

# Architecture Interne des Systèmes d'Exploitation (AISE)

Master CHPS

Jean-Baptiste BESNARD  
[jbbesnard@paratools.fr](mailto:jbbesnard@paratools.fr)

# Organisation du Cours

## Cour MATIN, TD après-midi


➡ 10/01: Généralités sur les OS et Utilisation de base

➡ 15/01 : La Chaine de Compilation et l'exécution d'un programme

➡ 16/01 : Les I/Os POSIX et Introduction aux Sockets

➡ 01/02 : Méthodes de communication Inter-Processus

➡ 06/02 : Programmation et reverse et Q/A projets (journée TD)

 14/02 : Mémoire avancée (mmap, madvise, pages, TLB, ...)

➡ 13/02 : TD Débogage (gdb, valgrind) && TD mesure du temps et profilage (perf, kcache-grind) et Q/A projets (journée TD)

➡ Un examen final (25 Mars matin)

# Généralités sur la Mémoire

# Isolation Mémoire

- Comment peut-on lancer plusieurs fois le même processus ?
- Peut-on garantir l'isolation entre les données de différents processus ?
- Comment assurer cette isolation ?
- Quelles sont les contraintes de performances sur la mémoire ?

# Cas de Plusieurs Processus

On désactive les adresses virtuelles aléatoires

```
# echo 0 > /proc/sys/kernel/randomize_va_space
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(){
    sleep(5);
    printf("'main' est a %p\n", main);
    return 0;
}
```

```
$ ./a.out &
```

```
[1] 10131
```

```
$ ./a.out &
```

```
[2] 10132
```

```
$ ./a.out &
```

```
[3] 10133
```

```
'main' est a 0x55555555546f0
```

```
'main' est a 0x55555555546f0
```

```
'main' est a 0x55555555546f0
```

**Plusieurs programmes concurents  
Sont à la même adresse !**

# Cas de Plusieurs Processus

Si on laisse les adresses aléatoires:

```
# echo 1 > /proc/sys/kernel/randomize_va_space
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(){
    sleep(5);
    printf("'main' est a %p\n", main);
    return 0;
}
```

```
$ ./a.out &
```

```
[1] 10131
```

```
$ ./a.out &
```

```
[2] 10132
```

```
$ ./a.out &
```

```
[3] 10133
```

```
'main' est a 0x55e53cfb56f0
```

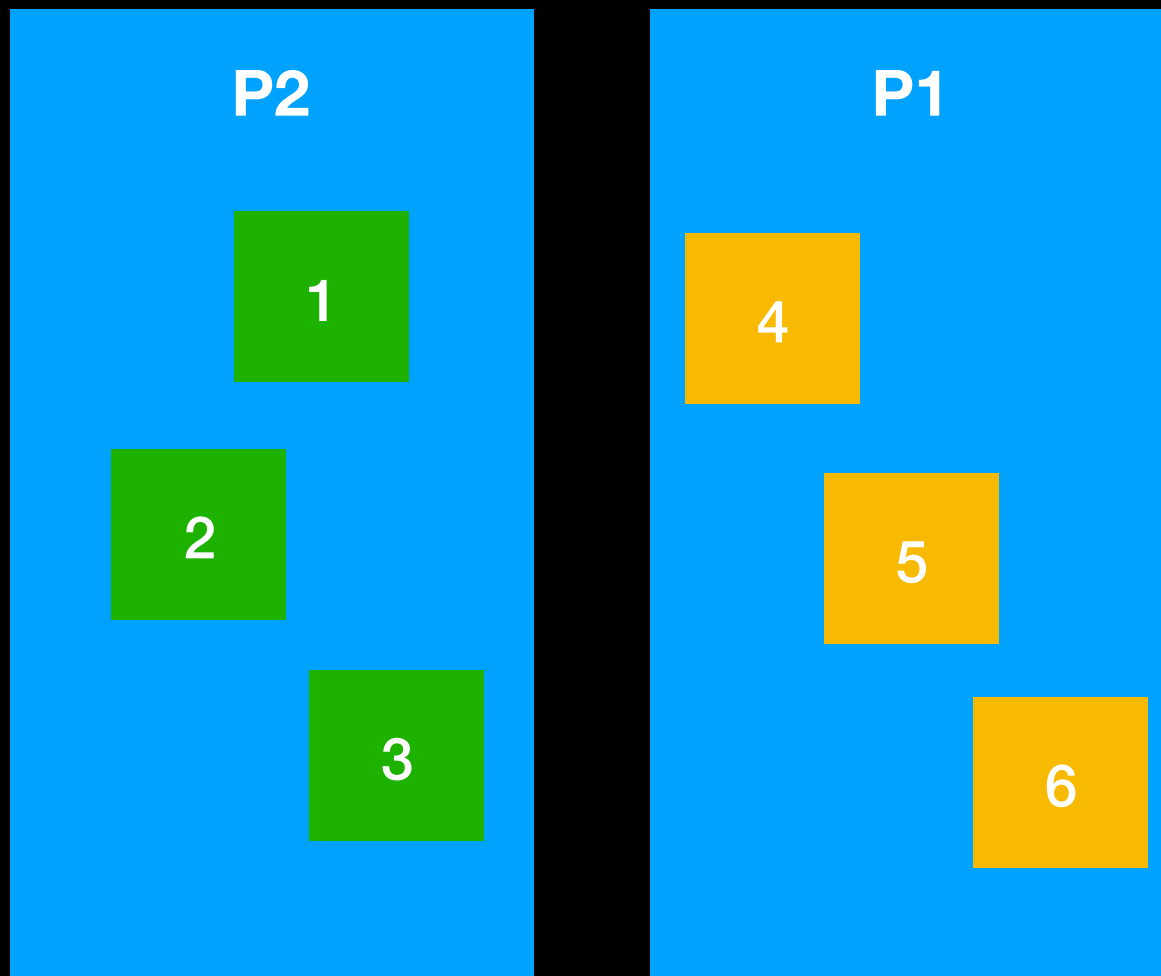
```
'main' est a 0x563936ce16f0
```

```
'main' est a 0x56110fd0d6f0
```

