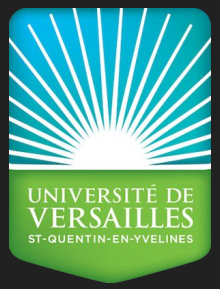
Structure mémoire d'un programme

M1 - CHPS

Architecture Interne des Systèmes d'exploitations (AISE)

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



Julien Adam
<julien.adam@paratools.com>

Programme du Semestre

- 1 - Généralités sur les OS et Utilisation de base
- 2 - Processus, Threads & Synchronisation
- 3 - Compilation et représentation Binaire
- **4 - Architecture Mémoire d'un processus**
- **5 - Programmation réseau et entrées/sorties avancées**
- **6 - Virtualisation et Conteneurs**
- **7 - Noyau Linux et bases d'ordonnancement**
- **Examen + Démo de projets**

Histoire de la mémoire

- 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

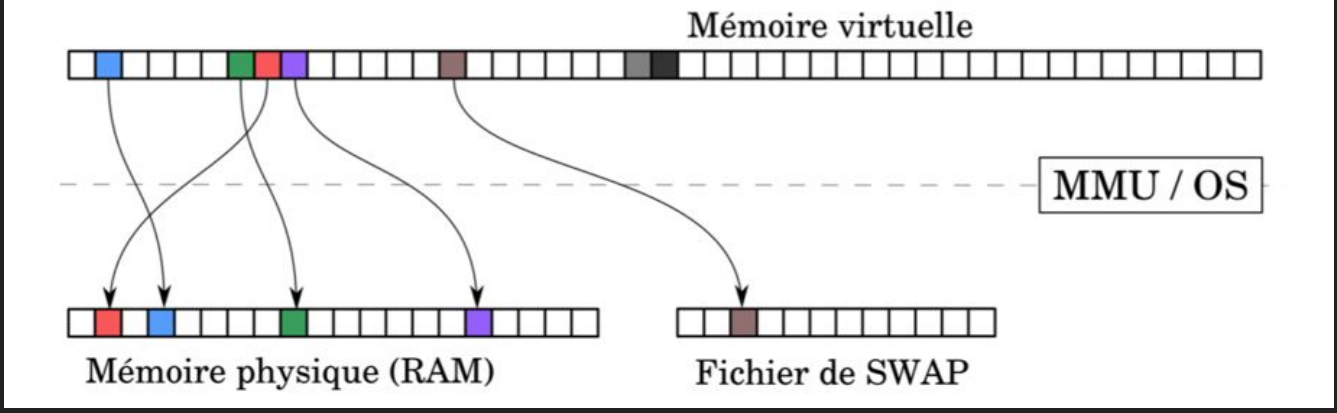
Composant dédié : la MMU (Memory Management Unit), composant noyau, généralement par application, qui s’occupe de gérer la traduction virtuelle <-> physique.

=> 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)

