

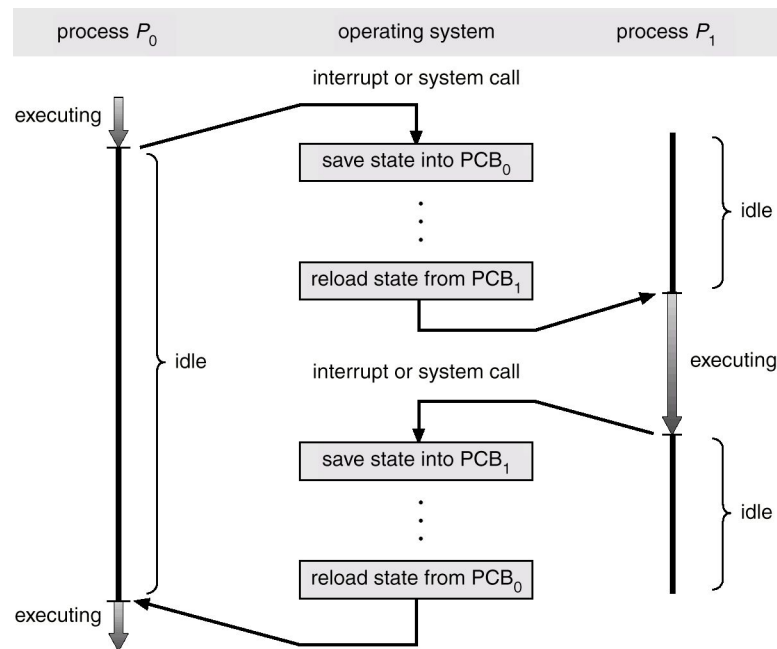
# **Principes des Systèmes d'exploitation**

IUT de Villetaneuse  
D. Buscaldi

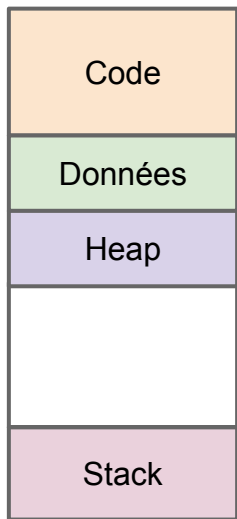
# 7. Threads

# Les Threads

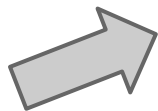
- Processus classique: *lourd*
  - Image mémoire: texte, données, pile
  - BCP
  - Commutation entre un processus et l'autre prend du temps
- idée: avoir une unité moins lourde à gérer que le processus, le thread
- Thread = processus léger
  - partage du code et des données du même processus
- Linux: librairie **pthread**



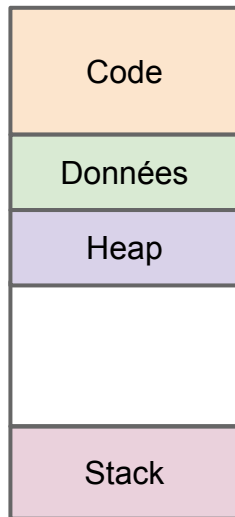
Avant fork():



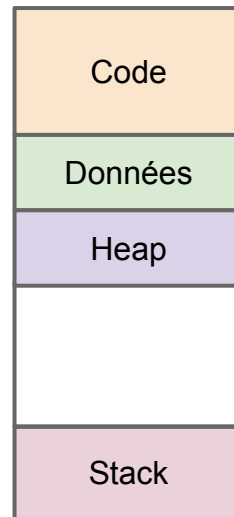
fork()



Père

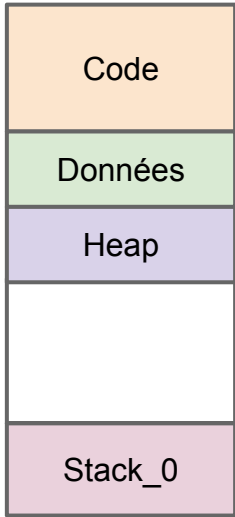


Fils



Fork: chaque processus a son espace de mémoire séparé

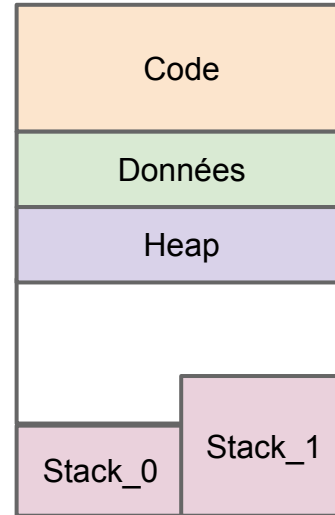
Avant création d'un  
nouveau thread:



pthread\_create()



Après création d'un  
nouveau thread:



- Même processus, code et données
- Chaque thread a son propre stack

# Création d'un thread

- Fonction **pthread\_create**:

```
int pthread_create(pthread_t *thread, pthread_attr_t* attr, void* (*start_routine)(void *), void* arg);
```

- *\*thread*: pointeur au thread qu'on demande de créer
- *attr*: structure qui spécifie si le thread sera synchronisable ou pas (synchronisation similaire à la **wait** pour les processus)
- *\*start\_routine*: pointeur vers la fonction à exécuter par le thread
- *arg*: pointeur des données à passer en paramètre à la fonction *start\_routine*
- Valeur de retour: -1 si erreur pendant la création du thread

Note: compilation avec la librairie **pthread**: ajouter **-lpthread** aux paramètres gcc

# Exemple

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NTHREADS 2

void *my_thread (void * arg) {
    int i;
    int n = *((int *) arg);
    for (i = 0 ; i < 5 ; i++) {
        printf ("Thread %d: %d\n", n, i);
        sleep(1);
    }
    pthread_exit (0);
}
```