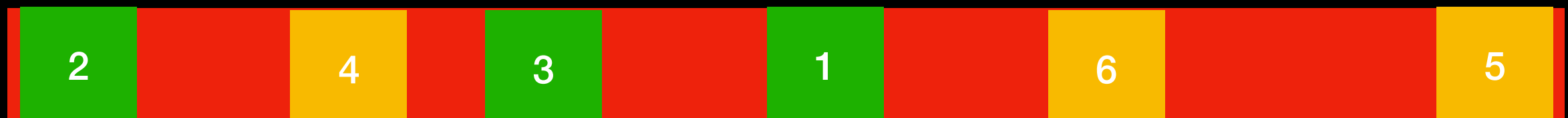
Les adresses sont légèrement différentes. Cependant, chaque processus peut **virtuellement** adresser toute la mémoire.

**Dans un but de sécurité**

# Notion de Mémoire Virtuelle



- Chaque processus a virtuellement accès à toute la mémoire du système;
- Cette mémoire est allouée à la demande et non immédiatement (elle est d'abord virtuelle puis physique);
- Chaque partie de la mémoire virtuelle peut (ou non) être associée à de la mémoire physique (soit en RAM ou bien sur le disque dur SWAP)



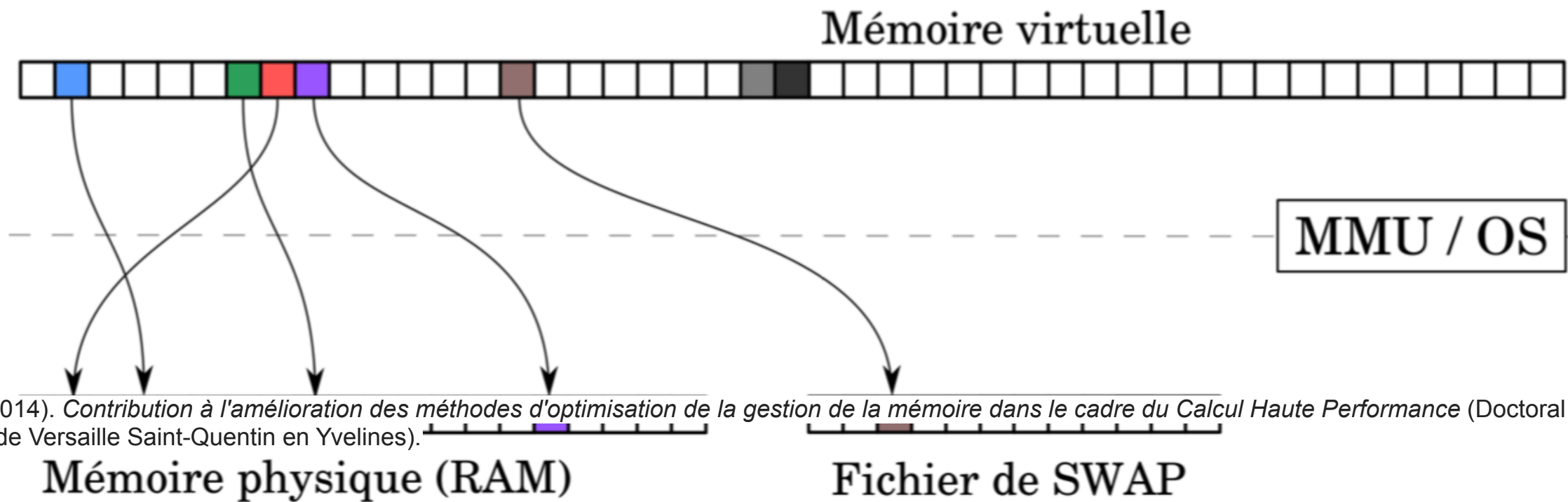
# Notion de Mémoire Virtuelle

## Avantages de la mémoire virtuelle:

- Efficacité de gestion: il est plus facile de ranger des données dans un grand espace de 64 bits au lieu de distribuer un sous ensemble de cet espace;
- Isolement: chaque programme dispose de son espace d'adressage entier;
- Multi-instances: deux programmes peuvent partager les même adresse ce qui simplifie la génération des codes (partiellement relocalisés);
- Projection passive: il est possible de mapper des fichiers les I/Os se faisant au moment de l'accès par de simple accès mémoire;
- Extension de la mémoire: la mémoire peut être étendue avec des ressources autre que la RAM, par exemple le disque dur via le SWAP;
- Compression de la mémoire: on peut optimiser les segments de mémoire virtuelle pour limiter l'utilisation mémoire (par exemple pour les machines virtuelles de type KSM).