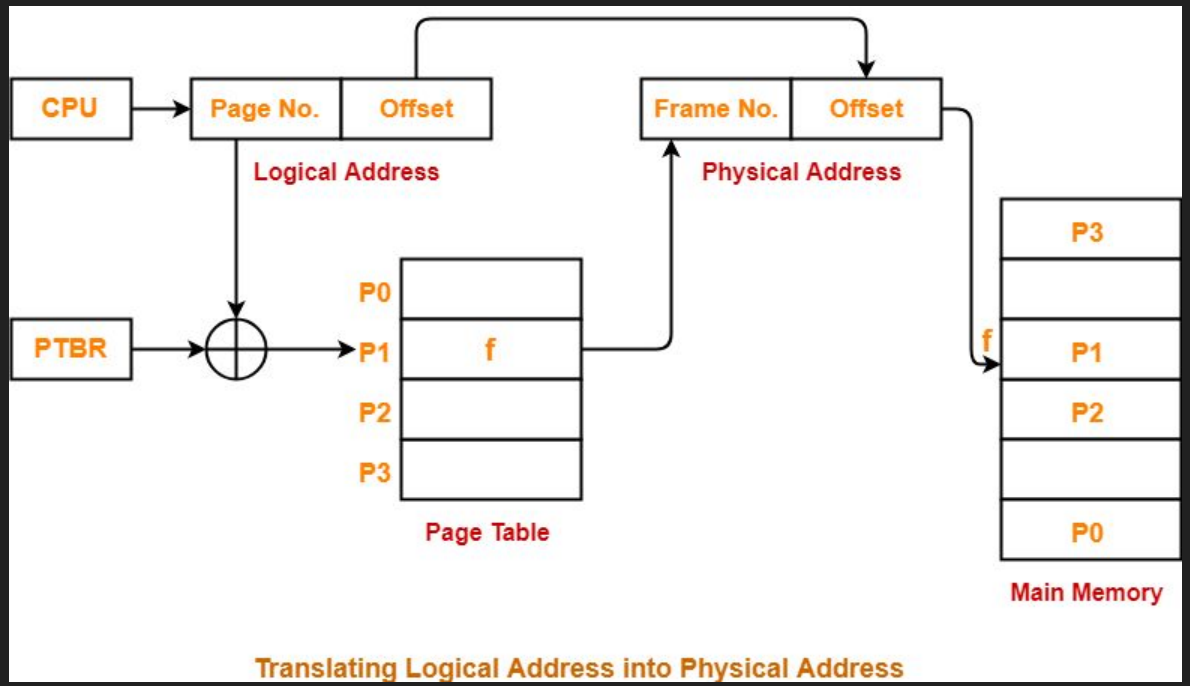


Pagination

- => 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 !!!

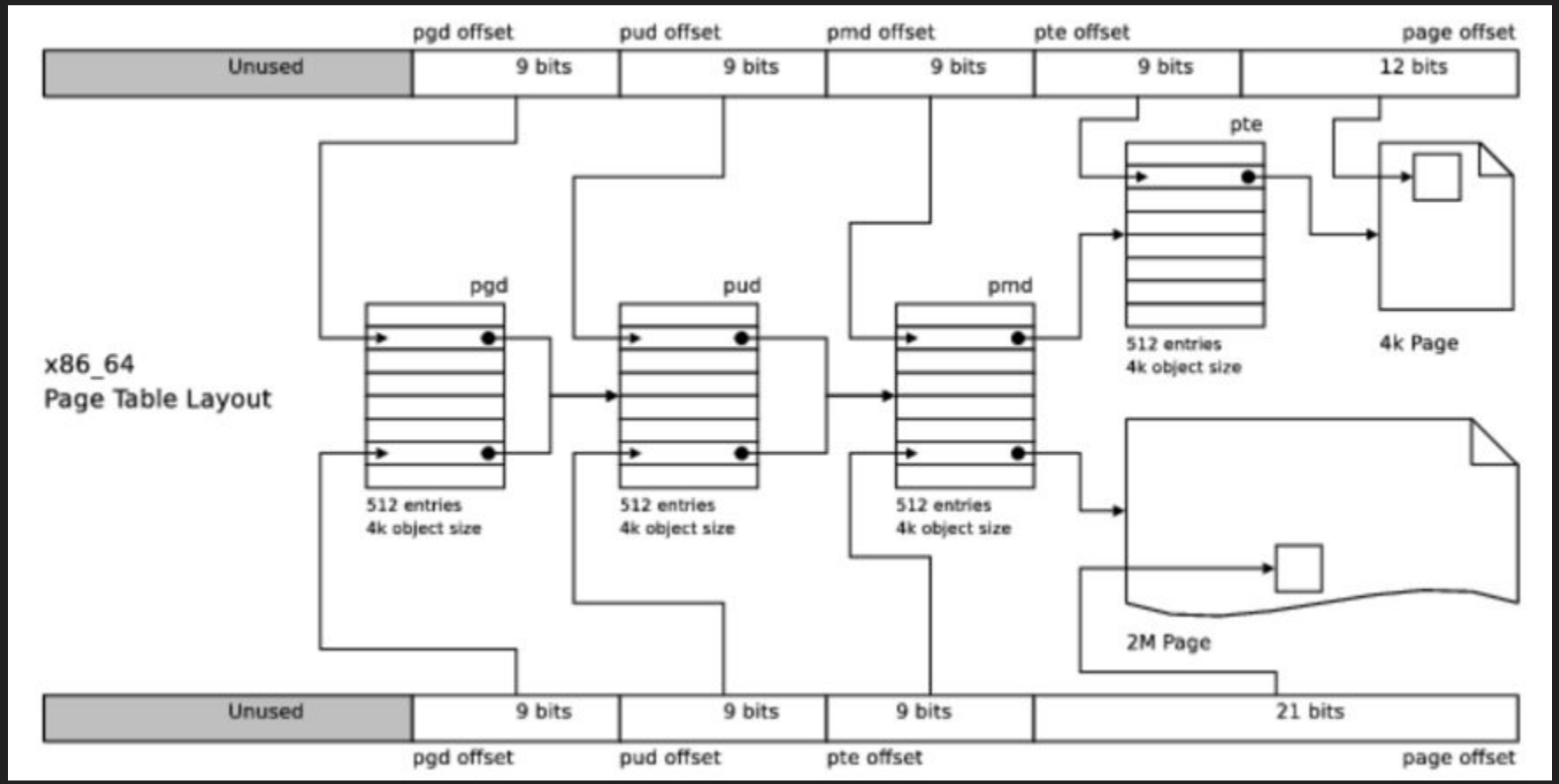
- Mais l'utilisation mémoire est rarement uniforme et des entrées de tables sont inutilisées
- Solution ?



