3]	.note.gnu.build-id	NOTE	0000000000400274	000274	000024	00	A	0	0	4
[4]	.gnu.hash	GNU_HASH	0000000000400298	000298	00001c	00	A	5	0	8
[5]	.dynsym	DYNSYM	00000000004002b8	0002b8	000048	18	A	6	1	8
[6]	.dynstr	STRTAB	0000000000400300	000300	000040	00	A	0	0	1
[7]	.gnu.version	VERSYM	0000000000400340	000340	000006	02	A	5	0	2
[8]	.gnu.version_r	VERNEED	0000000000400348	000348	000020	00	A	6	1	8
[9]	.rela.dyn	RELA	0000000000400368	000368	000030	18	A	5	0	8
[10]	.init	PROGBITS	0000000000400398	000398	000017	00	AX	0	0	4
[11]	.text	PROGBITS	00000000004003b0	0003b0	000181	00	AX	0	0	16
[12]	.fini	PROGBITS	0000000000400534	000534	000009	00	AX	0	0	4
[13]	.rodata	PROGBITS	0000000000400540	000540	000010	00	A	0	0	8
[14]	.eh_frame_hdr	PROGBITS	0000000000400550	000550	000044	00	A	0	0	4
[15]	.eh_frame	PROGBITS	0000000000400598	000598	000118	00	A	0	0	8
[16]	.init_array	INIT_ARRAY	0000000000600e40	000e40	000008	08	WA	0	0	8
[17]	.fini_array	FINI_ARRAY	0000000000600e48	000e48	000008	08	WA	0	0	8
[18]	.dynamic	DYNAMIC	0000000000600e50	000e50	0001a0	10	WA	6	0	8
[19]	.got	PROGBITS	0000000000600ff0	000ff0	000010	08	WA	0	0	8
[20]	.got.plt	PROGBITS	0000000000601000	001000	000018	08	WA	0	0	8
[21]	.data	PROGBITS	0000000000601018	001018	000004	00	WA	0	0	1
[22]	.bss	NOBITS	000000000060101c	00101c	000004	00	WA	0	0	1
[23]	.comment	PROGBITS	0000000000000000	00101c	000058	01	MS	0	0	1
[24]	.symtab	SYMTAB	0000000000000000	001078	0005a0	18		25	41	8
[25]	.strtab	STRTAB	0000000000000000	001618	0001c0	00		0	0	1
[26]	.shstrtab	STRTAB	0000000000000000	0017d8	0000f9	00		0	0	1

En-têtes de programme :								
Type	Décalage	Adr. vir.	Adr.phys.	T.Fich.	T.Mém.	Fan	Alignement	
PHDR	0x000040	0x0000000000400040	0x0000000000400040	0x0001f8	0x0001f8	R E	0x8	
INTERP	0x000238	0x0000000000400238	0x0000000000400238	0x00001c	0x00001c	R	0x1	
	[Réquisition de l'interpréteur de programme: /lib64/ld-linux-x86-64.so.2]							
LOAD	0x000000	0x0000000000400000	0x0000000000400000	0x0006b0	0x0006b0	R	0x200000	
LOAD	0x000e40	0x0000000000600e40	0x0000000000600e40	0x0001dc	0x0001dc	0	0x200000	
DYNAMIC	0x000e50	0x0000000000600e50	0x0000000000600e50	0x0001a0	0x0001a0	RW	0x8	
NOTE	0x000254	0x0000000000400254	0x0000000000400254	0x000044	0x000044	R	0x4	
GNU_EH_FRAME	0x0000550	0x0000000000400550	0x0000000000400550	0x000044	0x000044	R	0x4	
GNU_STACK	0x000000	0x0000000000000000	0x0000000000000000	0x000000	0x000000	RW	0x10	
GNU_RELRO	0x000e40	0x0000000000600e40	0x0000000000600e40	0x0001c0	0x0001c0	R	0x1	

Format ELF

```
typedef struct {  
    uint32_t p_type;  
    uint32_t p_flags;  
    Elf64_Off p_offset;  
    Elf64_Addr p_vaddr;  
    Elf64_Addr p_paddr;  
    uint64_t p_filesz;  
    uint64_t p_memsz;  
    uint64_t p_align;  
} Elf64_Phdr;
```

PT_NULL

PT_LOAD

PT_DYNAMIC

PT_INTERP

PT_NOTE

PT_SHLIB

PT_PHDR

```
typedef struct {
```

uint32_t sh_name;

uint32_t sh_type;

uint64_t sh_flags;

Elf64_Addr sh_addr;

Elf64_Off sh_offset;

uint64_t sh_size;

uint32_t sh_link;

uint32_t sh_info;

uint64_t sh_addralign;

uint64_t sh_entsize;

```
} Elf64_Shdr;
```

SHT_NULL

SHT_PROGBITS

SHT_SYMTAB

SHT_STRTAB

SHT_REL

SHT_HASH

SHT_DYNAMIC

SHT_NOTE

SHT_NOBITS

SHT_REL

SHT_SHLIB

SHT_DYNSYM

```
typedef struct {  
    uint32_t st_name;  
    unsigned char st_info;  
    unsigned char st_other;  
    uint16_t st_shndx;  
    Elf64_Addr st_value;  
    uint64_t st_size;  
} Elf64_Sym;
```

STT_NOTYPE

STT_OBJECT

STT_FUNC

STT_SECTION

STT_FILE

#include <elf.h>

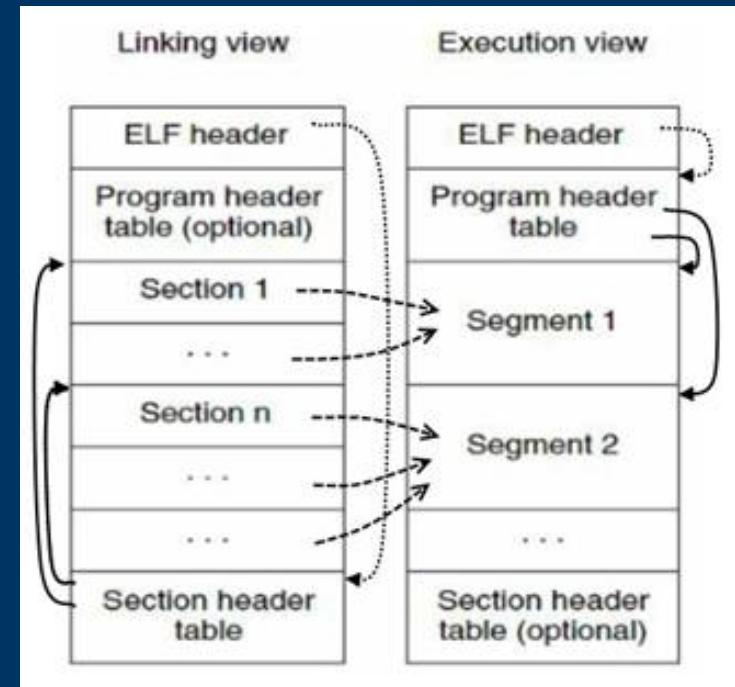
Format ELF

De par sa structure, il est facile de lire le contenu d'un ELF, standard dans l'exécution de programmes sous Linux. Les types sont disponibles nativement.

`dl_iterate_phdr(...)`

On peut charger un programme en le projetant en mémoire (ex: `mmap`).

Par commodité, il existe plusieurs implémentations permettant de manipuler la projection avec des « objets » ELF (ex: `elfutils`)



Gare aux licences.