# Notion de Pagination

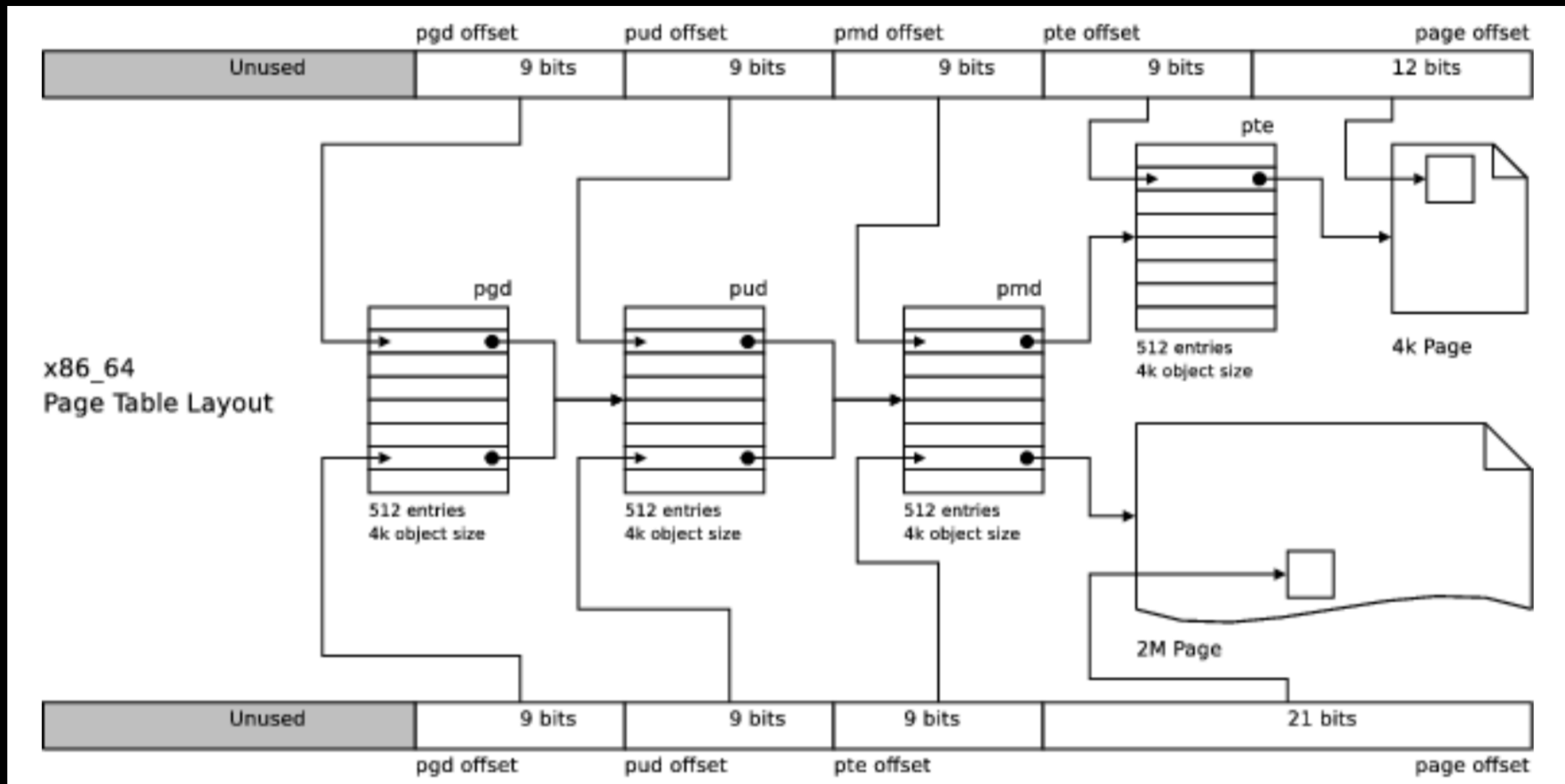


Valat, S. (2014). *Contribution à l'amélioration des méthodes d'optimisation de la gestion de la mémoire dans le cadre du Calcul Haute Performance* (Doctoral dissertation, University de Versailles Saint-Quentin en Yvelines).

La mémoire virtuelle est découpée en pages de 4KB (pages virtuelle) et chacune de ces page est associée (ou non) à une page physique (vrai mémoire). Par « défaut » les pages son de la mémoire virtuelle c'est à dire qu'elle n'a pas été « mappée » soit associée à de la mémoire physique. Cette opération s'appelle un défaut de page et demande une modification de la table de correspondance entre les page virtuelles et physiques : **la table des pages**.

Valat, S. (2014). *Contribution à l'amélioration des méthodes d'optimisation de la gestion de la mémoire dans le cadre du Calcul Haute Performance* (Doctoral dissertation, University de Versailles Saint-Quentin en Yvelines).