Translating Logical Address into Physical Address

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 généré, déclenchant le chargement de cette page en mémoire. C'est ce qu'on appelle la **pagination à la demande**. Les pages ne sont chargés 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*)

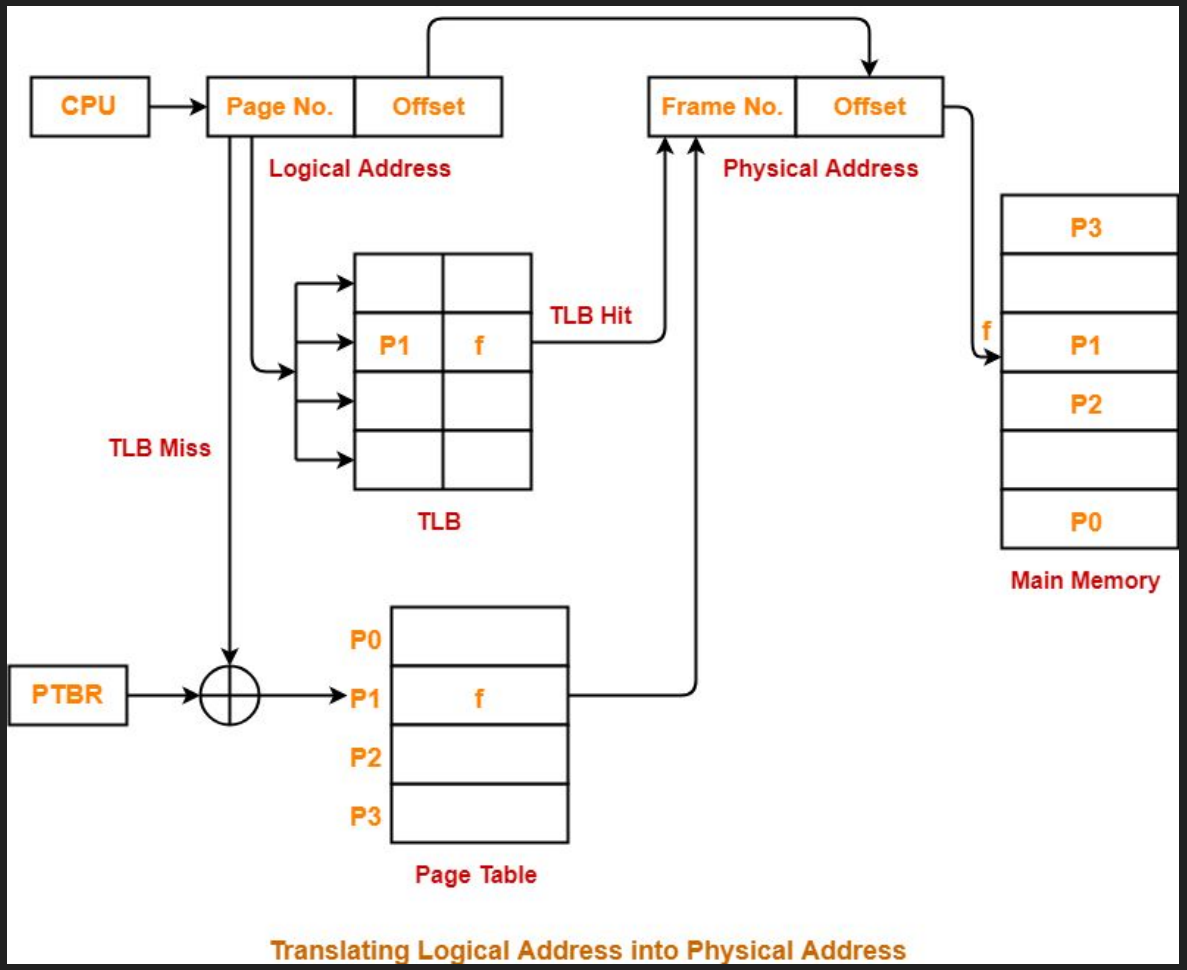
Quelle taille de page ?