```
void main () {
    pthread_t th;
    int rc, i;
    int thread_args[NTHREADS+1];

    for(i=1; i<= NTHREADS; i++){
        thread_args[i]=i;
        rc=pthread_create(&th, NULL, my_thread, &thread_args[i]);
    }
    sleep(3);
}
```

```
$> ./test_threads
Thread 1: 0
Thread 2: 0
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
```

Il faut synchroniser!

Problème: le processus termine avant la terminaison des threads...

# Synchronisation de threads

- Fonction **pthread\_join**:

```
int pthread_join(pthread_t *thread, void** retval);
```

- \*thread: pointeur au thread qu'il faut attendre
  - retval: le status final du thread (par exemple, s'il a terminé normalement ou il a été annulé)
  - Valeur de retour: 0 si tout va bien, -1 si erreur
- 
- Le thread doit être joignable (situation par défaut):
    - Un thread peut être dans un état soit joignable (*joinable*), soit détaché (*detached*)
    - Si un thread est joignable, un autre thread peut appeler **pthread\_join** pour attendre que ce thread se termine, et récupérer sa valeur de sortie.
    - Ce n'est que quand un thread terminé et joignable a été joint que ses ressources sont rendues au système



# Exemple

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NTHREADS 2

void *my_thread (void * arg) {
    int i;
    int n = *((int *) arg);
    for (i = 0 ; i < 5 ; i++) {
        printf ("Thread %d: %d\n", n, i);
        /* On peut enlever les sleep */
    }
    pthread_exit (0);
}

void main () {
    pthread_t threads[NTHREADS];
    int rc, i;
    int thread_args[NTHREADS+1];

    for(i=1; i<= NTHREADS; i++){
        thread_args[i]=i;
        rc=pthread_create(&threads[i-1], NULL, my_thread, &thread_args[i]);
    }
    for(i=0; i< NTHREADS; i++){
        rc=pthread_join(threads[i], NULL);
    }
}
```

```
$> ./test_threads2
Thread 1: 0
Thread 2: 0
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 2: 3
Thread 1: 3
Thread 2: 4
Thread 1: 4
$>
```

Et si on passe tout simplement i?

Synchronisation

# Exemple

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NTHREADS 2

void *my_thread (void * arg) {
    int i;
    int n = *((int *) arg);
    for (i = 0 ; i < 5 ; i++) {
        printf ("Thread %d: %d\n", n, i);
        /* On peut enlever les sleep */
    }
    pthread_exit (0);
}

void main () {
    pthread_t threads[NTHREADS];
    int rc, i;
    int thread_args[NTHREADS+1];

    for(i=1; i<= NTHREADS; i++){
        thread_args[i]=i;
        rc=pthread_create(&threads[i-1], NULL, my_thread, &i);
    }
    for(i=0; i< NTHREADS; i++){
        rc=pthread_join(threads[i], NULL);
    }
}
```

```
$> ./test_threads3
Thread 0: 0
Thread 0: 0
Thread 0: 1
Thread 0: 1
Thread 0: 2
Thread 0: 2
Thread 0: 3
Thread 0: 3
Thread 0: 4
Thread 0: 4
$>
```

- $i$  est visible à tous les threads
- le processus a déjà modifié la valeur de  $i$  quand les threads y accèdent...

On nécessite d'un mécanisme d'exclusion mutuelle pour l'accès aux variables globales

# Mutex (verrou)

- Un verrou est utilisé par le threads quand ils ont besoin d'accéder à une ressource partagée (variable globale)
- Fonctionnement:
  - Un thread prend le verrou
  - Il mène les opérations “critiques” (sur la ressource partagée)
    - Les autres threads ne peuvent pas prendre le verrou avant que le thread lâche le verrou
  - Quand il a fini les opérations “critiques”, il lâche le verrou
- Un verrou pour chaque ressource

# Mutex (version pthread)

- Déclarer et initialiser (par défaut) un verrou:

```
pthread_mutex_t nom_verrou = PTHREAD_MUTEX_INITIALIZER;
```

- Verrouiller:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- Déverrouiller:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

# Exemple

```
int sum=0;

void *my_sum(void * arg) {
    int n=0;
    for(n=0; n<10; n++) { sum=sum+1; fprintf(stderr, "Sum: %d\n", sum);}
    pthread_exit (0);
}

void *my_prod(void * arg) {
    int n=0;
    for(n=0; n<5; n++) {sum=sum*2; fprintf(stderr, "Prod: %d\n", sum);}
    pthread_exit (0);
}

void main () {
    pthread_t th1,th2;
    int rc;
    rc=pthread_create(&th1, NULL, my_sum, NULL);
    rc=pthread_create(&th2, NULL, my_prod, NULL);

    rc=pthread_join(th1, NULL);
    rc=pthread_join(th2, NULL);
    printf ("Res: %d\n", sum); //soit 10*32, soit 0+10
}
```

Sections critiques

\$> ./test\_threads4

Sum: 1  
Sum: 3  
Sum: 4  
Sum: 5  
Sum: 6  
Sum: 7  
Sum: 8  
Sum: 9  
Sum: 10  
Sum: 11  
Prod: 2  
Prod: 22  
Prod: 44  
Prod: 88  
Prod: 176  
Res: 176  
\$>



```
pthread_mutex_t var_lock = PTHREAD_MUTEX_INITIALIZER; /* verrou */

int sum=0;

void *my_sum(void * arg) {
    int n=0;
    pthread_mutex_lock(&var_lock);
    for(n=0; n<10; n++) { sum=sum+1; fprintf(stderr, "Sum: %d\n", sum);
    pthread_mutex_unlock(&var_lock);
    pthread_exit (0);
}

void *my_prod(void * arg) {
    int n=0;
    pthread_mutex_lock(&var_lock);
    for(n=0; n<5; n++) {sum=sum*2; fprintf(stderr, "Prod: %d\n", sum);
    pthread_mutex_unlock(&var_lock);
    pthread_exit (0);
}

void main () {
    pthread_t th1,th2;
    int rc;
    rc=pthread_create(&th1, NULL, my_sum, NULL);
    rc=pthread_create(&th2, NULL, my_prod, NULL);

    rc=pthread_join(th1, NULL);
    rc=pthread_join(th2, NULL);
    printf ("Res: %d\n", sum); //on attend: 32*10
}
```

# Example