# Table des Pages

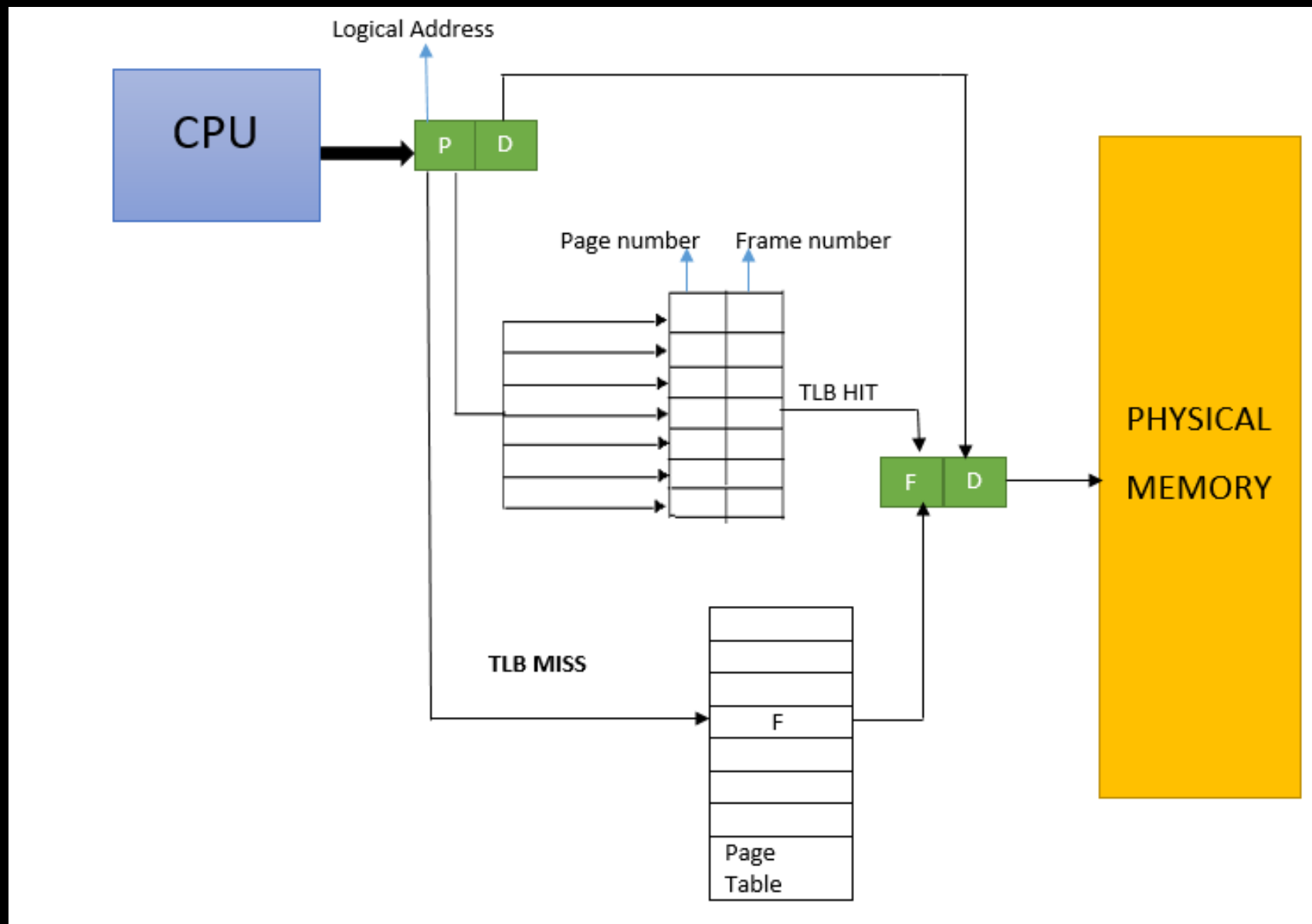


Adresses sur  $2^{48}$  soit maximum 256 TB de mémoire

On préfère une table hiérarchique à une table linéaire du fait que la mémoire d'un processus est principalement vide.

# Optimisation Matérielles

## Translation Lookaside Buffer (TLB)

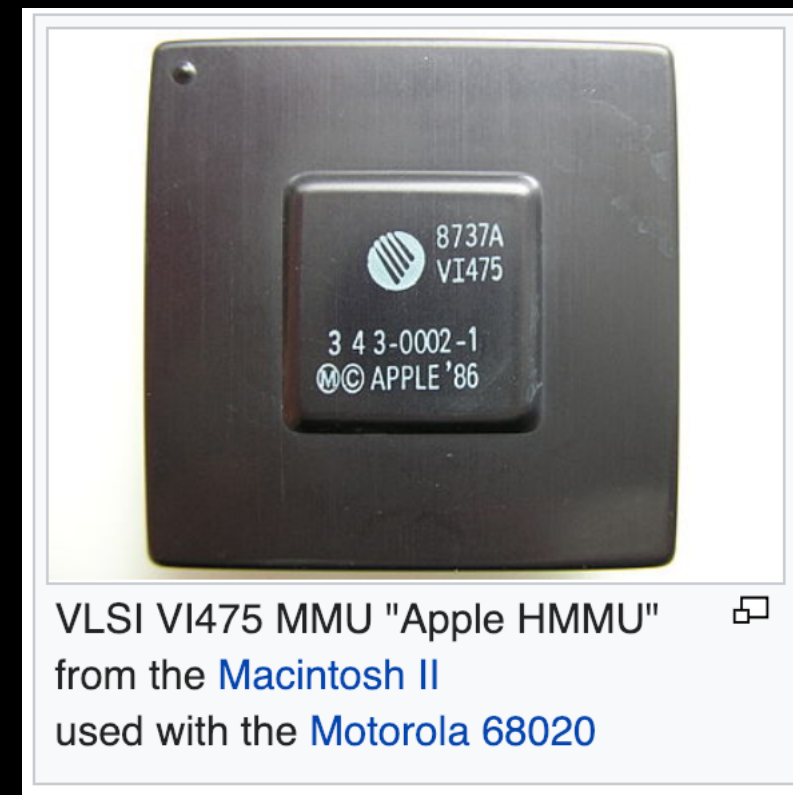


**Un cache associatif stockant des résolutions de pages (F veut dire Frame pour les cadres de pages) D pour déplacement, l'offset dans la page.**

# La Memory Management Unit

Quand le TLB est pris à défaut on fait ce qui s'appelle un TLB Miss, il faut donc parcourir la table des pages pour s'occuper la résolution d'adresse (page-walk).

