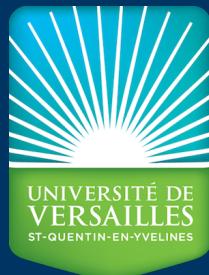


Compilation, bibliothèques & layout mémoire

M1 - CHPS

Architecture Interne des Systèmes d'exploitations (AISE)

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



Julien Adam
<julien.adam@paratools.com>

Organisation

- Chaque session est découpée en deux parties. Un cours théorique le matin et une mise en pratique l'après-midi (TD) portant sur les connaissances vues le matin.
- Des QCMs sur les bases **importantes** au fil des semaines et portant sur un cours précédent. Le QCM aura toujours lieu durant la matinée
- Un Projet, date de rendu au **18/12/2023 Minuit**
- Un Examen final le **22/12/2023 (APM)**

- 1. Généralités sur les OS et Entrées-Sortie**
- 2. Compilation, Bibliothèques et Layout Mémoire**
- 3. Mémoire partie 2, Layout Binaire, Runtime**
- 4. Virtualisation et Conteneurs**
- 5. Programmation réseau et entrées/sorties avancées**
- 6. Programmation Noyau**
- 7. Scheduling et Temps-Réel**
- 8. Examen Ecrit et Démos**

Type d'Examen	Coefficient
QCMs	10 %
Projet	40 %
EXAMEN	50 %

Cours et Corrections



https://github.com/besnardjb/AISE_24

Programme

1. ~~Généralités sur les OS et Entrées-Sortie~~
2. **Compilation, Bibliothèques et Layout Mémoire**
3. Mémoire partie 2, Layout Binaire, Runtime
4. Virtualisation et Conteneurs
5. Programmation réseau et entrées/sorties avancées
6. Programmation Noyau
7. Scheduling et Temps-Réel
8. Examen Ecrit et Démos

Construction d'un programme

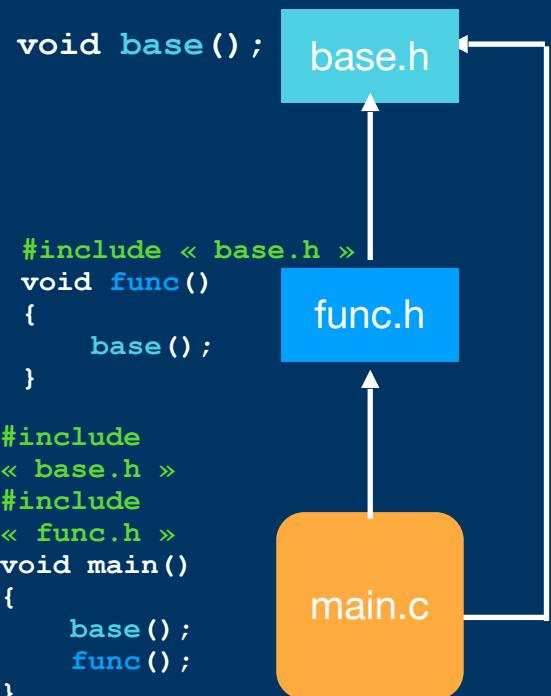
- Unité de compilation : un fichier source (parfois nommé TU pour « Translation Unit »)
- Préprocesseur : préparation d'une TU pour la compilation.
- Compilateur : exécution d'une TU, transformation en un set d'instructions spécifiques à l'architecture.
- Assemblage: Transformation des instructions assembleur en code machine (structure ELF)
- L'édition de liens: Assemblage des différentes TU pour créer un exécutable

Derrière GCC

- GCC signifie “GNU Compiler **Collection**”, ce n'est donc pas un programme, mais une collection d'outils, un pour chaque phase de la construction d'un programme:
- preprocessing: `cpp file.c > file.i`
- compilation: `cc1 file.i -o file.s`
- Assemblage: `as file.s -o file.o`
- Link: `collect2 file.o -o program`
- `collect2` est un wrapper GNU de link, qui repose sur LD, qui n'est pas un outil de la collection GNU. LD est distribué via les “binutils” et est commun à la majorité des compilateurs (intel, PGI, LLVM...)

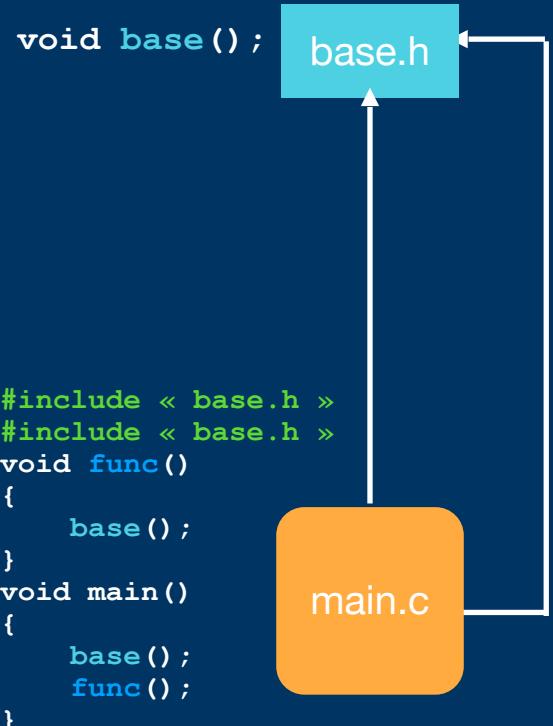
Preprocessing

- Interprétation de directives (#)
 - **#define / #undef** : Définition, déclaration de macro (fonctions, constantes...)
 - **#if(n)def / #else / #endif** : Compilation conditionnelle de sections de code
 - **#include** : Inclusion de fichiers récursives, nécessité des guards
 - **#error / #warning / #todo** : Influence la sortie de compilation
 - **#pragma...** : gestion compilateur
- Certaines constantes existent (**_FILE_**, **_LINE_**, **_DATE_**, **_TIME_**), et extensions selon l'architecture (**_WIN32**, **_APPLE**, **_linux**)
- Programme : **cpp main.c main.i**, Résultat obtenu avec : **gcc -E main.c**