Format ELF

- **.text** : code exécutable
- **.data / .bss** : données globales
- **.rodata** : constantes
- **.tdata/.tbss** : Section de données thread-specific (TLS)
- **.got** : Table globale permettant d'avoir un accès indirect aux symboles globaux
- **.got.plt** : GOT pour fonctions dynamiques
- **.rel[a].*** : Symbole repositionnable, à résoudre avant le début du programme
- **.init** : prologue
- **.fini** : épilogue
- **.dynamic** : données utiles au loader pour charger les bibliothèques dynamiques
- **.dynstr** : Chaîne de noms des symboles globaux
- **.dynsym** : Table des symboles globaux
- **.syntab** : table de symbole
- **.c/dtors** : Stockage des routines **pre-main()**

Chargement du programme

- En partant du Shell :
 - a. invocation de `execv*` ()
 - b. `search_binary_handler()` : détection du type d'exécutable
 - c. `load_elf_binary()`: mapping du binaire en mémoire (les deux segments `PT_LOAD`)
 - d. Si un interpréteur est nécessaire (section `.interp`), préparation de l'interpréteur pour charger les segments des bibliothèques (`load_elf_interp()`)
 - e. `start_thread()`: la main est enfin donné au code utilisateur
 - f. chargement des bibliothèques dans l'espace mémoire (PIC)
 - g. invocation de la fonction `_start()`
 - h. Invocation de `__libc_start_main()`(`_init` & `_fini`)
 - i. Invocation du `main()`

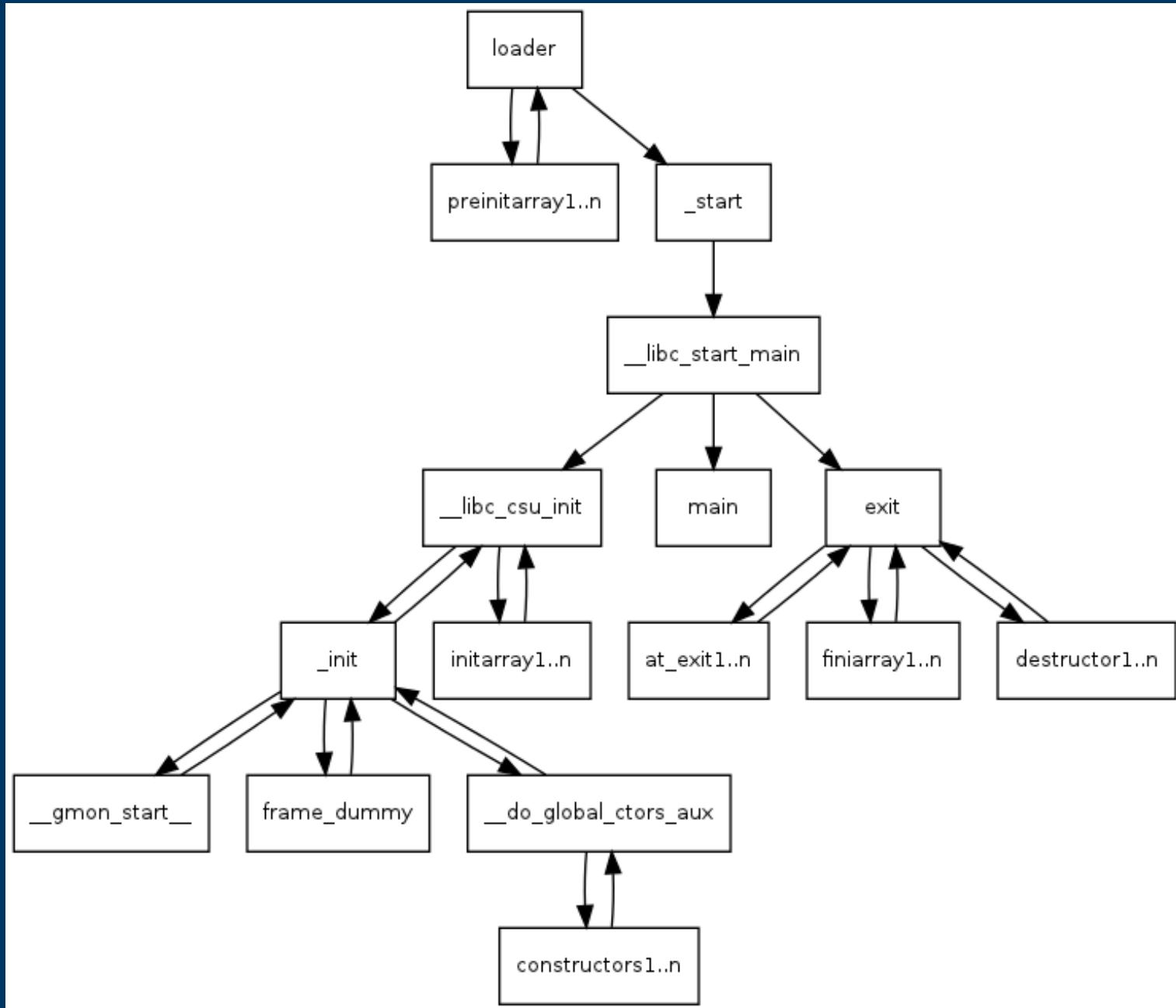
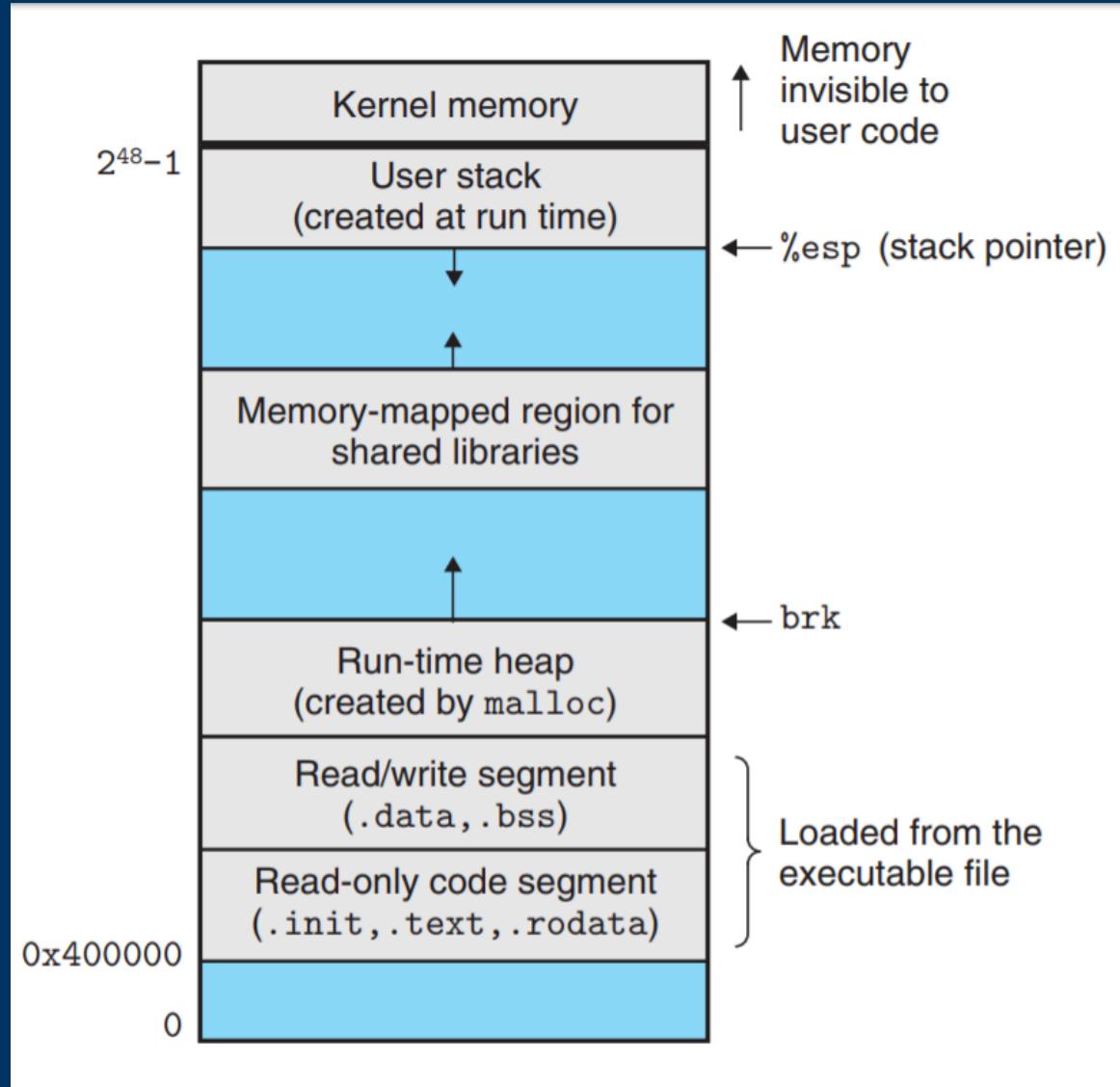