- 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

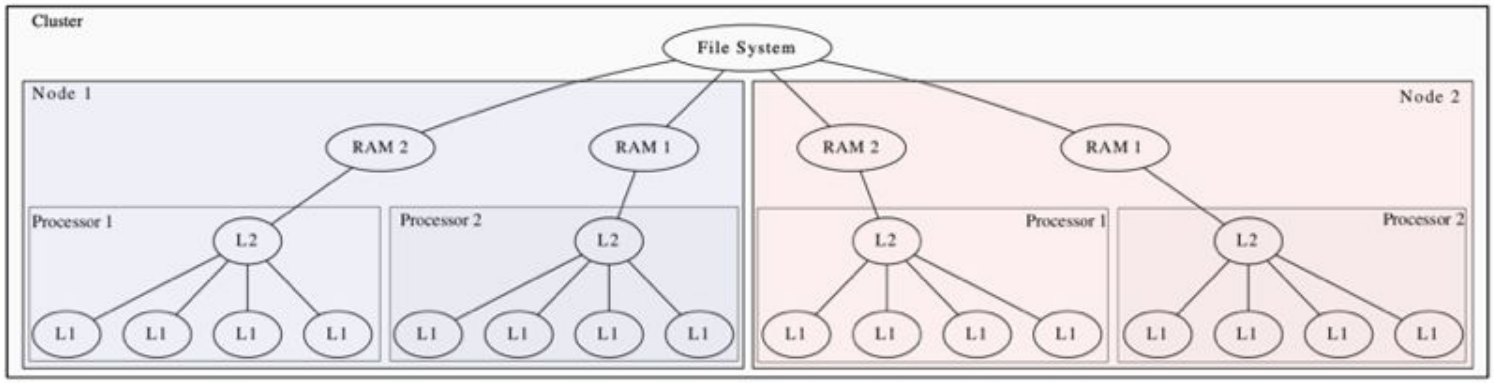
- Une indexation demande d’accéder à la mémoire
- On paye donc un coût multiple.
- Solution ?

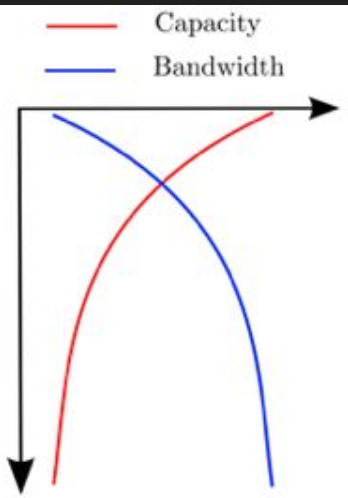
Translation Lookaside Buffer (TLB)