Preprocessing

- Interprétation de directives (#)
 - **#define / #undef** : Définition, déclaration de macro (fonctions, constantes...)
 - **#if(n)def / #else / #endif** : Compilation conditionnelle de sections de code
 - **#include** : Inclusion de fichiers récursives, nécessité des guards
 - **#error / #warning / #todo** : Influence la sortie de compilation
 - **#pragma...** :
- Certaines constantes existent (**_FILE_**, **_LINE_**, **_DATE_**, **_TIME_**), et extensions selon l'architecture (**_WIN32**, **_APPLE**, **_linux**)
- Programme : **cpp main.c main.i**, Résultat obtenu avec : **gcc -E main.c**



Preprocessing

- Interprétation de directives (#)
 - **#define / #undef** : Définition, déclaration de macro (fonctions, constantes...)
 - **#if(n)def / #else / #endif** : Compilation conditionnelle de sections de code
 - **#include** : Inclusion de fichiers récursives, nécessité des guards
 - **#error / #warning / #todo** : Influence la sortie de compilation
 - **#pragma...** :
- Certaines constantes existent (**_FILE_**, **_LINE_**, **_DATE_**, **_TIME_**), et extensions selon l'architecture (**_WIN32**, **_APPLE**, **_linux**)
- Programme : **cpp main.c main.i**, Résultat obtenu avec : **gcc -E main.c**

```
void base();  
void base();  
void func()  
{  
    base();  
}  
void main()  
{  
    base();  
    func();  
}
```

main.c

Preprocessing

- Invocation : `cpp main.c main.i`
- Résultat: `gcc -E main.c [-P]`
- Ajout de chemins :
 - `-I/usr/include`
 - `export C_INCLUDE_PATH`
- `-D` / `-undef`
- `-include`

```
# 1 "main.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 361 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "main.c" 2
# 1 "./base.h" 1
void base();
# 2 "main.c" 2
# 1 "./func.h" 1
void func()
{
    base();
}
# 3 "main.c" 2
int main(int argc, char const *argv[])
{
    func();
    return 0;
}
```

```
→ gcc -E main.c -o main.i
main.c:1:10: erreur fatale: base.h : Aucun fichier ou dossier de ce type
#include <base.h>
           ^
compilation terminée.
```

Compilation

- Objectif : Transformer un contenu d'un **langage source** vers un **langage destination** (=*target*)
- Un programme source doit suivre un ensemble de règles afin d'être compris par le compilateur : la **grammaire**
- Le processus de compilation se découpe en trois grosses phases principales (simplifié)



Front-end

- Réalise l'analyse grammaticale du langage source pour produire une représentation intermédiaire (IR)

1. Analyse Lexicale : lecture de la source un caractère après l'autre pour former des mots (=lexème). Toute information superflue est ignorée (espaces...)

2. Analyse Syntaxique : Chacun de ces lexèmes est soumis à validation pour s'assurer qu'il font parti du langage

3. Analyse Sémantique : un ensemble de lexème forme une phrase, qui doit être sémantiquement juste

- Tout un pan de l'informatique moderne s'intéresse au formalisme du langage (pour créer son propre langage : `lex` & `yacc`)

Token	Example lexeme
<code>const</code>	<code>const</code>
<code>if</code>	<code>if</code>
<code>rellop</code>	<code><, <=</code>
<code>id</code>	<code>pi, count, age</code>
<code>num</code>	<code>3.14, 0</code>
<code>literal</code>	<code>"hello world"</code>

Middle-end

- Une fois le code généré, on obtient un programme sémantiquement juste mais loin d'être optimisé. De nombreuses passes sont en jeu ici
 - Graphe de control-flow, inlining
 - Élimination de code « mort » (DCE)
 - Transformation de boucles
 - Propagation de constantes
- Ce composant est indépendant de tout langage et de toute architecture. Réutilisation infinie, tant que la grammaire fourni la même sémantique (représentation intermédiaire)

```
int foo(void)
{
    int a = 24; ↑
    int b = 25; ↑
    int c;
    c = a * 4;
    return c;
    b = 24; ↑
    return 0;
}
```

Back-end

- Génération du programme pour la machine cible. Chaque architecture ayant un jeu d'instructions différent
- le code généré possède ses propres optimisations (=*machine-dependent Optimisations*), Vectorisation (SSE, AVX...)
- Registres, Pipelining, mode d'adressage (Absolute, PC-centric, register-*...), code redondant
- Génération des fichiers contenant le code assembleur (.s)
- Résultat de **gcc -S main.c**