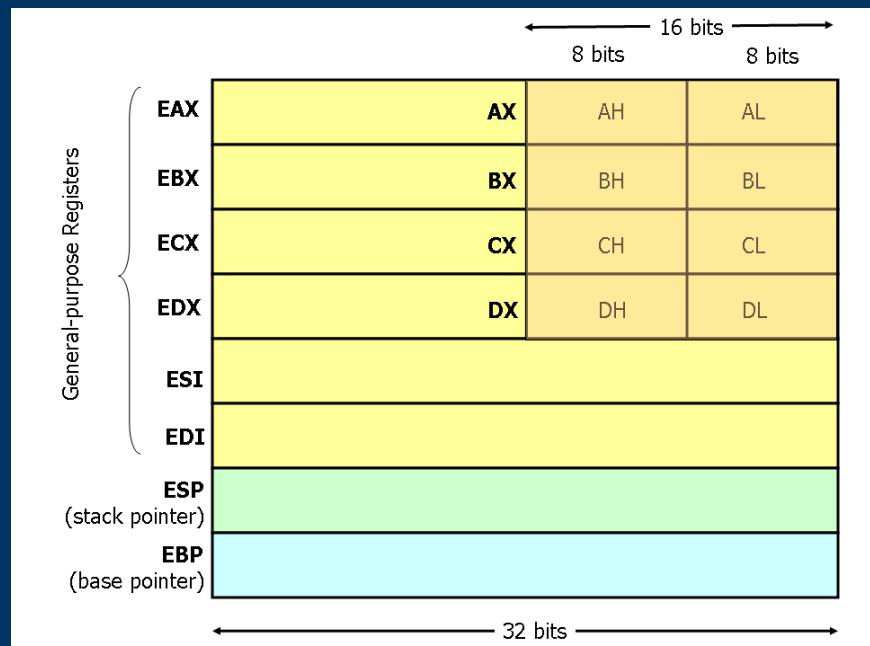
Schéma de la mémoire d'un processus

- 64 bits théoriques
- mais 48 bits câblés ($\Rightarrow 2^{48} = 256$ Tio adressables)
- .text : segment de code
- .data / .bss segments de données (initialisées ou non) = Mémoire **statique**



Les registres

- Plus petite structure décrivant l'information qu'un processeur peut manipuler.
- Chaque registre a un rôle à jouer
 - General-purpose registers
 - State registers
 - Segment registers
- Les registres de segments sont des artefacts historiques de la mémoire à segmentation
 - CS : Code segment
 - DS : Data segment
 - ES : Extra segment
 - SS: Stack segment
 - FS/GS...



Manipulation des Registres:

- **r*** = 64 bits
- **e*** = 32 bits
- **l*** = 16 bits

Le tas (=heap)

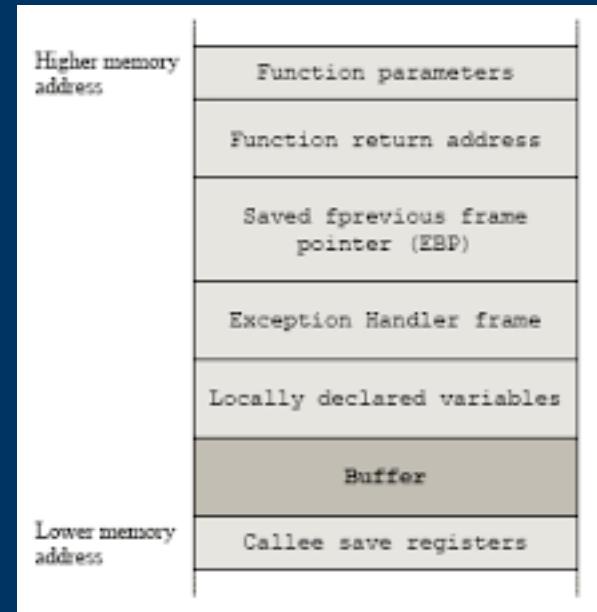
- Mémoire dynamique, persistente
- La gestion de cette mémoire est à la charge de l'utilisateur
- Commence après les segments ELF de données. Ce point précis s'appelle **l'interruption de programme** (*program break*)
- Grossit au fil des allocations au cours de la vie du programme.
- Est manipulé par les fonctions de gestion mémoire standard
 - malloc, calloc, realloc allouent un nouveau bloc de mémoire
 - free libère la mémoire
- **Toujours libérer la mémoire après utilisation (fuite !)**

Legacy (déprécié)

- **brk** : prend une adresse où l'interruption de programme commence
- **sbrk** : prend un incrément (signé) ajouté à l'interruption de programme courante

La pile (=stack)

1. La pile est une superposition de couches appelées « frames », toutes identiques. Une stackframe est créée à chaque fois qu'une nouvelle fonction est appelée (instruction x86 **call***)
2. Dans une stack-frame sont stockés :
 - a. Les arguments de fonctions
 - b. L'adresse de retour **RIP** dans la fonction parente (pour *Return Instruction Pointer*)
 - c. Le pointeur de pile **RBP** de la frame précédente
 - d. Les variables automatiques (dites « locales »)
3. Il existe deux pointeurs de pointeurs de pile
 - a. **RBP** : « Base pointer » = l'adresse où commence la frame courante
 - b. **RSP** : « Stack Pointer » = l'adresse qui suit la dernière adresse accessible pour la frame courante



Manipulation des Registres:

- r* = 64 bits
- e* = 32 bits
- l* = 16 bits

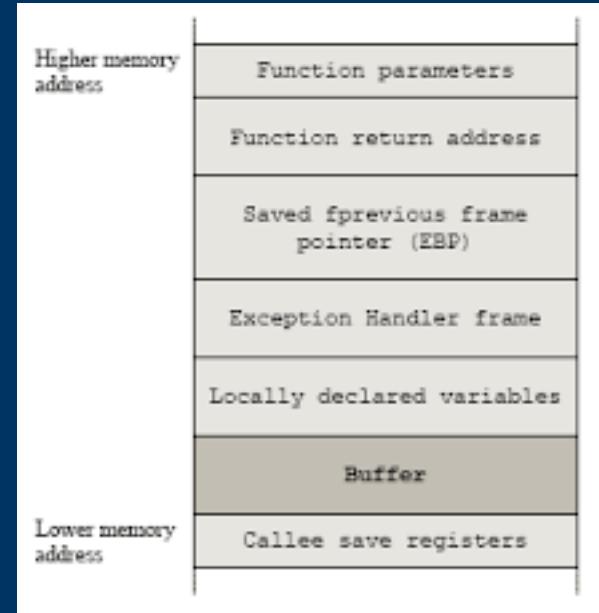
La pile (=stack)

Chaque fonction générée dispose d'un **prologue** destinée à préparer la nouvelle frame courante :

1. Enregistrement du RBP précédent sur la pile
2. Mise à jour de la nouvelle base à partir de l'adresse de la pile
3. Dimension des variables automatiques

Ainsi que d'un **épilogue**, en charge de remettre le contexte de la fonction appelante:

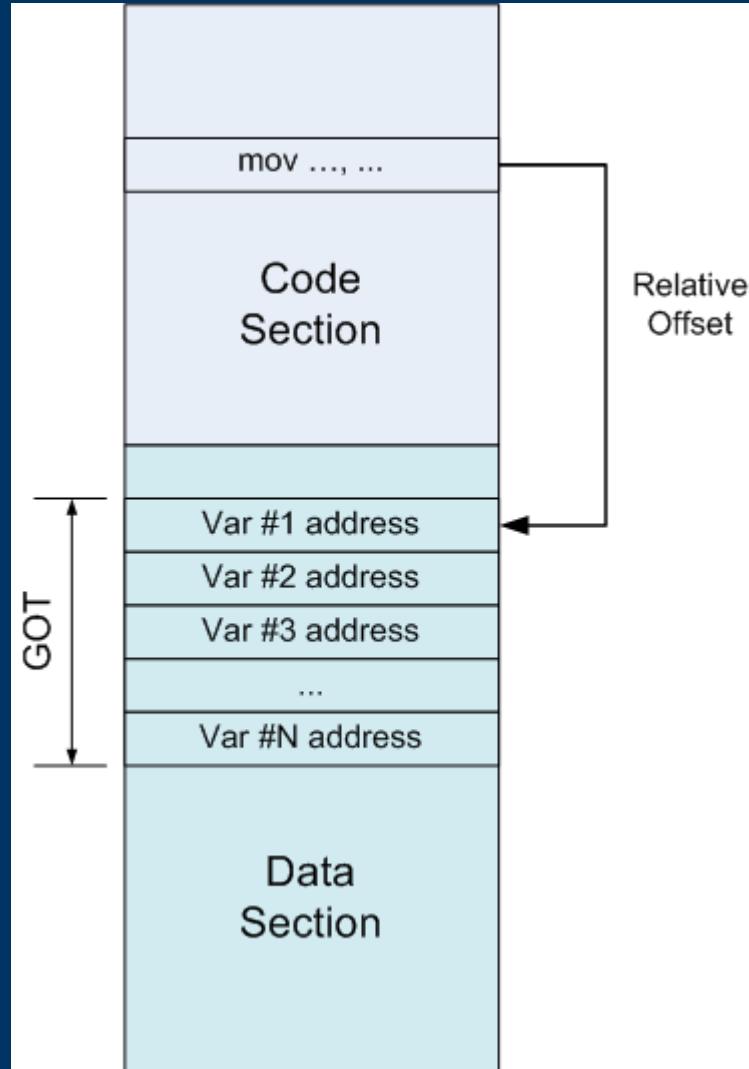
1. Mise à jour de RAX avec le pointeur de retour
2. "Désallocation" des variables automatiques en ramenant RSP à RBP
3. Lecture du RBP précédent sur la pile
4. JMP à l'adresse de retour



```
gdb >> disas main
Dump of assembler code for function main:
0x0000000000040052e <+0>:    push   %rbp
0x0000000000040052f <+1>:    mov    %rsp,%rbp
0x00000000000400532 <+4>:    sub    $0x10,%rsp
0x0000000000000009e3 <+250>:  mov    $0x0,%eax
0x0000000000000009e8 <+255>:  leaveq 
0x0000000000000009e9 <+256>:  retq
```

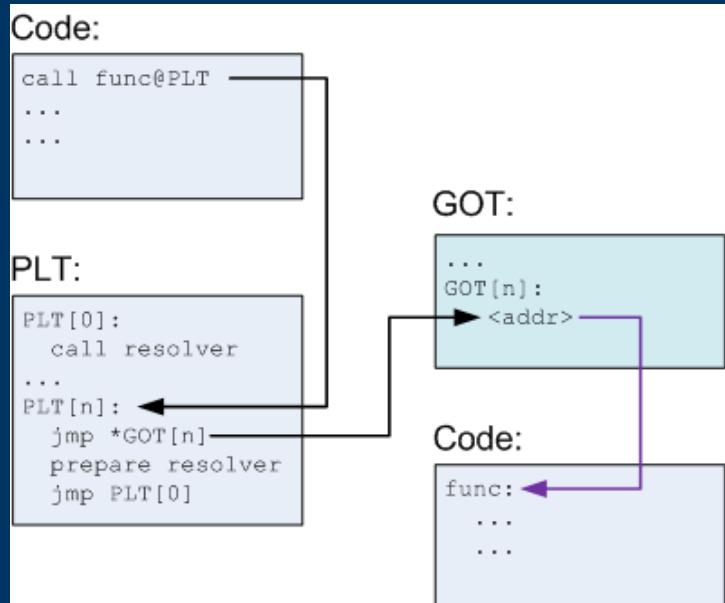
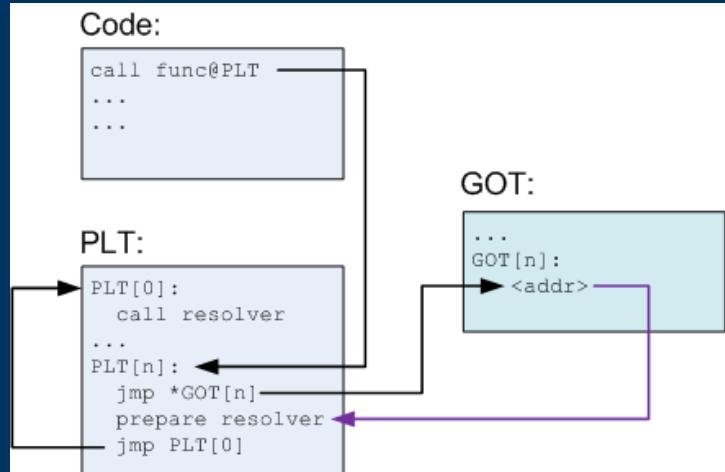
“Relocation” dynamique

- Les codes compilés en “position-independent” (PIC/PIE) sont des codes qui ne possèdent pas de lien mémoire “en dur”. Il faut donc les résoudre pour le programme fonctionne
- Les bibliothèques, qui peuvent être placés n’importe où dans l’espace mémoire, sont toujours en -fPIC.
- Mais quand faire cette “relocation” ?
 - à la compilation
 - au chargement du programme
 - à l’exécution
- GOT = Global Offset Table
- => Indexation de toutes les variables non déterministes