Ce parcours de la table des pages est généralement fait en hardware sur les architectures modernes mais aura un coût supérieur à une résolution directe. Le composant logiciel qui s'en charge est la MMU qui parcourt de manière hardware la table des pages du processus courant.



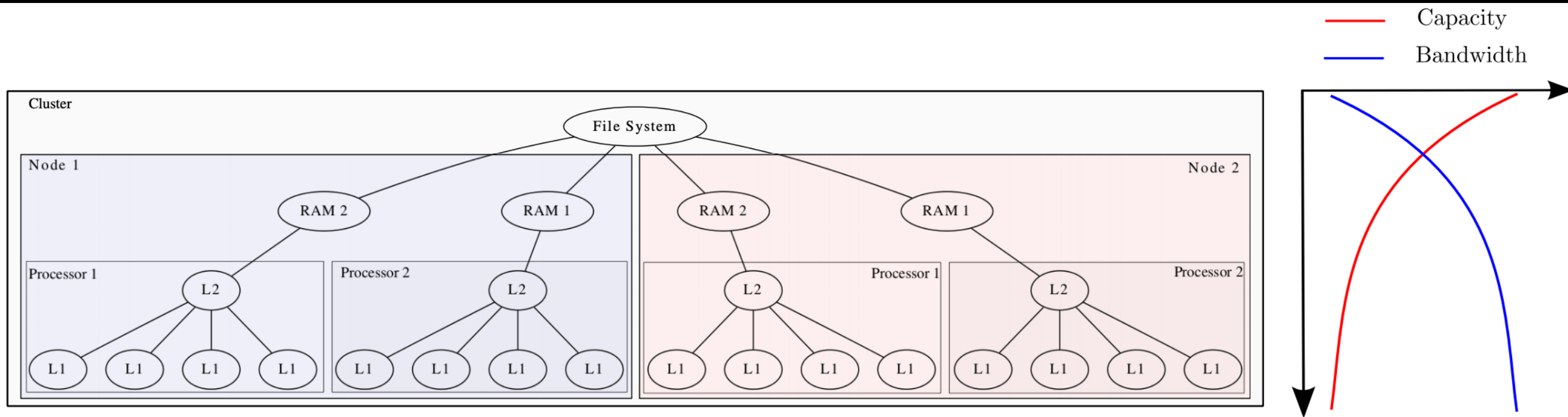
# Défaut de Page

La mémoire virtuelle est par défaut vide. Quand on accède pour la première fois à une page le kernel se charge de la « créer » en associant de la mémoire physique. Cette opération a un coût non négligeable car elle est faite en espace noyau.

**Le premier accès coûte toujours plus cher.**

**Dans la politique « first touch » c'est également le premier accès qui définit la localité mémoire.**

# Hiérarchie 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.**

**Allouer de la Mémoire**

# Allouer de la mémoire

```
void *malloc(size_t size);
```

The `malloc()` function allocates `size` bytes and returns a pointer to the allocated memory. The memory is not initialized. If `size` is 0, then `malloc()` returns either `NULL`, or a unique pointer value that can later be successfully passed to `free()`.



# Allouer de la mémoire « vide »

```
void *calloc(size_t nmemb, size_t size);
```

The `calloc()` function allocates memory for an array of `nmemb` elements of `size` bytes each and returns a pointer to the allocated memory. The memory is set to zero. If `nmemb` or `size` is 0, then `calloc()` returns either `NULL`, or a unique pointer value that can later be successfully passed to `free()`.

**Mieux que malloc memset.**

# Redimensionner une allocation

```
void *realloc(void *ptr, size_t size);
```

The `realloc()` function changes the size of the memory block pointed to by `ptr` to `size` bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will not be initialized. If `ptr` is `NULL`, then the call is equivalent to `malloc(size)`, for all values of `size`; if `size` is equal to zero, and `ptr` is not `NULL`, then the call is equivalent to `free(ptr)`. Unless `ptr` is `NULL`, it must have been returned by an earlier call to `malloc()`, `calloc()` or `realloc()`. If the area pointed to was moved, a `free(ptr)` is done.

# Libérer de la mémoire

```
void free(void *ptr);
```

The `free()` function frees the memory space pointed to by `ptr`, which must have been returned by a previous call to `malloc()`, `calloc()`, or `realloc()`. Otherwise, or if `free(ptr)` has already been called before, undefined behavior occurs. If `ptr` is `NULL`, no operation is performed.

# Primitive d'allocation

- L'allocateur recycle de la mémoire provenant du système
- Le système fournit de la mémoire via un appel principal mmap
- Et on rend de la mémoire via munmap
- L'allocateur « malloc/free » peut recycler des blocs de mémoire pour éviter de faire trop souvent des appels système
- Le système renvoie toujours de la mémoire mise à « 0 » pour des raison de sécurité (pas de données d'autres processus) cela s'appelle le « zero-page ».