```
$> ./test_threads4
Sum: 1
Sum: 2
Sum: 3
Sum: 4
Sum: 5
Sum: 6
Sum: 7
Sum: 8
Sum: 9
Sum: 10
Prod: 20
Prod: 40
Prod: 80
Prod: 160
Prod: 320
Res: 320
$>
```

## **8. Gestion de la Mémoire**

# La mémoire principale

- Fonctionne comme un grand tableau linéaire. Une adresse est un index dans ce tableau
- Tout emplacement de la mémoire peut être lu ou écrit
- Un emplacement est défini par son adresse
- Selon la taille (en bits) des adresses, on a une quantité de mémoire adressable
  - Espace adressable =  $2^{\text{taille des adresses}}$
  - 4 Gio en 32 bits, 16 Eio en 64 bits
- Exemple : 32 bits pour coder une adresse, mémoire adressable de  $2^{32}$  octets
- Mémoire physique : quantité de mémoire réellement existante
- Certaines adresses ne peuvent pas être obtenues (p.e. réservées au système)  $\Rightarrow$  Mécanisme de translation de pages



# Structuration de la mémoire

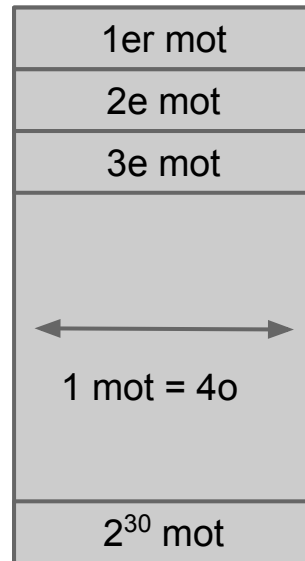
- Adresses sur  $n$  bits, mots de  $x$  octets
- Notation hexadécimale pour les adresses
- $2^n$  octets adressables, soit  $2^n / x$  mots
- Une opération sur un mot-mémoire doit se faire avec une adresse multiple de sa taille
- Sinon, erreur dite d'alignement

0x00000000

0x00000004

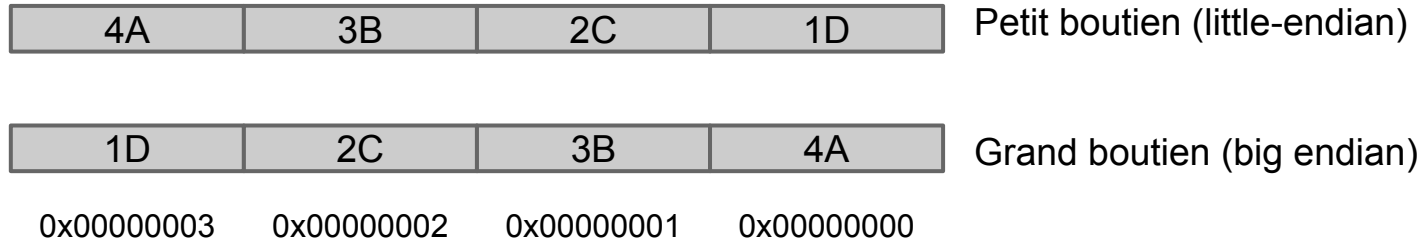
0x00000008

0xFFFFFFFFC



# Grand et petit boutien

- Exemple Valeur 0x4A3B2C1D, adresse 0x00000000 ?

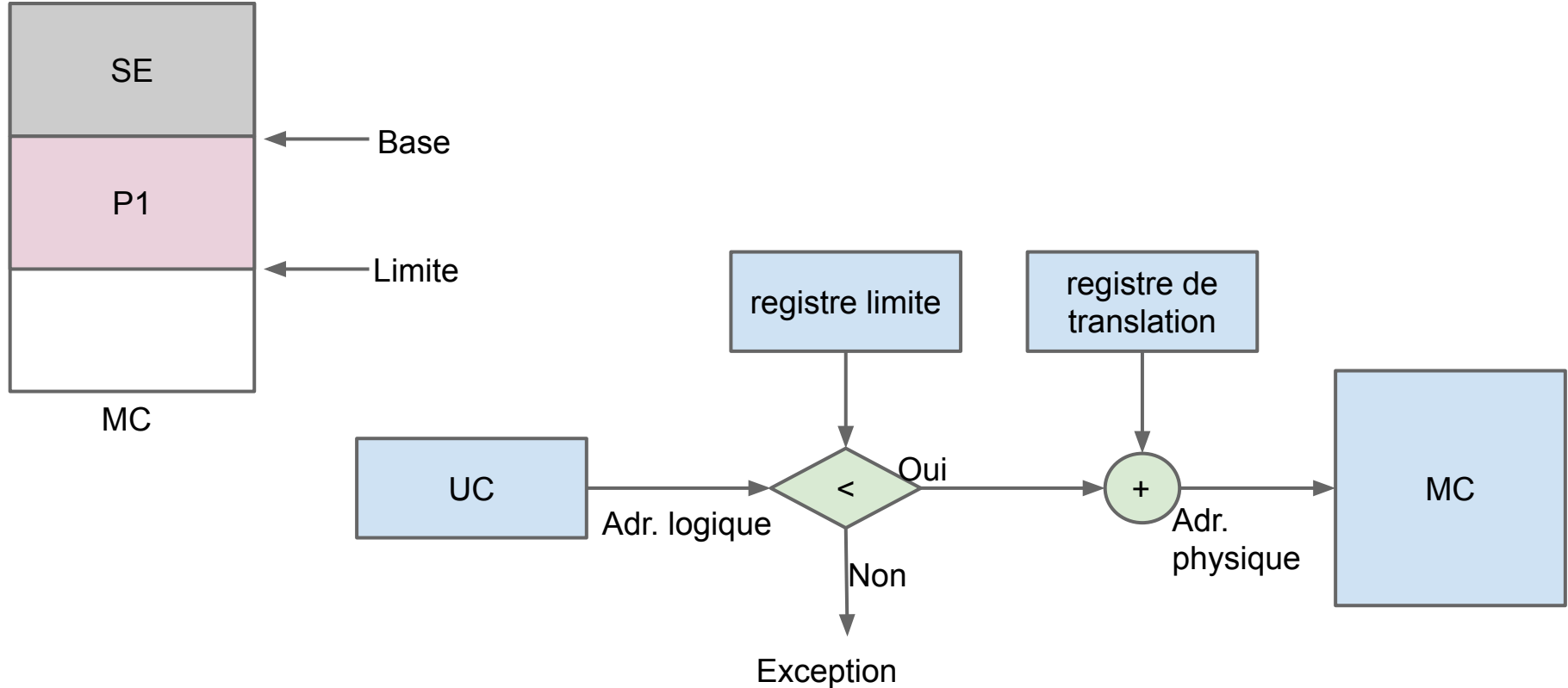


# Translation

## adresse logique - physique

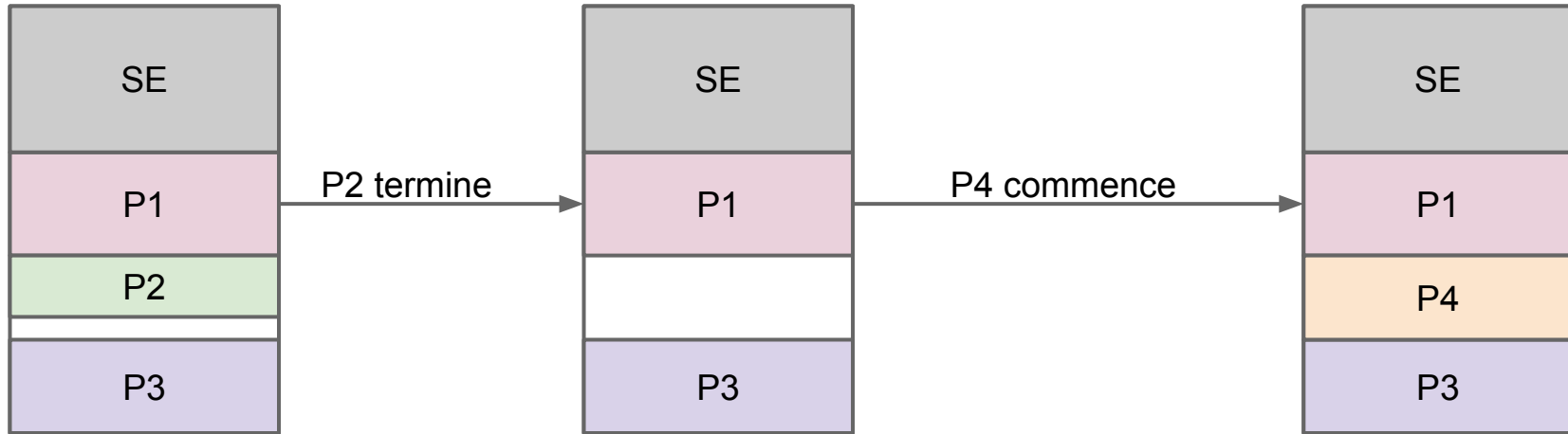
- Deux options:
  - Définitivement au moment du chargement du programme:
    - translation **statique**
      - Lors de l'exécution, aucune allocation n'a lieu
      - En général, utilisée pour le segment texte et données