“Relocation” dynamique de fonctions

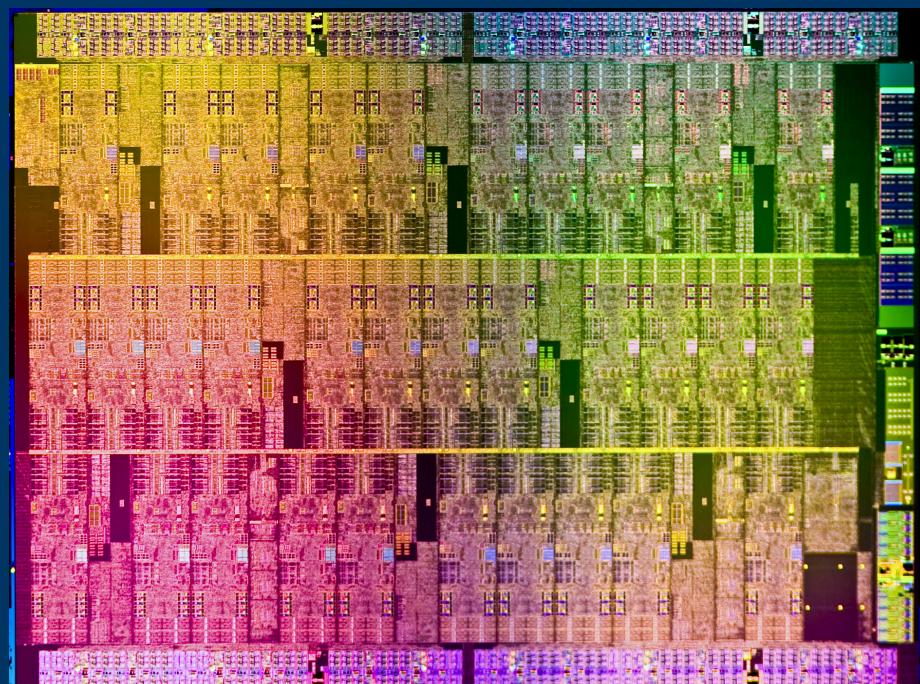
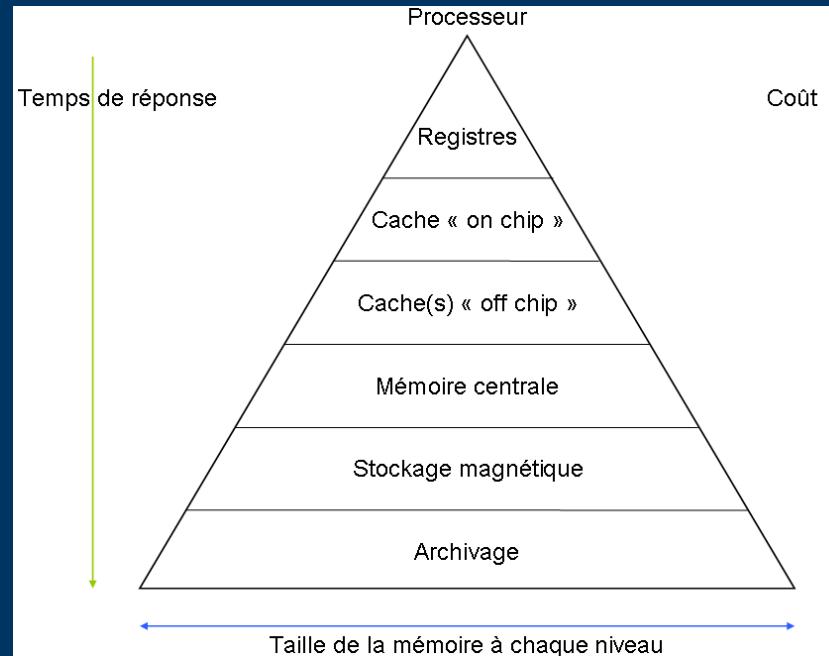
- Comment gérer ce même comportement pour les fonctions ?
- Résolution de toutes les fonctions au chargement est peu efficace (latence)
- => ***Lazy binding*** !
- La PLT ajoute un niveau d'indirection supplémentaire à une GOT dédié
- Au 1er appel, la PLT invoque une fonction du loader (lookup).
- L'adresse est mise à jour dans la GOT
- Les futurs appels n'auront qu'un JMP
- Forcer une résolution complète avec `LD_BIND_NOW`



Gestion mémoire

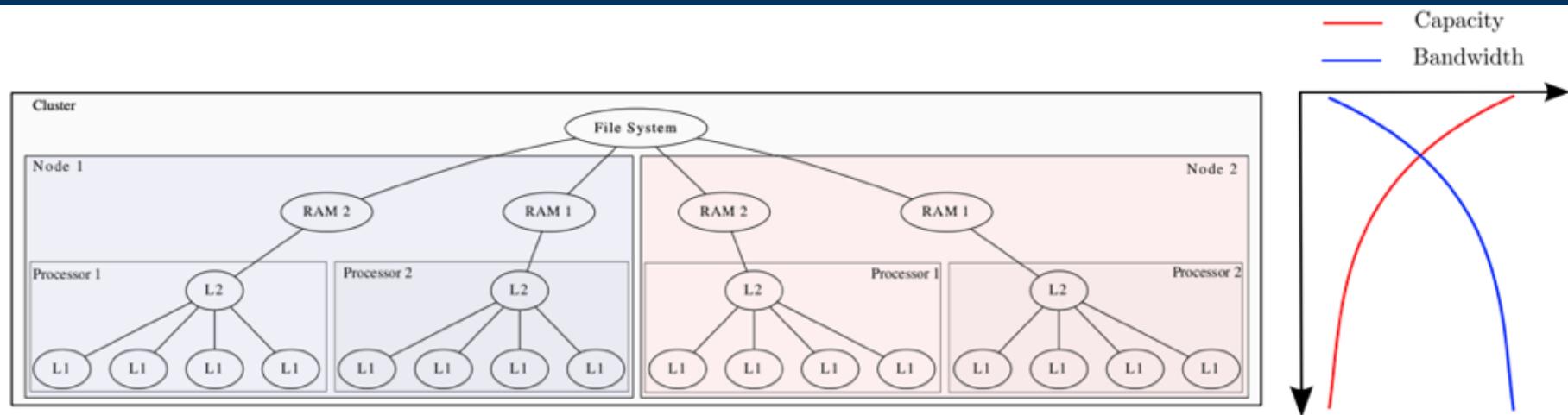
Histoire de la mémoire

- Mémoire = Système de stockage de l'information
- Plus ce stockage est proche de l'unité de calcul, plus elle est chère (cout de câblage) => on diminue donc sa quantité
- Différents types de mémoire:
 - **Mémoire de masse** (grande quantité, Tio, Pio...)
 - **Mémoire « vive »** (quantité raisonnable d'exécution (Mio, Gio))
 - **Mémoire « cache »** (stockage à court terme, Kio, Mio)
 - **Registres** (manipulation de données (de l'ordre de l'octet))



Histoire de la mémoire

Plus la mémoire est à faible latence (proche des coeurs) plus elle est petite pour des raisons de coût en surface sur le chipset.



Histoire de la mémoire volatile

- aux débuts de l'informatique, la mémoire était très chère et donc peu présente
 - Au maximum 2000 mots de 16 bits -> 4 Kio !!
 - Favorisation des codes *lents* (car moins gourmand)
 - Utilisation des *overlays* (=branches)
 - écrasement par couche successive d'une mémoire secondaire
 - début des années 1970 => arrivée de la mémoire virtuelle
 - Notion d'espace d'adressage (ensemble des mots adressables)
-
- Les programmes considèrent toujours que la mémoire est suffisante
 - Aucune contrainte sur l'existence (ou non) d'un mot en machine
 - Levée d'une contrainte jusque là imposée au programmeur
 - => Comment créer une mémoire “infinie” sur des ressources finies ?