# Mmap / Munmap

```
#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` 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)`.

# Example 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;
```

```
}
```

# Cas Particulier des Strings

# Petit Example

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

```
int main(){  
    char * v = "toto";  
    v[0] = '1';  
    printf("v = %s\n", v);
```

```
    return 0;
```

```
}
```

Que fait ce code ?



# Petit Example

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

**Pourquoi ?**

# Petit Example

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

**Car tout string en C est  
une constante lorsque  
défini 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;
}
```

Ce code est-il correct ?

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

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

```
    free(v);
```

```
    return 0;
}
```

# 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**

# Mesure Précise du Temps

# Le TSC

C'est une horloge bas niveau à l'échelle de la nanoseconde sur les architectures x86. Le timestamp counter. Il se lit en assembleur. On utilise en général le header de la lib FFTW et cycle.h



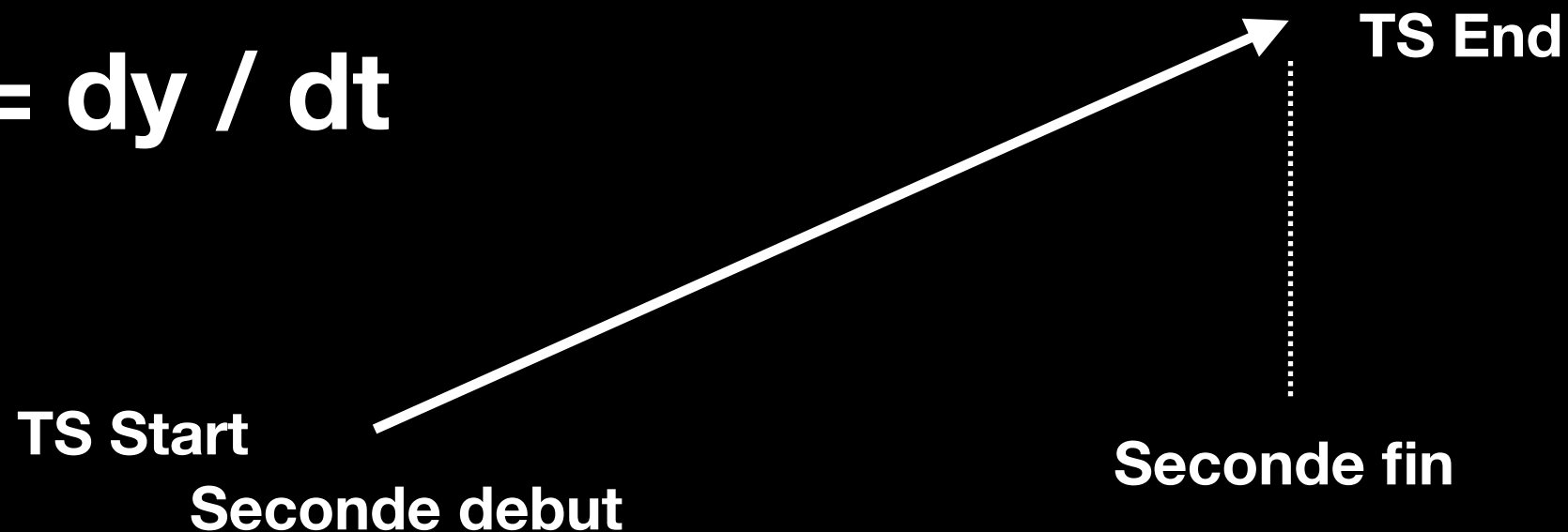
# Le TSC

Il suffit alors d'inclure `cycle.h` et de faire un appel à « `getticks()` ». Cette valeur est monotone avec un incrément constant.

# Estimation de Fréquence

Pour estimer la fréquence du TSC il faut faire une approximation linéaire avec un autre timer en secondes via `gettimeofday` par exemple.

$$F = dy / dt$$



# Exemple de Code

```
double ticks_per_second ;

void calibrate()
{
    struct timeval tv_start, tv_end;
    gettimeofday( &tv_start , NULL );
    double start_ts = getticks();
    sleep(5);
    gettimeofday( &tv_end , NULL );
    double end_ts = getticks();
    double start_time = (tv_start.tv_usec) * 1.0e-06 + (tv_start.tv_sec) * 1.0;
    double end_time = (tv_end.tv_usec) * 1.0e-06 + (tv_end.tv_sec) * 1.0;
    ticks_per_second = ( (double)(end_ts - start_ts) ) / ( end_time - start_time );
}

int main(int argc, char **argv){
    calibrate();
    printf("Ticks per sec is %g\n", ticks_per_second);
    return 0;
}
```

\$ ./a.out

Ticks per sec is 3.59162e+09

# Période du Timer

$$T = 1 / F$$

$$T = 1 / 3.59162e+09$$

$$T = 0,27e-09$$

**Une unité décrit un temps inférieur à la NS !**

# Exemple de Mesure de la Hiérarchie Mémoire

# Remplir et Lire un tableau de Taille croissante