  - Au cours de l'exécution
    - translation **dynamique**
      - Utilisée pour la pile (automatique) et le tas (utilisateur)
      - Allocation sur le tas:
        - malloc, free (langage C)
        - new, delete (C++)

# Résolution d'adresse

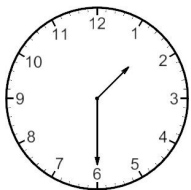
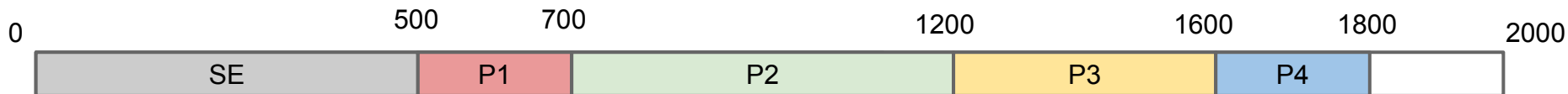


# Allocation contiguë

- On cherche de charger un processus dans un espace contiguë



# Allocation contiguë



t=0

t=2

t=3

t=5

t=7 P5 prêt mais pas d'espace

t=9 P3 fini mais encore pas assez d'espace

t=10 P1 fini -> assez d'espace pour P5 mais

pas contiguë

t=15 P2 fini, P5 a son espace alloué

Fragmentation  
Externe

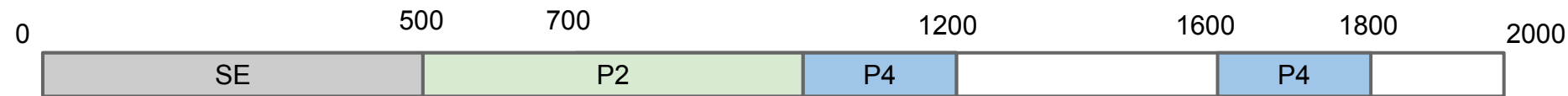
Exemple (avec 500K pour le SE, RAM 2M):

Proc	Mémoire	Arrivée	Fin
P1	200K	0	10
P2	500K	2	15
P3	400K	3	9
P4	200K	5	17
P5	650k	7	23

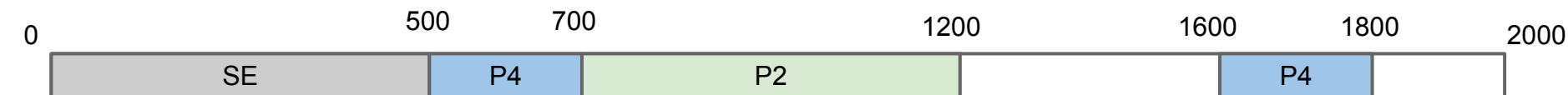
# Fragmentation Externe

- Répartition de l'espace libre en partitions de tailles insuffisantes
  - $\Rightarrow$  mémoire sous utilisée
  - $\Rightarrow$  perte de temps
- Solution: compactage (defragmentation)
  - Recopier les parties occupées pour fusionner les fragments libres

Ex en  $t=10$  deux possibilités:



700K à copier



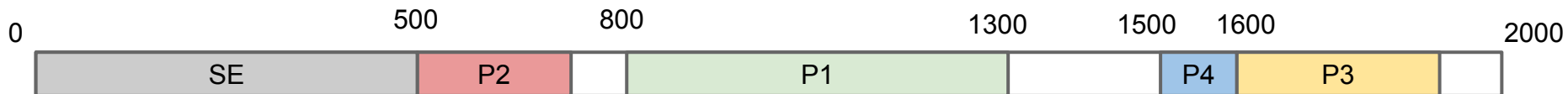
200K à copier

# Algorithmes d'allocation

- First Fit: premier trouvé
  - Best Fit: au plus petit reste - au meilleur choix
  - Worst Fit: au plus grand reste - au pire choix
- 
- First Fit plus rapide mais moins efficace
  - Un algorithme optimal n'existe pas: ça dépend de la situation de la mémoire



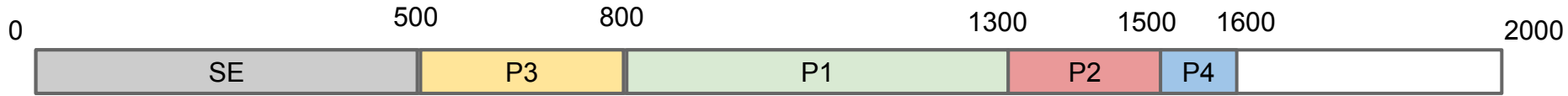
# First-Fit



Exemple (avec 500K pour le SE, RAM 2M):