Contexte d'un programme

- Manipulation d'adresses “virtuelles” par le processus
- Isolation dans un espace d'adressage “infini”
- => Nécessite d'être traduite en adresses réelles, de DRAM



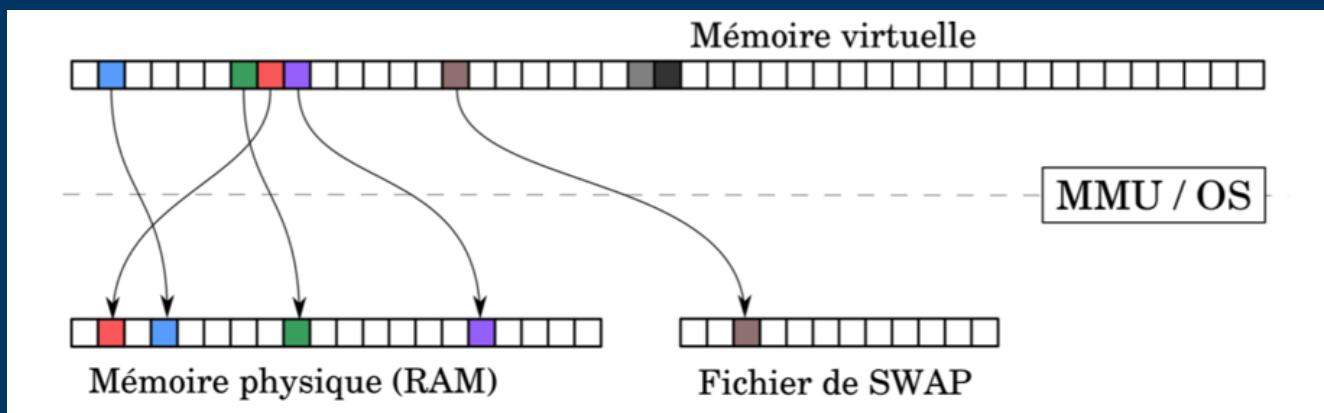
Composant dédié : la MMU (Memory Management Unit), composant noyau, généralement par application, qui s'occupe de gérer la traduction virtuelle <-> physique.

=> Offrir une mémoire linéaire et continu à l'application, quel que soit les caractéristiques et/ou contraintes matérielles.

Un espace d'adressage est construit sur l'ensemble des mots adressables sur une dimension donnée (ex: 64 bits -> 2^{64} mots codables -> 8 EB !)

Pagination

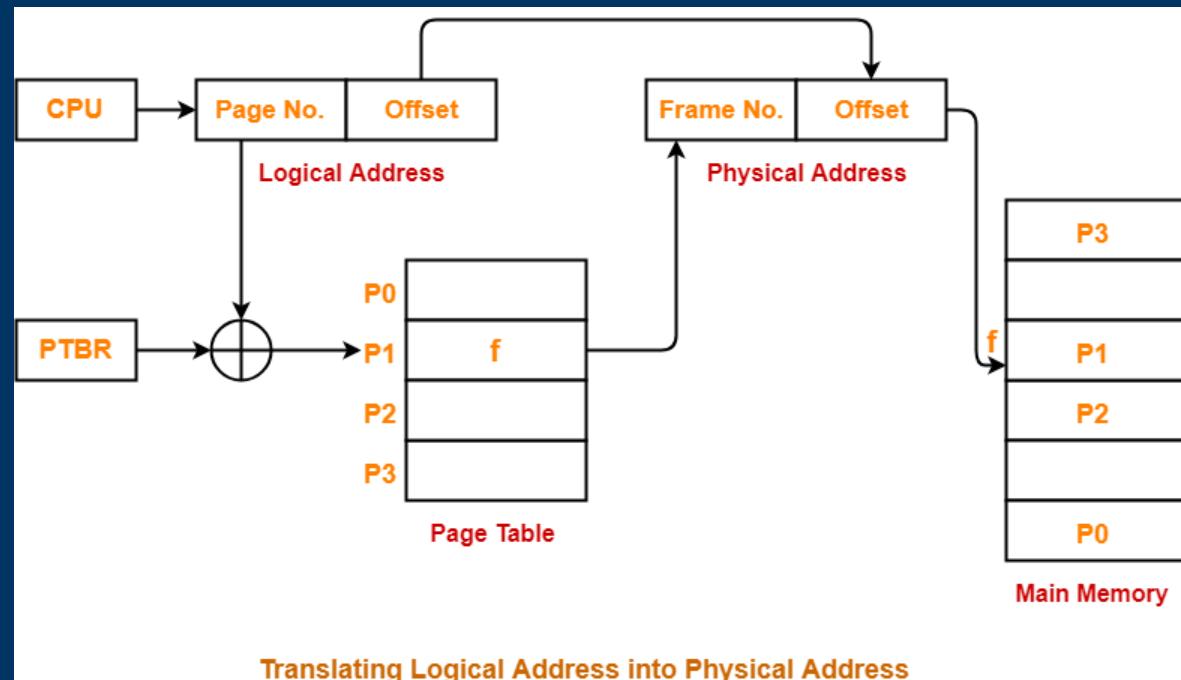
- => Nécessité d'un mapping virtuel <-> physique
- Découpage en blocs de taille définie. Pour être accessible à l'application, une bloc doit être chargée en mémoire centrale (DRAM).
- 1 bloc en mémoire physique = 1 cadre
- 1 bloc en mémoire virtuelle = 1 page
- 1 cadre = 1 page
- la MMU s'assure de fournir, pour chaque page utilisée dans l'application, un cadre valide (une projection)