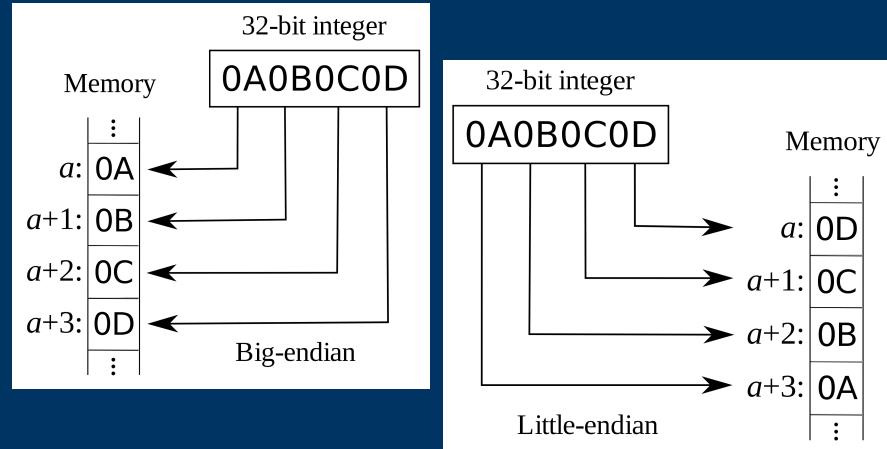
```
.section    __TEXT,__text,regular,pure_instructions
.build_version macos, 10, 14
.globl _func
.p2align 4, 0x90
_func:
.cfi_startproc
## %bb.0:
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq %rsp, %rbp
.cfi_def_cfa_register %rbp
movb $0, %al
callq _base
popq %rbp
retq
.cfi_endproc

.globl _main
.p2align 4, 0x90
_main:
.cfi_startproc
## %bb.0:
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq %rsp, %rbp
.cfi_def_cfa_register %rbp
subq $16, %rsp
movl $0, -4(%rbp)
movl %edi, -8(%rbp)
movq %rsi, -16(%rbp)
callq _func
xorl %eax, %eax
addq $16, %rsp
popq %rbp
retq
.cfi_endproc
```

Assemblage

Source : <https://en.wikipedia.org/wiki/Endianness>

- Transformation du code dépendant machine en code binaire
- Prise en compte de « l'Endianness » (Little / big)
- Création d'un fichier objet (.o) suivant le format ELF
 - **.text** : code défini dans dans le fichier
 - **.data / .bss** : variables globales du fichier initialisées / ou non



- **.rodata** : Constantes
- **.shstrtab** : tableau des chaînes de caractères
- Commande : **as main.s -o main.o**
- Résultat : **gcc -c main.c -o main.o**

readelf -SW main.o												
Il y a 9 en-têtes de section, débutant à l'adresse de décalage 0x180:												
En-têtes de section :												
[Nr]	Nom	Type	Adr	Décalage	Taille	ES	Fan	LN	Inf	Al		
[0]		NULL	0000000000000000	000000	000000	00		0	0	0		
[1]	.text	PROGBITS	0000000000000000	000040	000042	00	AX	0	0	1		
[2]	.data	PROGBITS	0000000000000000	000082	000000	00	WA	0	0	1		
[3]	.bss	NOBITS	0000000000000000	000082	000000	00	WA	0	0	1		
[4]	.rodata	PROGBITS	0000000000000000	000082	00000d	00	A	0	0	1		
[5]	.comment	PROGBITS	0000000000000000	00008f	00002d	01	MS	0	0	1		
[6]	.note.GNU-stack	PROGBITS	0000000000000000	0000bc	000000	00		0	0	1		
[7]	.eh_frame	PROGBITS	0000000000000000	0000c0	000078	00	A	0	0	8		
[8]	.shstrtab	STRTAB	0000000000000000	000138	000047	00		0	0	1		

Clé des fanions :
W (écriture), A (allocation), X (exécution), M (fusion), S (chaînes), I (info), L (ordre des liens), O (traitement supplémentaire par l'OS requis), G (groupe), T (TLS), C (compressé), x (inconnu), o (spécifique à l'OS), E (exclu), l (grand), p (processor specific)

Édition de liens

- Addition de plusieurs fichiers objets pour créer un exécutable
 - Fonction du « linker » : **ld** / **ld.gold** (version GNU)
 - Fusion des sections identiques
 - « Relocations » de symboles = réarrangement de l'espace d'adressage
 - Résultat : **gcc main.c**

```
↳ readelf -h ./a.out
En-tête ELF:
Magique:    7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Classe:          ELF64
Données:         complément à 2, système à octets de poids faible d'abord (little endian)
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
```

```
└─[Betelgeuse] ~
└─[Betelgeuse] └── objdump -d ./a.out | grep "<_start>"  
00000000004003b0 <_start>:  
└─[Betelgeuse] ~
└─[Betelgeuse] └── objdump -d ./a.out | grep "<main>"  
00000000004004ae <main>:
```

Édition de liens

Artefacts compilo-spécifiques:

- **crt1.o / crt0.o...** : Symboles de chargement du programme qui contient surtout le point d'entrée (fonction `_start()`) pour le déchargement du Shell.
- **crti.o / crt_n.o**: Symboles `_init` et `_fini`, constructeur et destructeur (old-style). Ils sont conservés par rétrocompatibilité avec les anciens systèmes. Remplacé par les segments des sections `.init_array` et `.fini_array`.
- **crtbegin.o / crtend.o**: C++ constructeurs
- **crtbeginS.o / crtendS.o** : remplace son équivalent quand `-fPIC`
- **crtbeginT.o / crtendT.o**: remplace son équivalent quand `-static`