Proc	Mémoire	Arrivée	Fin
P1	500K	0	10
P2	200K	2	15
P3	300K	3	9
P4	100K	0	17

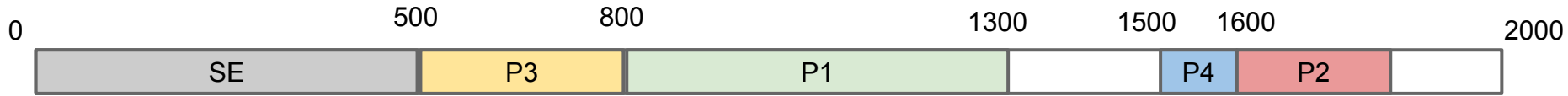
# Best-Fit



Exemple (avec 500K pour le SE, RAM 2M):

Proc	Mémoire	Arrivée	Fin
P1	500K	0	10
P2	200K	2	15
P3	300K	3	9
P4	100K	0	17

# Worst-Fit



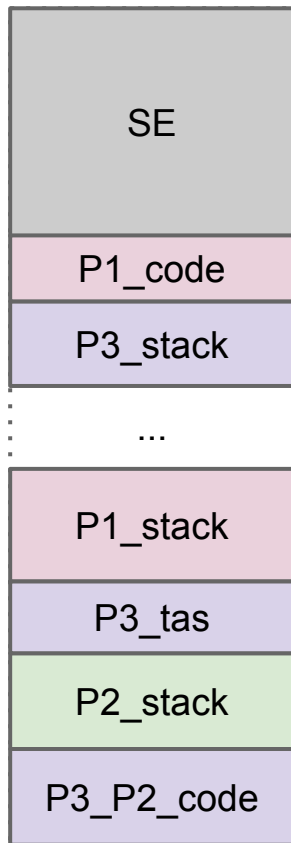
Exemple (avec 500K pour le SE, RAM 2M):

Proc	Mémoire	Arrivée	Fin
P1	500K	0	10
P2	200K	2	15
P3	300K	3	9
P4	100K	0	17

# Segmentation

- Diviser l'image du processus en parties non-contiguës
- Permet au programmeur de voir un processus en mémoire sous la forme de plusieurs espaces d'adressages : les *segments*
  - Chaque composantes du processus (pile, code, tas, données, etc.) peut avoir son propre segment ;
  - On peut mettre des droits sur l'accès aux segments ;
  - La taille des segments est variable (tas, pile);
  - Un segment peut être partagé par plusieurs processus.

# Segmentation



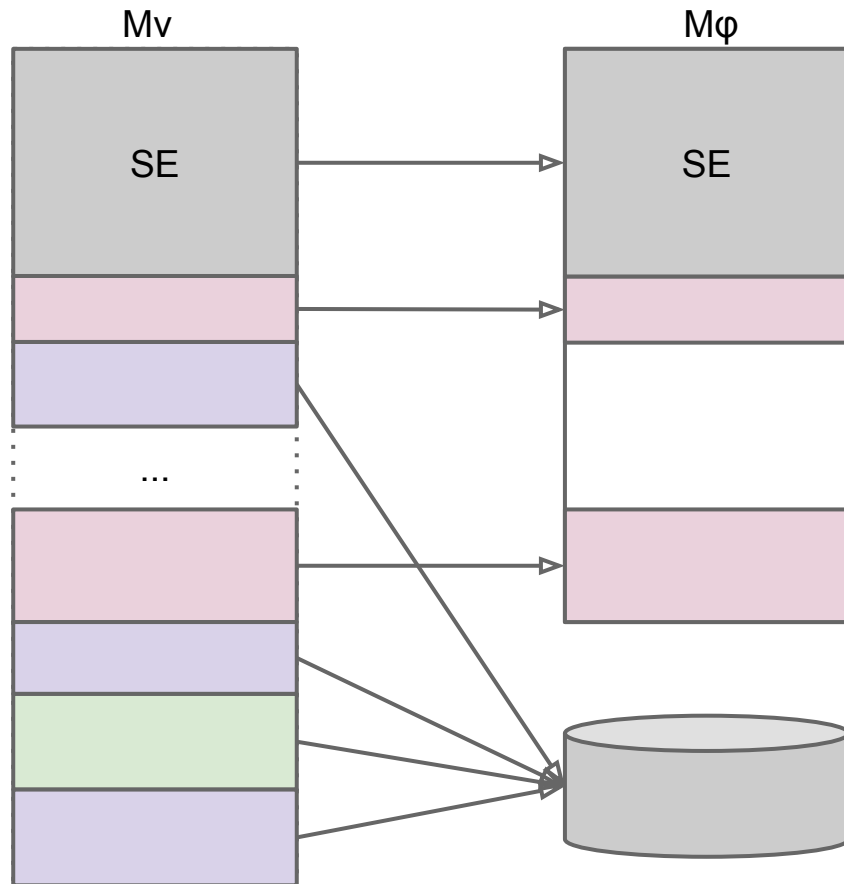
- Les segments sont numérotés
- Adresse logique = (numéro segment, décalage)=(s,d)
- Descripteur d'un segment:
  - Adresse base
  - Taille (limite)
  - Type d'accès
- Table des segments en MC
  - Deux accès à MC pour lire données! Ralentissement!
- Mais on gagne en espace:
  - Plusieurs processus, même programme -> partage segment du code
  - Segment de données communs -> partage des données (threads)

- On a besoin de beaucoup de mémoire physique ( $M_\phi$ ) dans les systèmes multi-processus
- Mais souvent la plupart des processus ne sont pas actifs
- Solution: **Mémoire virtuelle** ( $M_v$ )
- En général Taille  $M_v >$  Taille  $M_\phi$

Ex: Les vieux Windows 32 bit ont un espace d'adressage de 4Gio (2Gio utilisateur et 2Gio système), quand leur RAM était habituellement plus petite.