Pagination

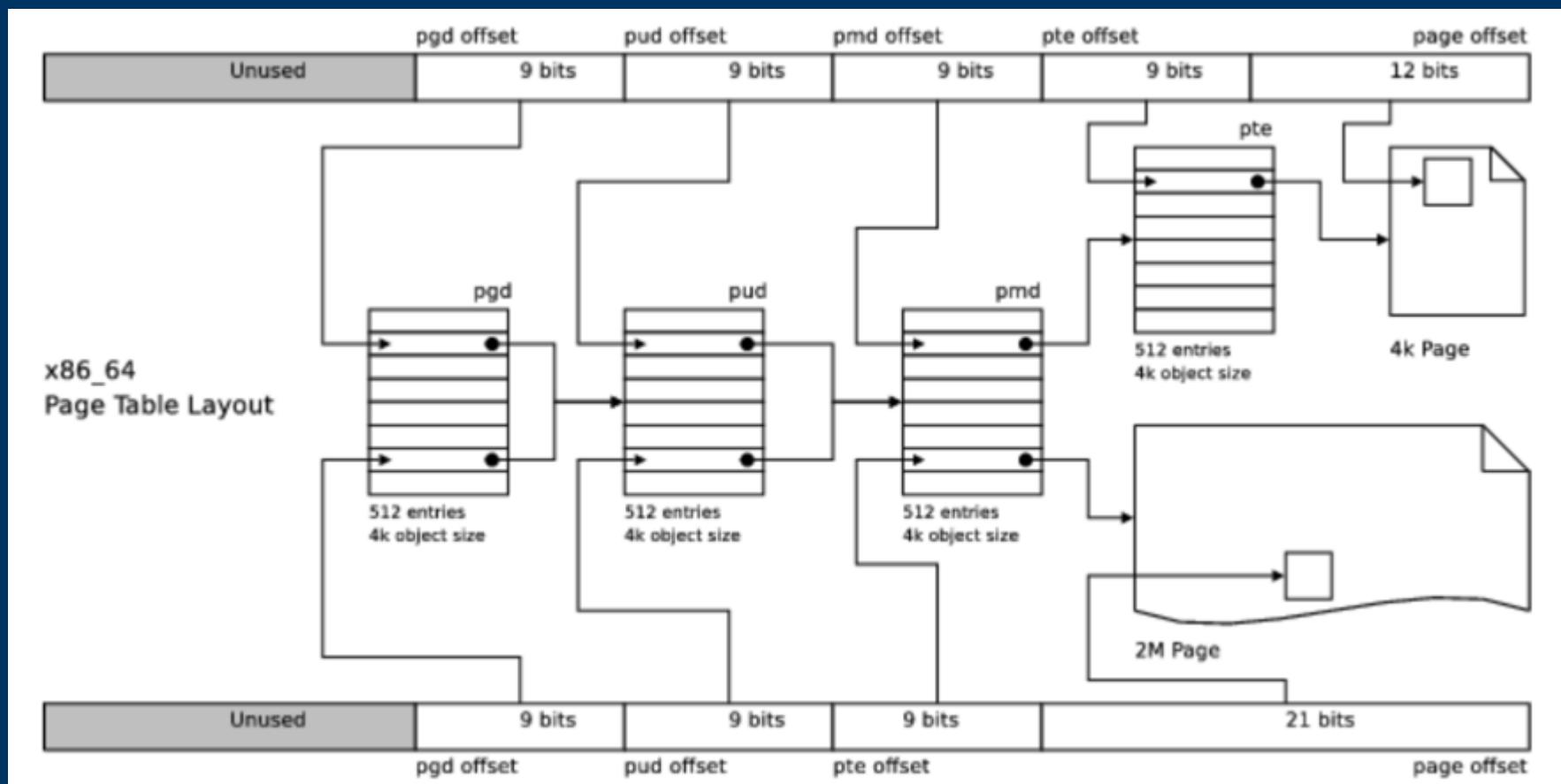
- => Nécessité d'indexer les correspondances pages <-> cadre
- Table des pages, stockées dans la mémoire noyau du processus)
- L'adresse est découpée, pour créer un ID de page et un offset
- La table référence, pour chaque ID de page, l'ID de cadre associé
- Couplé à l'offset il permet d'avoir l'adresse physique accédée
- Support de champs (validité,...)

- Une grande table implique une grande consommation mémoire
 - Si 1 entrée (PTE) = 4 octets
 - => Table de 256 GB !!!
-
- Mais l'utilisation mémoire est rarement uniforme et des entrées de tables sont inutilisées
 - Solution ?



Pagination

- Fractionnement de la table en niveau
- Gain mémoire important
- MAIS latence supplémentaire



Pagination à la demande

La mémoire virtuelle excède toujours la mémoire réelle (centrale). Lorsqu'une référence est faite à une page qui n'existe pas en mémoire réelle, il se produit un **défaut de page** (*page fault*).

Une interruption noyau est générée, déclenchant le chargement de cette page en mémoire. C'est ce qu'on appelle la **pagination à la demande**. Les pages ne sont chargées qu'en cas de besoin et non à l'avance.

Un défaut de page implique le noyau, la MMU et plusieurs accès mémoire (mise à jour des tables), ce qui implique **un coût plus important**.

Le défaut de page le plus connu est lors du premier accès à une page nouvellement allouée (first-touch) qui définit aussi l'affinité mémoire.

Remplacement de pages

Il peut arriver qu'il n'y ait plus de cadres disponibles en mémoire physique. Dans le cas de surcharge (d'un ou plusieurs processus/ utilisateurs), il est nécessaire de supprimer une page en DRAM et être envoyé sur le disque (=swapping).

Plusieurs algorithmes:

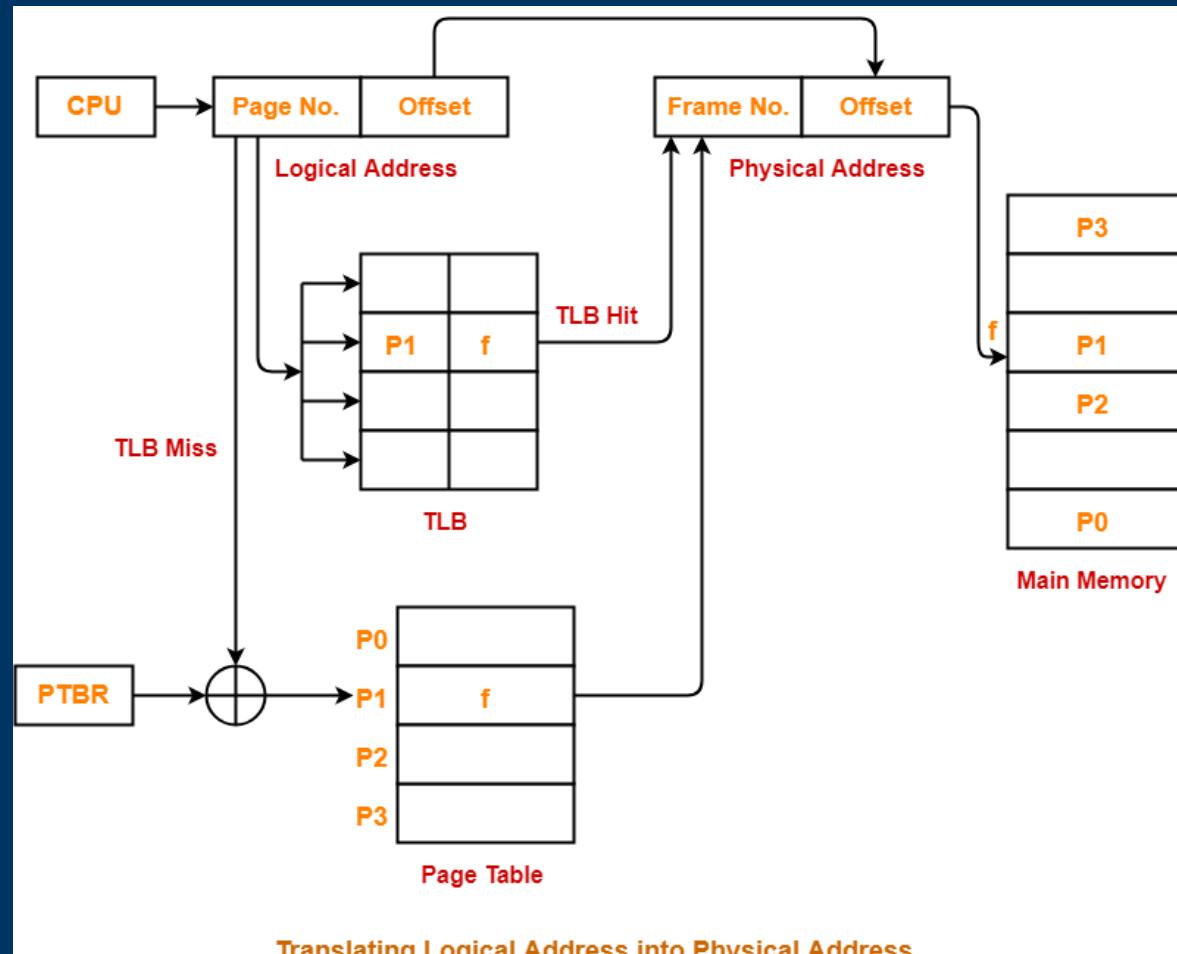
- **LRU**: La plus vieille page utilisée (coûteux)
- **FIFO**: Première mappée, première sortie
- **Deuxième chance**: 1 bit d'utilisation est associée à chaque page et passé à “1” à chaque utilisation. Au moment du remplacement, une page marquée dispose d'une “seconde chance”
- Remplacement **optimal** (connaissance *a priori*)

Quelle taille de page ?