- 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



Hiérarchie mémoire

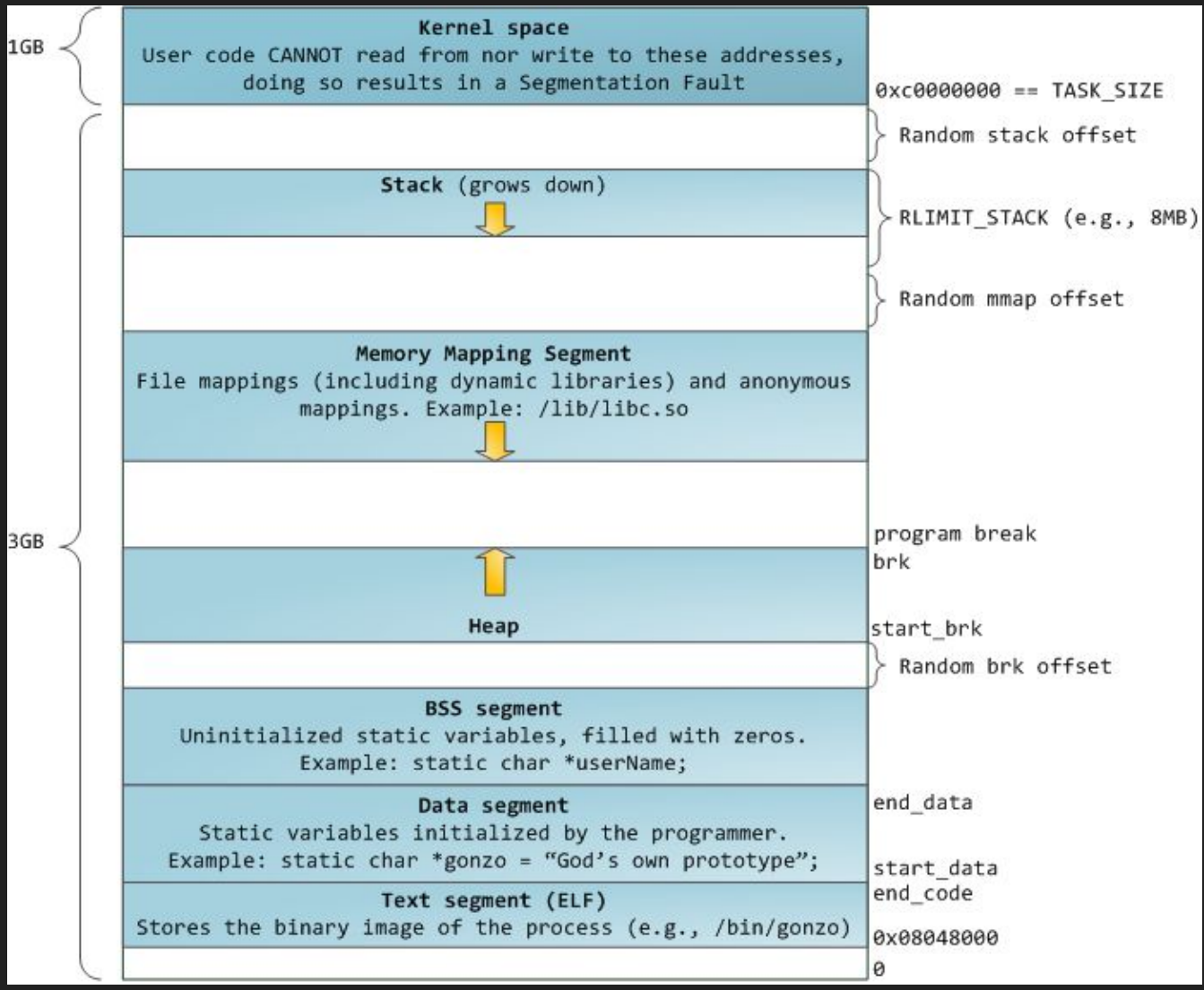




Plus la mémoire est à faible latence (proche des coeurs) plus elles est petite pour des raisons de coût en surface sur le chipset.