Pour info: Windows 7: 8Tio utilisateur et système, Windows 8 et Linux-64bit: 128Tio; OSX: 4Gio (32bits), 18Eio (64bits)

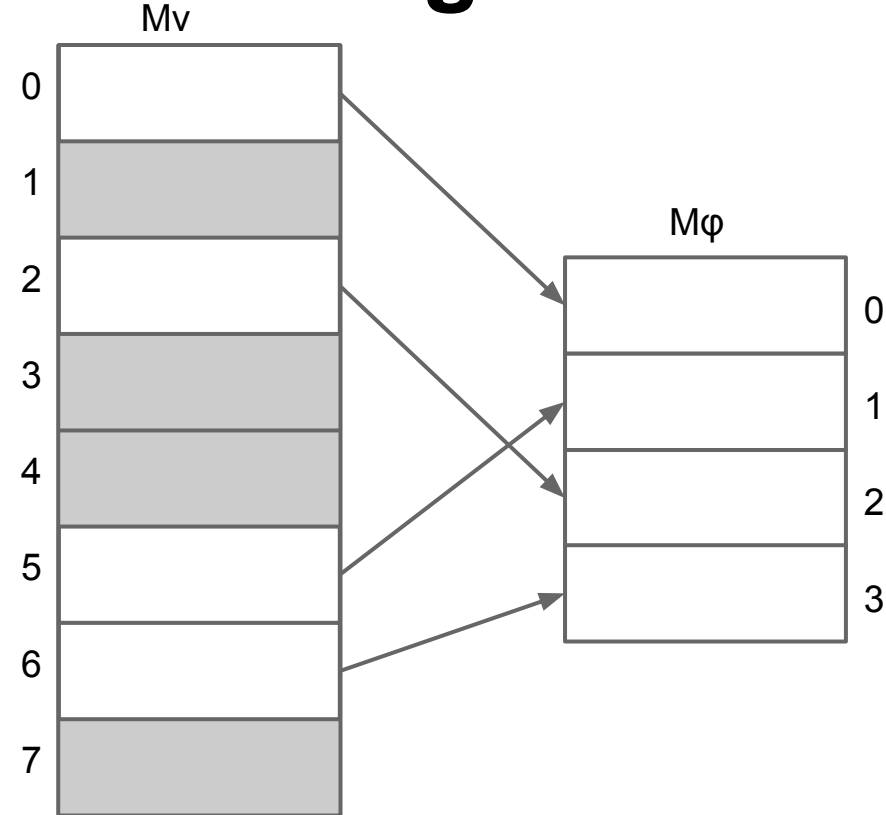
# Mémoire Virtuelle



# Pagination

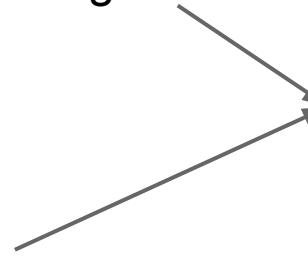
- Pour simplifier la translation, on choisi de découper  $M_v$  et  $M_\phi$  en blocs de même taille (fixe):
  - **Pages** pour la  $M_v$
  - **Cadres** pour la  $M_\phi$
- Table des pages**
  
nécessaire pour résoudre les adresses

P	C
0	0
1	-
2	2
3	-
4	-
5	1
6	3
7	-



# Taille d'une page

- Comment choisir la taille d'une page B ?
  - $B \in \{512o, 1Kio, 2Kio, \underline{4Kio}, \underline{8Kio}, 16Kio\}$
- B petite:
  - Moins de fragmentation
  - Taille de la table des pages augmente
- B grande:
  - Fragmentation augmente
  - Taille de table inferieure
  - Transfert MS  $\leftrightarrow$  MC plus efficace

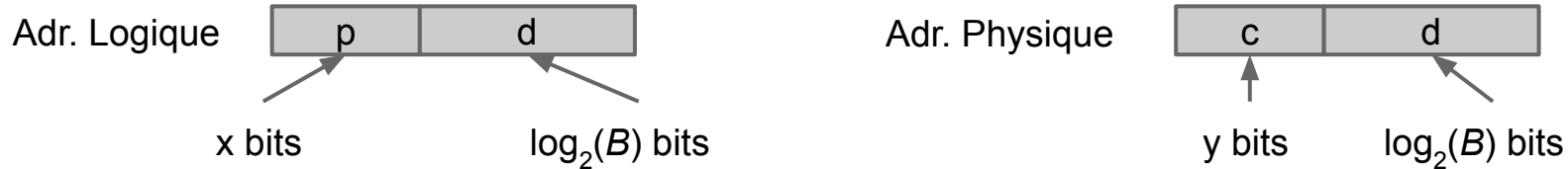


#Pages = taille(Mv)/B



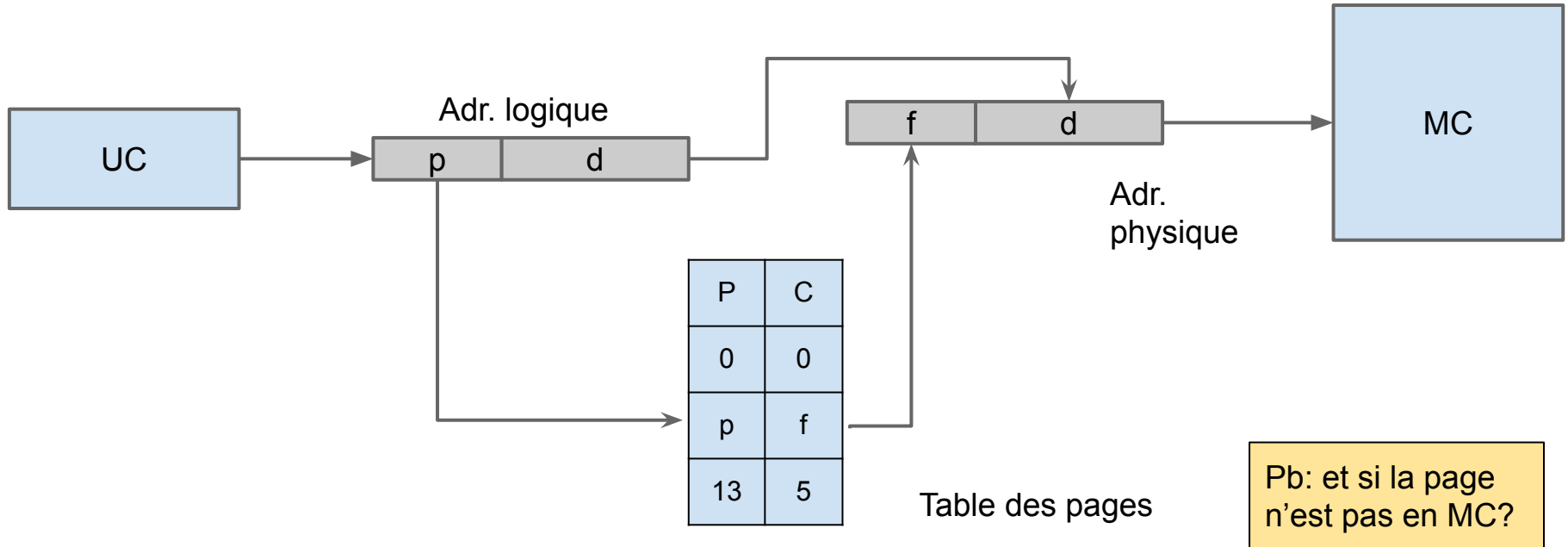
# Résolution d'adresse

- Numéro de page, décalage (p,d)  $\leftrightarrow$  Numéro de cadre, décalage (c,d)



- Exercice: Espace d'adressage Mv de 4Gio, B=4Kio, Mø de 2Mio
  - Trouver la taille de p, c et d:
    - $d = \log_2(4 * 2^{10}) \text{ bits} = \log_2(2^{12}) = 12 \text{ bits} \Rightarrow d \text{ sur } 12 \text{ bits}$
    - Num pages:  $4\text{Gio}/4\text{Kio} = 2^{30}/2^{10} = 2^{20} \text{ pages} \Rightarrow p \text{ sur } 20 \text{ bits}$
    - Num cadres:  $2\text{Mio}/4\text{Kio} = 2^{20}/(2 * 2^{10}) = 2^9 \text{ pages} \Rightarrow c \text{ sur } 9 \text{ bits}$
  - Si le processus P1 occupe 640Kio, il va avoir besoin de combien de pages?