Bibliothèque / Module

- Une application contient rarement tout le code dont elle a besoin et repose sur l'inclusion de modules déjà implémentés : réutilisation de code
- Une déclaration de la partie publique du module. C'est le header inclus, exposant variable & fonctions (ex: /usr/include)
 - Inclusion avec -I, C_INCLUDE_PATH, -include...
 - Invocation avec : #include <mymodule.h>
- Les code du module précompilé, qui est chargé lors de l'édition de liens pour l'optimisation du binaire final (exemple : /usr/lib[64])
 - Inclusion avec -L, [LD_]LIBRARY_PATH
 - Invocation avec: -l<nom du module> (ex: -lgcc pour libgcc.a)
 - Pas nécessaire pour les fonctions comme printf/scanf, pourquoi ?

```
↳ gcc main.c -I./include -L./lib -lmylib
```

```
↳ tree -L 1 /usr/include
/usr/include
├── aio.h
├── aliases.h
├── alloca.h
├── a.out.h
├── argp.h
├── argz.h
├── ar.h
├── arpa
├── asm
├── asm-generic
├── assert.h
├── bits
├── byteswap.h
├── bzlib.h
├── c++
├── complex.h
├── cpio.h
├── crypt.h
├── ctype.h
├── cursesapp.h
└── ...
↳ ls /usr/lib64/*.so -1
/usr/lib64/BugpointPasses.so
/usr/lib64/eppic_makedumpfile.so
/usr/lib64/ld-2.27.so
/usr/lib64/libanl-2.27.so
/usr/lib64/libanl.so
/usr/lib64/libasm-0.174.so
/usr/lib64/libbfd-2.29.1-23.fc28.so
/usr/lib64/libBrokenLocale-2.27.so
/usr/lib64/libBrokenLocale.so
/usr/lib64/libbtparse.so
/usr/lib64/libbz2.so
/usr/lib64/libc-2.27.so
/usr/lib64/libcc1.so
/usr/lib64/libclangAnalysis.so
/usr/lib64/libclangApplyReplacements.so
/usr/lib64/libclangARCMigrate.so
/usr/lib64/libclangASTMatchers.so
```

Bibliothèque / Module

- **STATIQUE (extension .a)**

- Le module est lié & injecté à l'application (archive de fichiers .o)
- Avantage : Indépendant de l'exécution
- Inconvénients : Binaire + lourd, fonction externes référencées « en dur » (pas de **dlopen()**)
- Outil : **ar** (-x : extract, -s : create, -t : list) Souvent : **ar rcs libmodule.a module.o**

- **DYNAMIQUE (extension .so)**

- Le module est référencé à la compilation et injecté à **l'exécution**
- Avantage : Binaire plus léger, rien n'est en dur dans le binaire, plus de souplesse à l'exécution
- Inconvénient : crée un overhead au runtime, dépendance entre environnement de compilation & d'exécution
- Outil : **ld**
- Via compilateur : **gcc -shared module.o -o libmodule.so**
- **-fPIC** indispensable dans 90% des cas de bibliothèques dynamiques (*Position Independent Code*)

Bibliothèque / Module

```
└─[Betelgeuse ~]─▶ gcc -static main.c -I.  
/usr/bin/ld : ne peut trouver -lc  
collect2: error: ld a retourné le statut de sortie 1
```

- Par défaut, il n'y a pas de distinctions entre statique et dynamique à l'édition de liens. Possibilité de forcer un link statique : `gcc -static` (génère une erreur si la version statique n'existe pas)
- Recherche de .so à la compilation : `-L` / `-Wl,-rpath`
- Chargement au runtime: `LD_LIBRARY_PATH` / `LD_PRELOAD`

```
└─[Betelgeuse ~]─▶ ldd ./a.out  
linux-vdso.so.1 (0x00007ffffded6000)  
libc.so.6 => /lib64/libc.so.6 (0x00007ff6e0cf2000)  
/lib64/ld-linux-x86-64.so.2 (0x00007ff6e10b1000)
```

```
└─[Betelgeuse ~]─▶ gcc main.c -I. -Wl,-rpath=/lib64/; \  
> readelf -dW ./a.out  
  
Section dynamique à l'offset 0xe50 contient 21 entrées :  
Étiquettes Type Nom/Valeur  
0x0000000000000001 (NEEDED) Bibliothèque partagée: [libc.so.6]  
0x000000000000000f (RPATH) Bibliothèque rpath: [/lib64/]  
0x000000000000000c (INIT) 0x400398
```

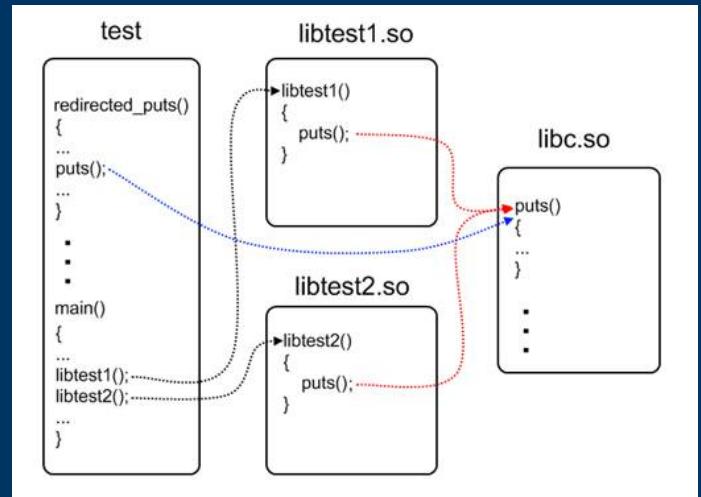
```
└─[Betelgeuse ~]─▶ readelf -dW ./a.out  
  
Section dynamique à l'offset 0xe60 contient 20 entrées :  
Étiquettes Type Nom/Valeur  
0x0000000000000001 (NEEDED) Bibliothèque partagée: [libc.so.6]  
0x000000000000000c (INIT) 0x400390  
0x000000000000000d (FINI) 0x400534  
0x0000000000000019 (INIT_ARRAY) 0x600e50  
0x0000000000000001b (INIT_ARRAYSZ) 8 (octets)  
0x0000000000000001a (FINI_ARRAY) 0x600e58
```

Bibliothèque / Module

- Chargement dynamique via **libdl.so**
 - **h = dlopen(« mylib.so »)**: Charge une bibliothèque (appel du loader, chargement mémoire, etc...)
 - **dlsym(h, « i »)**: Renvoie l'adresse d'un symbole chargé en mémoire (variable, fonction, etc...)
 - **dlclose(h)**: Ferme la bibliothèque, déchargement...
- Requiert une bibliothèque dynamique !