```
for( i = 0 ; i < s ; i++ )
{
    buf[i] = j;
}

for( j = 0; j < 500; j++)
{

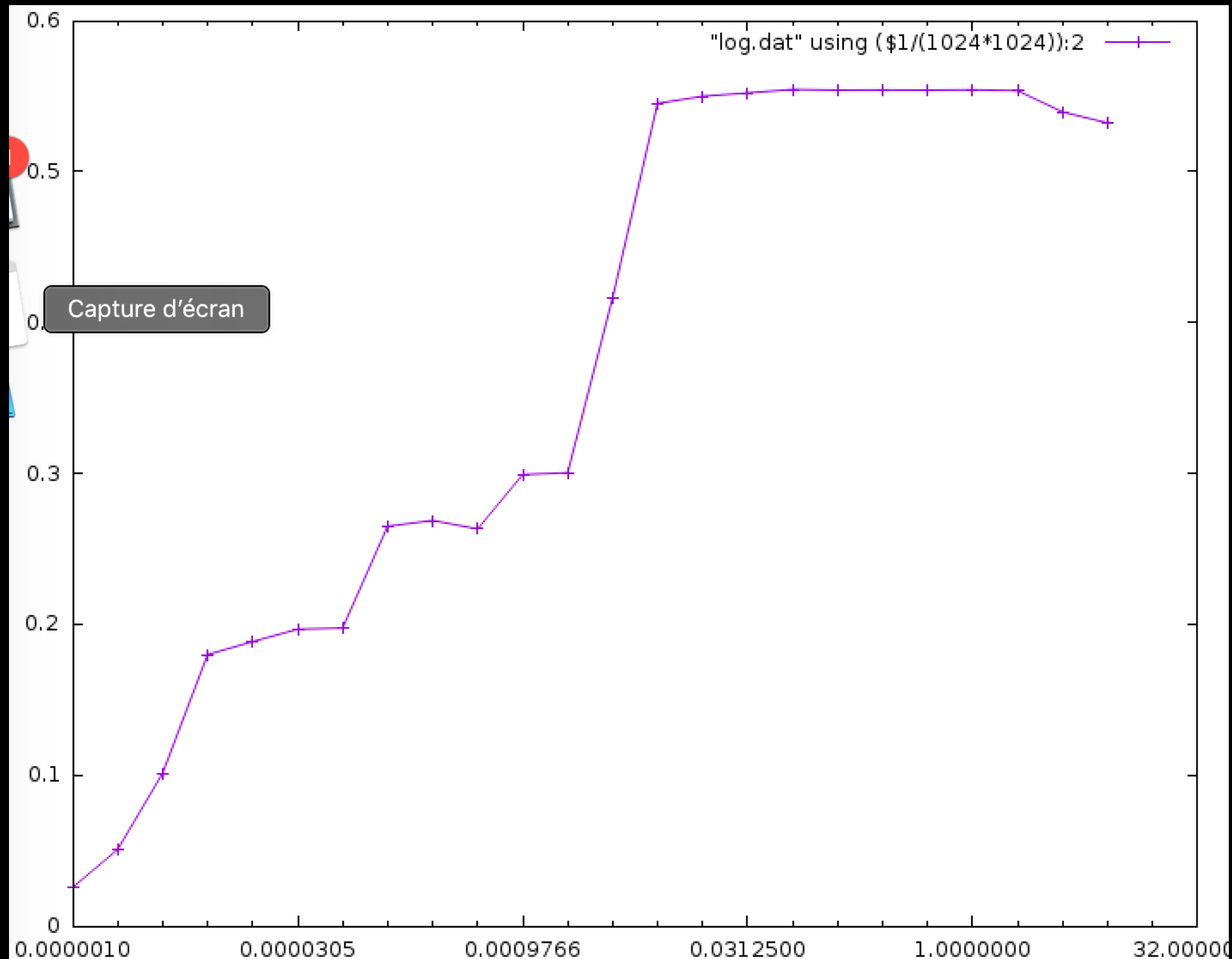
    start = getticks();
    for( i = 0 ; i < s ; i++ )
    {
        obuf[i] = buf[i];
    }
    end = getticks();
    total += end - start;
}