    - $640\text{Kio}/4\text{Kio} = 640/4 = 160 \text{ pages}$

# Résolution d'adresse



# Défaut de page

- La page référencée n'est pas en MC => Défaut de page
  - Trouver un cadre libre ou en libérer un
  - Copier la page
  - MAJ de la table des pages
  - Reprendre l'exécution à la même instruction
- Remplacement de page:
  - Si le cadre choisi est modifié, on copie la page à remplacer dans le swap
  - On copie la page demandé dans la MC (cadre)

# Algorithmes de remplacement

- Comment choisir la page à remplacer?
- Deux options:
  - **FIFO**: remplacer la page la plus ancienne
  - **LRU**: remplacer la page qui a été utilisée le moins récemment (Least Recently Used)

# Remplacement FIFO

Séquence des pages demandées:

0	1	0	2	0	3	1	2	1	4	3	2
---	---	---	---	---	---	---	---	---	---	---	---

3 cadres:

<b>0*</b>	0	<b>0</b>	0	<b>0</b>	<b>3*</b>	3	3	3	3	<b>3</b>	3
-	<b>1*</b>	1	1	1	1	1	1	1	<b>4*</b>	4	4
-	-	-	<b>2*</b>	2	2	2	2	2	2	2	<b>2</b>

taux de défaut: 5/12  
(2/8 après démarrage)

# Remplacement LRU

Séquence des pages demandées:

0	1	0	2	0	3	1	2	1	4	3	2
---	---	---	---	---	---	---	---	---	---	---	---

3 cadres:

<b>0*</b>	0	<b>0</b>	0	<b>0</b>	0	0	<b>2*</b>	2	2	<b>3*</b>	3
-	<b>1*</b>	1	1	1	<b>3*</b>	3	3	3	<b>4*</b>	4	4
-	-	-	<b>2*</b>	2	2	<b>1*</b>	1	<b>1</b>	1	1	<b>2*</b>

taux de défaut: 9/12  
(6/8 après démarrage)

# Pagination dans Linux

- Table des pages: structure à arbre (plus efficace)
- Taille d'une page: habituellement 4Kio (défini in PAGE\_SIZE)
- Information sur les pages dans la structure **page**:
- Remplacement pseudo-LRU

Informations de la table des pages: si en MC ou MS, si écriture possible ou pas, si utilisateur ou noyau, etc.