Bibliothèque / Module

- L'introspection est l'art de charger une bibliothèque à l'exécution, pour venir « écraser » les symboles existants par ceux re-définis.
- Exemple : `LD_PRELOAD=myalloc.so ./a.out`
- Conserver la cohérence de l'application : rappeler la fonction originale via `dlsym(« func », RTLD_NEXT)` ;
- Le prochain symbole est déterminé par l'ordre des bibliothèques tel qu'indiqué à la compilation



```
$ ldd ./IMB-MPI1
    linux-vdso.so.1 (0x00007fff493b1000)
    libmpc_framework.so => $INSTALL_PATH//x86_64/x86_64//lib/libmpc_framework.so (0x00007f74b2717000)
    libextls.so.0 => $INSTALL_PATH//x86_64/x86_64//lib/libextls.so.0 (0x00007f74b250d000)
    libportals.so.4 => /opt/sources/portals4/INSTALL/lib/libportals.so.4 (0x00007f74b22e7000)
    libm.so.6 => /lib64/libm.so.6 (0x00007f74b1f53000)
    libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f74b1d34000)
    libert.so.1 => /lib64/libert.so.1 (0x00007f74b1b2c000)
    libsctk_arch.so => $INSTALL_PATH//x86_64/x86_64//lib/libsctk_arch.so (0x00007f74b1929000)
    libhwloc.so.5 => $INSTALL_PATH//x86_64/x86_64//lib/libhwloc.so.5 (0x00007f74b16f0000)
    libxml2.so.2 => $INSTALL_PATH//x86_64/x86_64//lib/libxml2.so.2 (0x00007f74b138e000)
    libmpcgetopt.so.0 => $INSTALL_PATH//x86_64/x86_64//lib/libmpcgetopt.so.0 (0x00007f74b118a000)
    libc.so.6 => /lib64/libc.so.6 (0x00007f74b0dc000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f74b2f3a000)
    libdl.so.2 => /lib64/libdl.so.2 (0x00007f74b0bc7000)
    libev.so.4 => /lib64/libev.so.4 (0x00007f74b09b8000)
    liblzma.so.5 => /lib64/liblzma.so.5 (0x00007f74b0791000)
```

En résumé

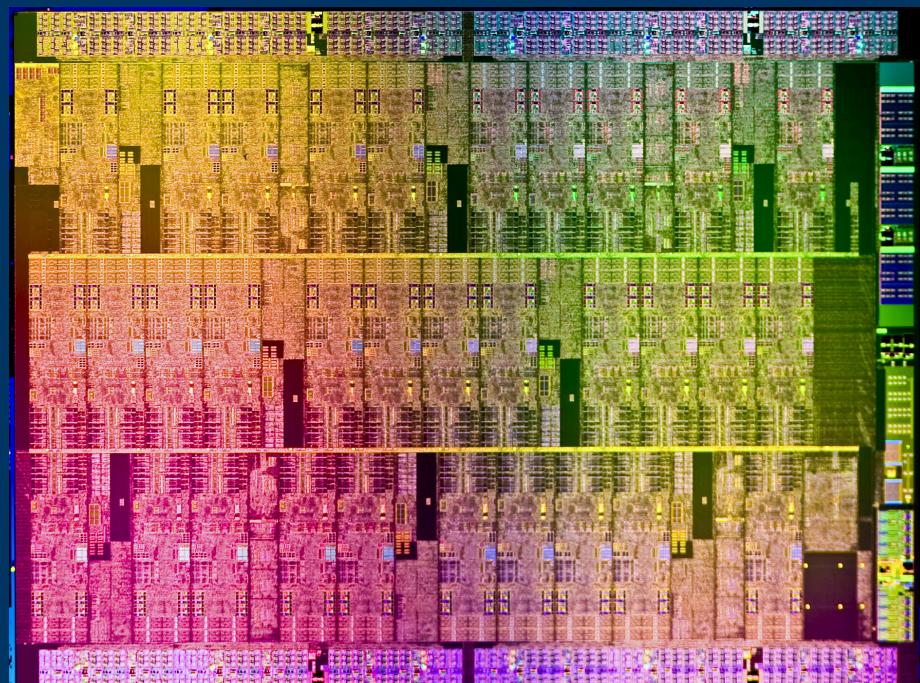
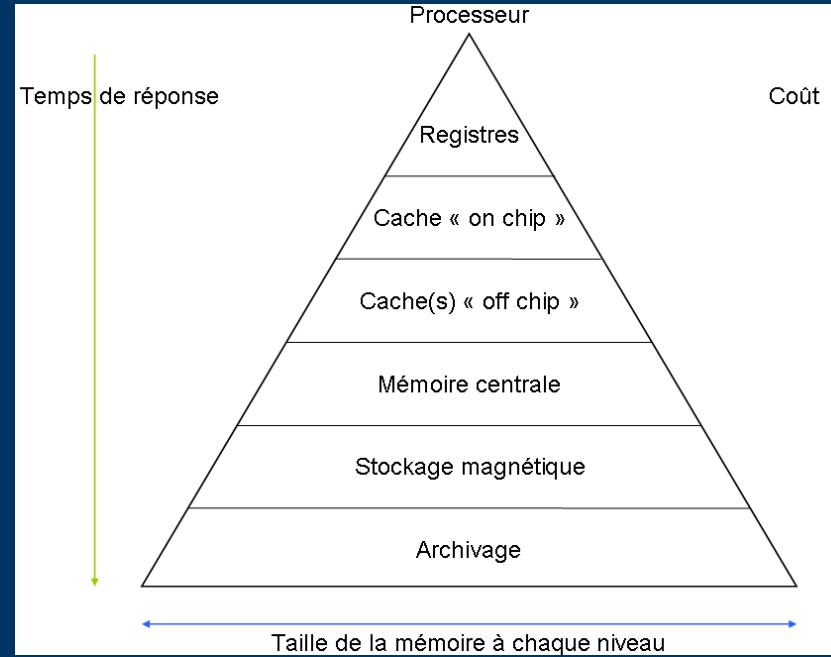