printf("%g %g\n", s, ( 500 * s ) / total );
```

Mesure du temps d'accès en cycles:

$$Ta = total / (1024 * s)$$

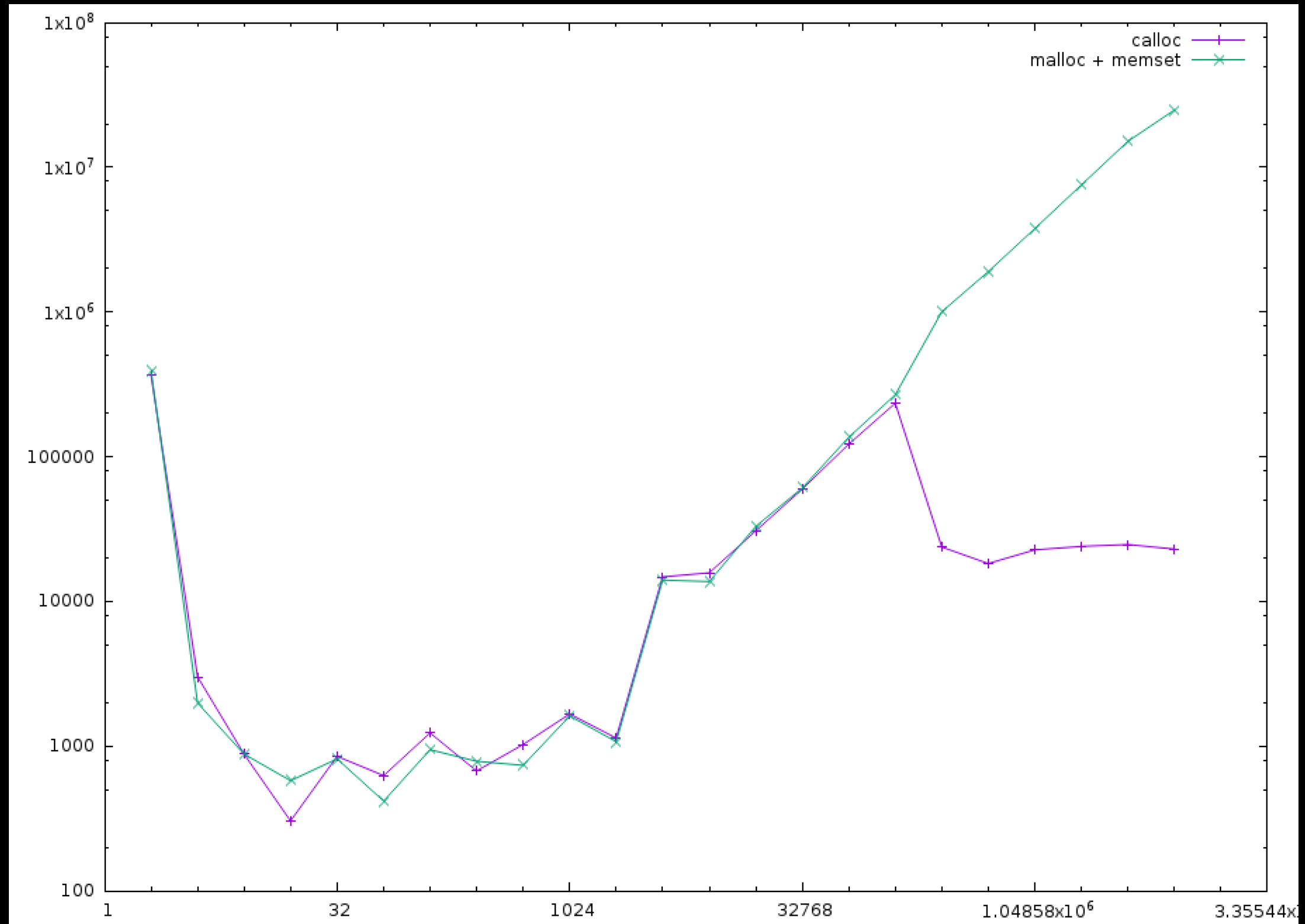
# Remplir et Lire un tableau de Taille croissante



# Calloc vs Malloc + memset



# Comparaison des Allocation à 0



**Base de GNPLOT**

# Lancement

```
$ gnuplot <script>
```

**Lancer sans script lance en interactif**

# Plot Basique

> plot sin(x) w lp

> plot  $x^{**2}$  w lp

> plot « file » using 1:2 w lp

# Multiplot

```
> plot sin(x) w lp, cos(x) w lp  
> plot « file1 » using 1:2 w lp,  
« file2 » using 1:2 w lp
```

# Décorations

Types de lignes:

> test (paramètres pour lt)

Titres:

> plot x\*\*2 title «  $x^2$  »

Axes:

> set xlabel « mon label »

> set ylabel « mon label »

# Exemple de Script

```
set terminal png  
set output "plot.png"
```

```
set title "Sinus et Cosinus"  
set xlabel "Axe des abscices"  
set ylabel "Axe des ordonnees"
```

```
plot sin(x) title "Sinus" w lp, cos(x) title "Cosinus" w lp
```

# Exercices de TP



# Bibliothèque « String »

Réalisez une bibliothèque gérant un type « string » supportant les opérations suivantes:

- **string \* string\_init(char \*)**  
crée un nouveau string
- **int string\_release(string \*)**  
libère un string précédemment créé
- **int string\_append( string \*, char \*)**  
ajoute un char \* à la fin d'un string
- **int string\_display( string \*)**  
affiche un string
- **string \* string\_slice( string \*, int start, int end)**  
renvoie un sous ensemble du string
- **string \* string\_repeat(string \*)**  
renvoie un string où le paramètre est répété

# Définition d'un Timer Calibré

En utilisant « `cycle.h` » et `gettimeofday`, calculez la fréquence d'un timer basé sur le TSC (double `getticks` dans `cycle.h`)

<http://www.fftw.org/cycle.h>

Regroupez ces fonctionnalités dans une petite bibliothèque.

- **`double ticks_per_sec();`**  
renvoie le nombre de ticks par seconde
- **`double getticks()`**  
renvoie le nombre de ticks courant

# Comparaison Calloc / Malloc + Memset

**Réalisez une mesure permettant de comparer calloc et malloc +memset. Vous devrez générer un graphique avec les axes correctement nommés.**

# Dictionnaire

**Charger la liste de mots français présente dans Cours\_6/dico/dico.txt dans une liste chaînée. Puis parcourez cette liste à la recherche d'anagrammes.**

# Mesure des Effets de Cache

En reprenant le code:

```
for( i = 0 ; i < s ; i++ )
{
    buf[i] = j;
}

for( j = 0; j < 500; j++)
{

    start = getticks();
    for( i = 0 ; i < s ; i++ )
    {
        obuf[i] = buf[i];
    }
    end = getticks();
    total += end - start;
}

printf("%g %g\n", s, ( 500 * s ) / total );
```

**Vérifiez si vous pouvez vous aussi observer les variations des performances en fonction de la taille du dataset.**