```
struct page {  
    page_flags_t    flags;  
    atomic_t        _count;  
    atomic_t        _mapcount;  
    unsigned long   private;  
    struct address_space *mapping;  
    pgoff_t         index;  
    struct list_head lru;  
    void            *virtual;
```

Nombre de processus qui utilisent la page; si `_count=0` alors la page peut être libérée

Adresse virtuelle de la page

Pagination dans MacOSX:

<https://developer.apple.com/library/mac/documentation/Performance/Conceptual/ManagingMemory/Articles/AboutMemory.html>

# Mémoire virtuelle en Linux

- Voir les régions de mémoire virtuelles d'un processus:

`cat /proc/[PID]/maps`

adresses	perm.	décalage	périph	inoeud	chemin d'accès
00400000-00401000	r-xp	00000000	00:18	78127628	
/users/buscaldi/Teaching/M3101/mem/test2					
06000000-00601000	rw-p	00000000	00:18	78127628	
/users/buscaldi/Teaching/M3101/mem/test2					
00a81000-00aa2000	rw-p	00000000	00:00	0	[heap]
7f826bf7c000-7f826bf91000	r-xp	00000000	08:01	1180164	/lib/x86_64-linux-gnu/libgcc_s.so.1
7f826bf91000-7f826c191000	---p	00015000	08:01	1180164	/lib/x86_64-linux-gnu/libgcc_s.so.1
7f826c191000-7f826c192000	rw-p	00015000	08:01	1180164	/lib/x86_64-linux-gnu/libgcc_s.so.1
7f826c192000-7f826c193000	---p	00000000	00:00	0	
...					
7f826e51b000-7f826e520000	rw-p	00000000	00:00	0	
7f826e95d000-7f826e95e000	rw-p	00000000	00:00	0	
7fff9c8f2000-7fff9c913000	rw-p	00000000	00:00	0	[stack]

Code

données



# Mmap

- On peut créer une région de mémoire:
  - Vide (stack, BSS)
  - Comme projection d'un fichier

- Fonction **mmap**:

MAP\_SHARED si on souhaite partager la région avec des autres processus  
MAP\_PRIVATE en cas contraire  
plus MAP\_ANONYMOUS (avec l'opérateur | ) si pas de fichier

offset initial du fichier *fd*

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

Adresse désiré (si 0, c'est le SE qui choisit)

Taille de la région à créer

La protection de mémoire qu'on souhaite:  
PROT\_NONE  
PROT\_EXEC  
PROT\_WRITE  
PROT\_READ

descripteur du fichier à référencer

# Projection d'un fichier avec mmap()

- Exemple de chargement en mémoire virtuelle d'un fichier "data.txt" composé par 4 octets (caractères)

```
...  
#include <sys/types.h>  
#include <sys/mman.h>  
#include <sys/fcntl.h>  
  
void main() {  
    int i,fd;  
    char* buf;  
    fd = open("data.txt",O_RDONLY);  
    buf = mmap (0, 4, PROT_READ, MAP_PRIVATE, fd, 0);  
    for (i=0; i<4; i++) printf ("%c\n",buf[i]);  
}
```

# Partage de mémoire entre processus

- On peut utiliser mmap pour partager une région de mémoire entre processus différents

```
...
#include <sys/types.h>
#include <sys/mman.h>
#define N 100

void main() {
    int *result_ptr = mmap (0, 4, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, 0, 0);
    pid_t pid = fork();
    int i=0,sum=0;
    if (pid==0) { // FILS
        for (i=1; i<=N; i++) sum+=i;
        *result_ptr = sum;
    } else { // PERE: attend le résultat
        wait(0);
        printf("result=%d\n", *result_ptr);
    }
}
```

# Conséquences

- La mémoire virtuelle est partagée par les processus
- Ils peuvent communiquer par mémoire partagée
- Attention aux accès concurrents!
  - Utilisation de verrous (mutex)