- Preprocessing : `gcc -E main.c (cpp)`
 - Ajout de règles : `INCLUDE_PATH= / -I / -include / -D / -undef`
- Compilation : `gcc -S main.c`
- Code objet : `gcc -c main.c (as)`
- Édition de liens : `gcc main.c (ld)`
- Ajout de librairies : `-L<chemin> / -l<chemin> / <chemin absolu>`
 - Statique (.a) : `ar rcs / -static / LIBRARY_PATH`
 - Dynamique (.so) : `-fPIC -shared / -Wl,-rpath / LD_LIBRARY_PATH`

Gestion mémoire

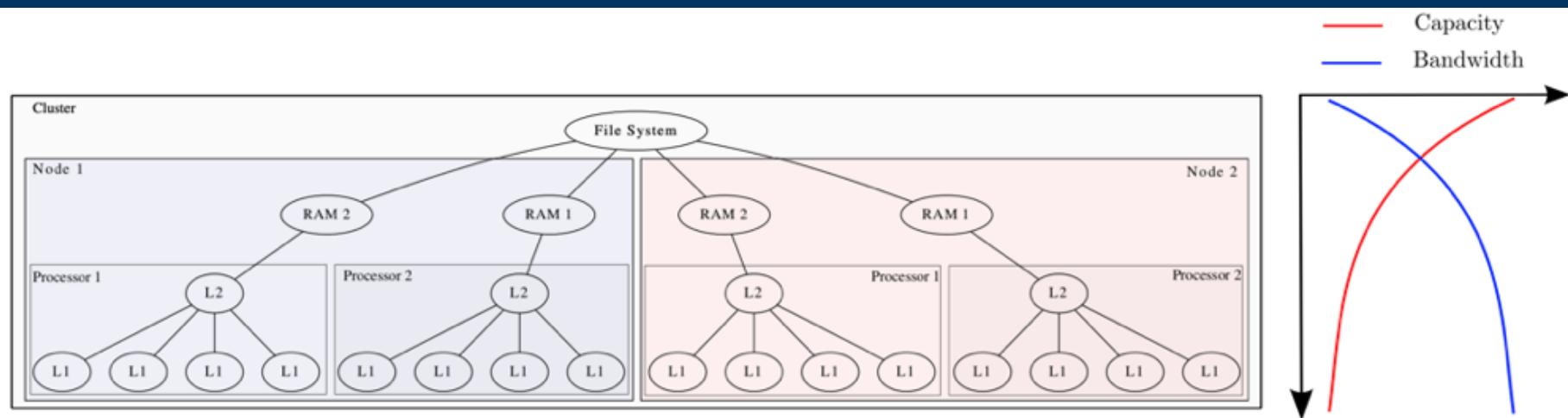
Histoire de la mémoire

- 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 elles 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



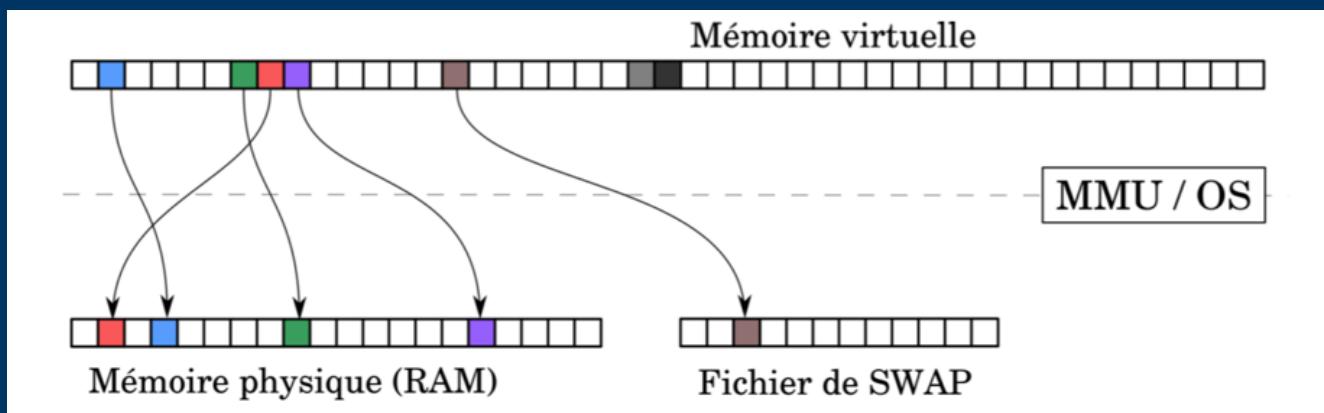
Composant dédié : la MMU (Memory Management Unit), composant noyau, généralement par application, qui s'occupe de gérer la traduction virtuelle <-> physique.