- Fragmentation interne : une page n'est pas intégralement utilisée par l'application
 - => minimisation du coût = plus petites pages
 - ==> Plus de pages ! consommation mémoire !!
 - ==> Sous exploitation de la bande passante
-
- Une indexation demande d'accéder à la mémoire
 - On paye donc un coût multiple.
 - Solution ?

Translation Lookaside Buffer (TLB)

- Cache du processeur pour les tuples (page,cadre) les plus utilisés
- Crée une indirection supplémentaire qui doit pouvoir être mis à jour/ invalidé lorsque la MMU met à jour la table des pages
- taille figée par le matériel



L'allocateur mémoire

L'allocateur mémoire

Un processus manipule trois types de variables:

- statique : variables globales qui ont un segment dédié
- automatique : variables locales propres à chaque fonction
- dynamique: géré par l'application, qui ont une portée explicite

Cette dernière catégorie est manipulée par malloc/free. Ces fonctions ont une manipulation fine de la mémoire (à l'octet) tandis que nous avons vu que la mémoire se gérait par **pages**.

La page crée donc la projection de la DRAM dans l'espace d'adressage, par bloc de 4K. La charge de l'allocateur est d'optimiser les besoins de l'application en mémoire au travers d'une politique adaptée

mmap / munmap

Associe à une page de la mémoire virtuelle, une projection réelle.

Peut projeter un fichier, un device ou de la mémoire
“pure” (MAP_ANONYMOUS)

- **prot** détermine la protection de la page (R, W, E)
- **flags** détermine la visibilité de la page (PRIVATE, SHARED, ANON, FILE...)

Modification des protections sur une page existante : mprotect()

Aider le noyau sur la portée d'une page : madvise()

NAME
`mmap, munmap - map or unmap files or devices into memory`

SYNOPSIS
`#include <sys/mman.h>`

`void *mmap(void *addr, size_t length, int prot, int flags,
 int fd, off_t offset);
int munmap(void *addr, size_t length);`

See NOTES for information on feature test macro requirements.

DESCRIPTION
`mmap()` creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in `addr`. The `length` argument specifies the length of the mapping.

If `addr` is NULL, then the kernel chooses the address at which to create the mapping; this is the most portable method of creating a new mapping. If `addr` is not NULL, then the kernel takes it as a hint about where to place the mapping; on Linux, the mapping will be created at a nearby page boundary. The address of the new mapping is returned as the result of the call.

The contents of a file mapping (as opposed to an anonymous mapping; see `MAP_ANONYMOUS` below), are initialized using `length` bytes starting at offset `offset` in the file (or other object) referred to by the file descriptor `fd`. `offset` must be a multiple of the page size as returned by `sysconf(SC_PAGE_SIZE)`.

The `prot` argument describes the desired memory protection of the mapping (and must not conflict with the open mode of the file). It is either `PROT_NONE` or the bitwise OR of one or more of the following flags:

`PROT_EXEC` Pages may be executed.
`PROT_READ` Pages may be read.
`PROT_WRITE` Pages may be written.
`PROT_NONE` Pages may not be accessed.

```
mmap(NULL, s, PROT_WRITE, MAP_FILE | MAP_SHARED, fd, 0);
mprotect(addr, s, PROT_WRITE);
madvise(addr, s, MADV_DONTNEED);
```

Exemple MMAP

```
#include <stdio.h>
#include <sys/mman.h>

#define SIZE 1024*4096

int main(){
    char * v = mmap(NULL, SIZE,
                    PROT_READ | PROT_WRITE,
                    MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

    if( v == MAP_FAILED){
        perror("mmap");
        return 1;
    }

    size_t i;

    for( i = 0 ; i < SIZE; i++ )
    {
        v[i] = i;
    }

    munmap(v, SIZE);

    return 0;
}
```

L'allocateur mémoire

- Gestion sur deux fronts :
 - bas niveau : en gérant les pages allouées pour le processus en cours (mmap/munmap)
 - haut niveau : en implémentant les fonctions type malloc/free pour répartir ses allocations dynamiques sur les pages allouées
- Plusieurs problématiques:
 - Faire un allocateur basique est simple, avec une bijection :
 - malloc = mmap
 - free = munmap
 - Mais on est très loin de la performance
 - mmap/munmap sont des appels systèmes coûteux, l'objectif est de les minimiser
 - La gestion des blocs est aussi un grand challenge, notamment les blocs libres.
 - Deux solutions:
 - Indexation centrale (liste(s) globale(s))
 - Distribution d'une partie de l'information sur chaque bloc (*piggyback*)
 - **Alignement mémoire indispensable !**

Politiques d'allocation

Un point critique va être de déterminer comment l'allocateur choisit l'espace mémoire projetée (=mappée) qui conviendra selon l'appel à malloc(N)

- **First** fit: Le premier espace disponible > N est choisi (fragmentation !)
- **Next** fit: Extension de la solution précédente, où la recherche d'un bloc libre commence là où le précédent a été trouvé (avec modulo)
- **Best** fit: L'espace qui convient le *mieux* (minimisation de l'espace restant)

Mais, consommer simplement tout le bloc est souvent inefficace:

- **fragmentation** : Diviser un bloc trop grand pour pouvoir récupérer l'espace disponible
- **Fusion** : Agréger deux blocs libres successifs en un seul

Cas Particulier des Strings

Petit Exemple

```
#include <stdio.h>

int main( ) {
    char * v = "toto";
    v[0] = 'l';
    printf( "v = %s\n", v );

    return 0;
}
```

Que fait ce code ?

Petit Exemple

```
$ gcc ./t.c && ./a.out  
Erreur de segmentation
```

Pourquoi ?

Petit Exemple

```
$ gcc ./t.c && ./a.out  
Erreur de segmentation
```

Car tout string en C est une constante lorsque définit dans le code.

Copier un String

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

char * sdup(char *src)
{
    int len = strlen(src);
    char *ret = malloc(len);
    memcpy(ret, src, len);
    return ret;
}

int main()
{
    char * v = sdup("toto");

    v[0] = 'l';
    printf("v = %s\n", v);

    free(v);

    return 0;
}
```

Ce code est-il correct ?

Copier un String

```
$ gcc lotocp.c && ./a.out  
v = loto
```

**Cependant le code est
faux.**

Copier un String

```
char * sdup(char *src)
{
    int len = strlen(src);
    char *ret = malloc(len+1);
    memcpy(ret, src, len+1);
    return ret;
}
```

\$ gcc lotocp_memset.c && ./a.out
v = loto

BIEN

Copier un String

```
int main(){
    char * v = strdup("toto");
    v[0] = 'l';
    printf("v = %s\n", v);
    free(v);
    return 0;
}
```

\$ gcc lotocp_memset.c && ./a.out
v = loto

MIEUX

Calloc vs Malloc + memset

Comparaison des Allocations à 0