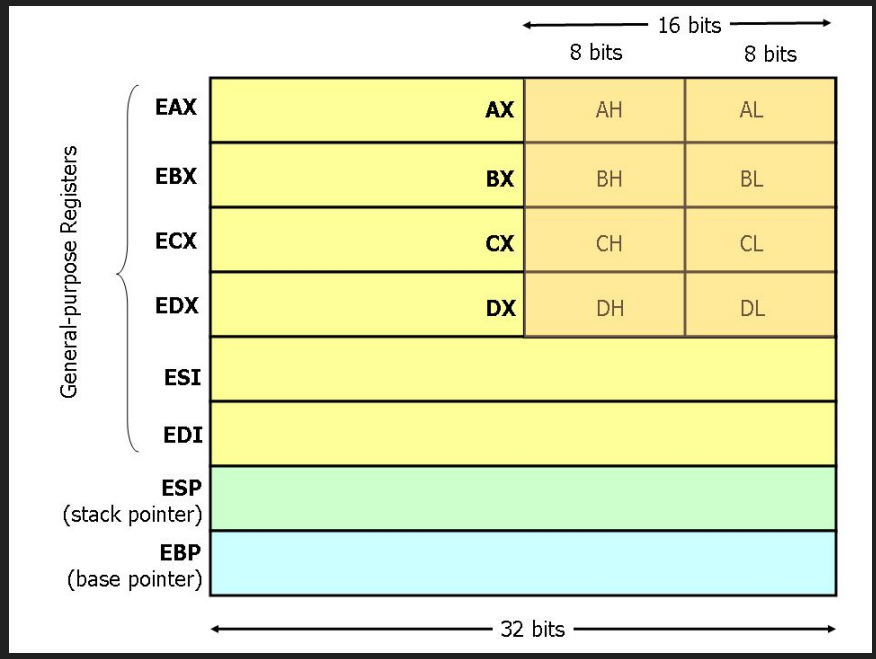
Schéma de la mémoire d'un processus

- 64 bits théoriques
- mais 48 bits câblés (soit 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 "raccourcis" vers leurs blocs associés:
  - CS : Code segment
  - DS : Data segment
  - ES : Extra segment
  - SS: Stack segment
  - FS/GS...



Manipulation des Registres:
- r* = 64 bits
- e* = 32 bits
- 1* = 16 bits

Le tas (=heap)

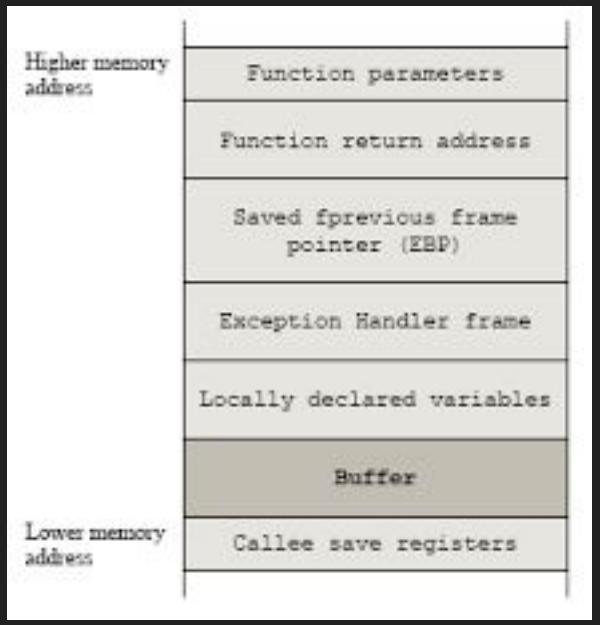
- Mémoire dynamique, persistente
- La gestion de cette mémoire est à la charge de l'utilisateur
- Commence après les segments ELF de données. Ce point précis s'appelle **l'interruption de programme** (*program break*)
- Grossit au fil des allocations au cours de la vie du programme.
- Est manipulé par 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

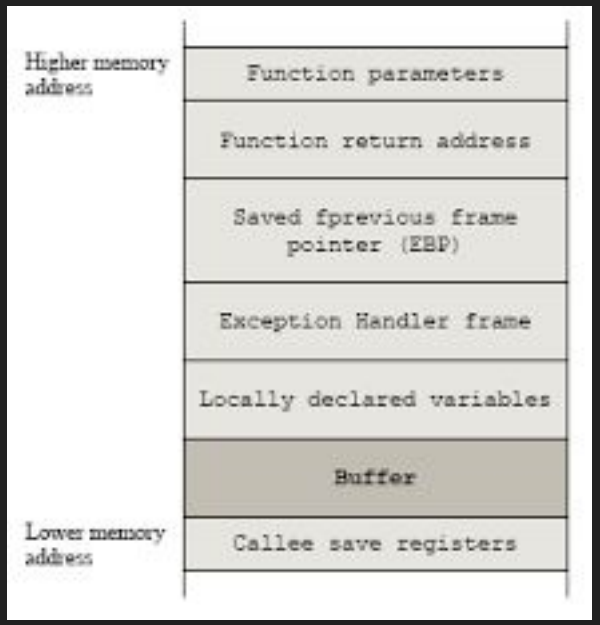
La pile (=stack)

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
   0x0000000000400532 <+4>:     sub    $0x10,%rsp
```

```
 0x00000000000009e3 <+250>:   mov    $0x0,%eax
 0x00000000000009e8 <+255>:   leaveq
 0x00000000000009e9 <+256>:   retq
```

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  argu-
       ment specifies the length of the mapping.

       If  addr is NULL, then the kernel chooses the address at which to create the map-
       ping; 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  off-
       set  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);
```

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