=> 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

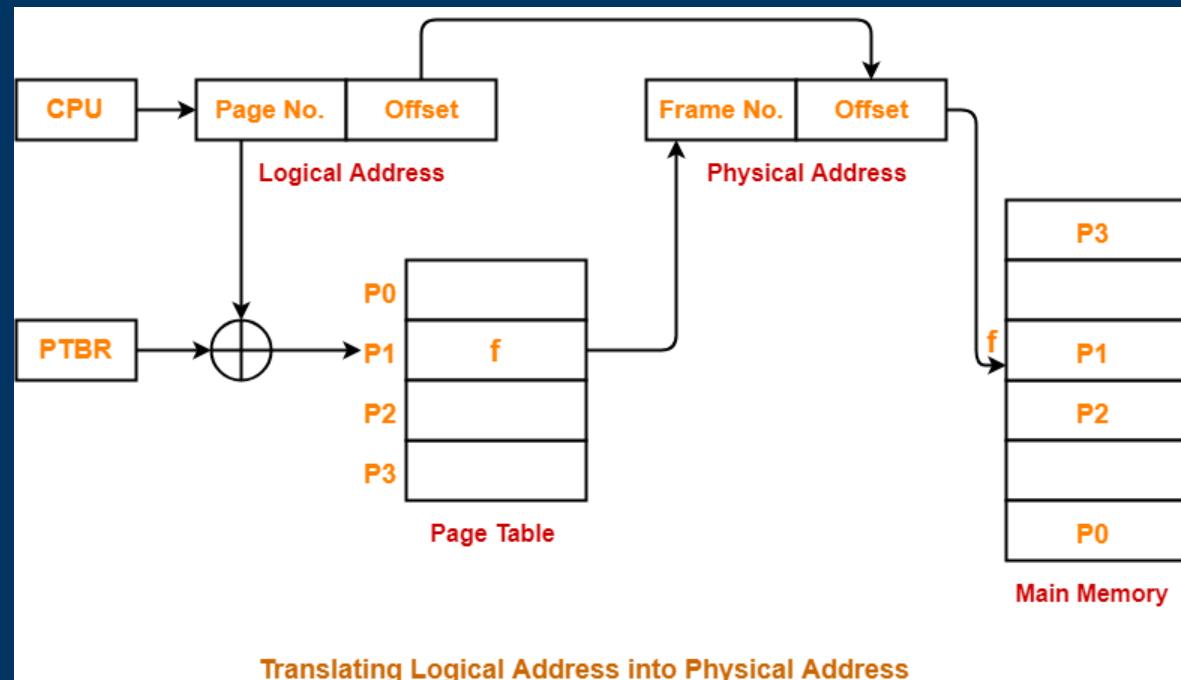
- => Nécessité d'un mapping virtuel <-> physique
- Découpage en blocs de taille définie. Pour être accessible à l'application, une bloc doit être chargée en mémoire centrale (DRAM).
- 1 bloc en mémoire physique = 1 cadre
- 1 bloc en mémoire virtuelle = 1 page
- 1 cadre = 1 page
- la MMU s'assure de fournir,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



Pagination

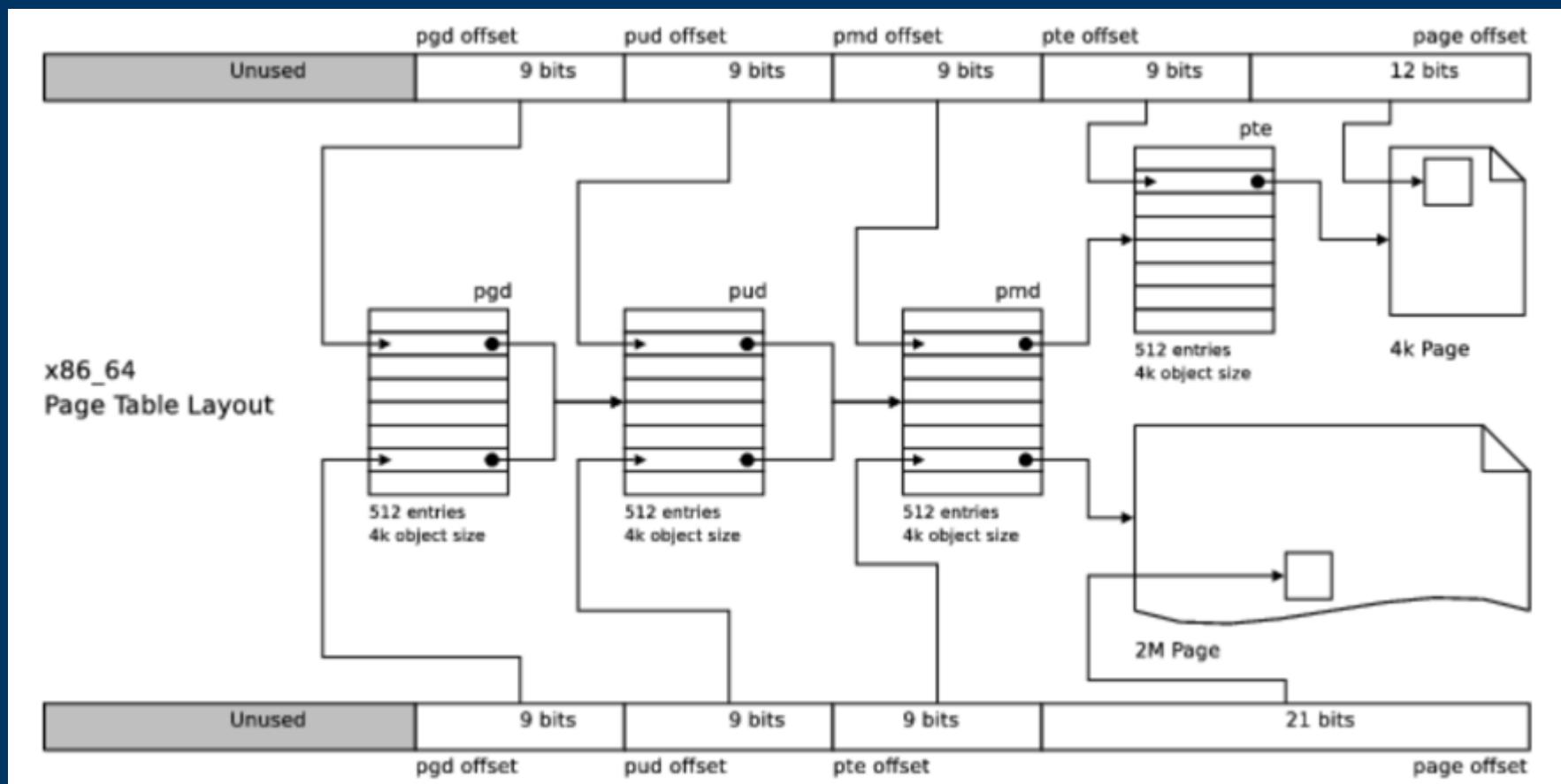
- => Nécessité d'indexer les correspondances pages <-> cadre
- Table des pages, stockées dans la mémoire noyau du processus)
- L'adresse est découpée, pour créer un ID de page et un offset
- La table référence, pour chaque ID de page, l'ID de cadre associé
- Couplé à l'offset il permet d'avoir l'adresse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

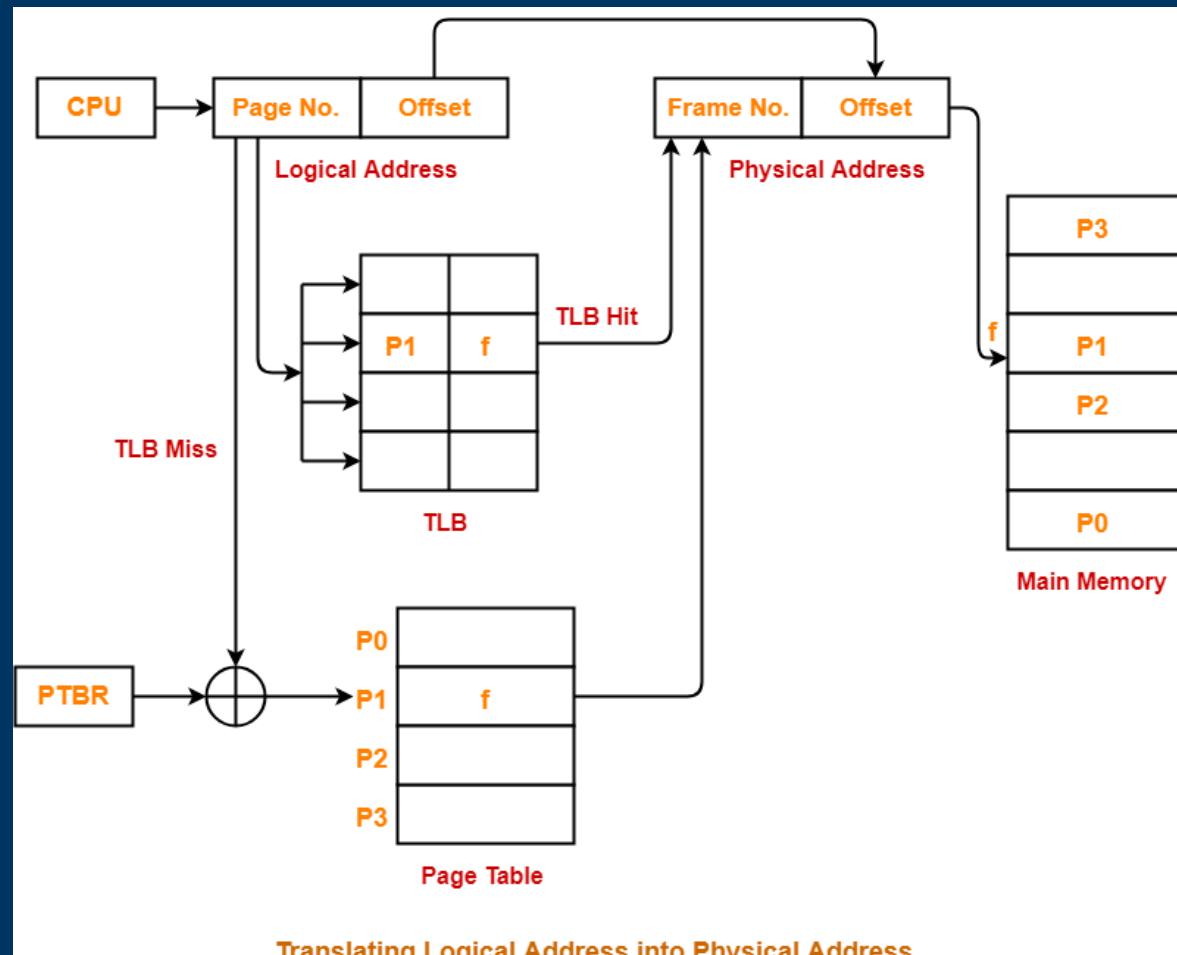
- **LRU**: La plus vieille page utilisée (coûteux)
- **FIFO**: Première mappée, première sortie
- **Deuxième chance**: 1 bit d'utilisation est associée à chaque page et passé à “1” à chaque utilisation. Au moment du remplacement, une page marquée dispose d'une “seconde chance”
- Remplacement **optimal**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



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>