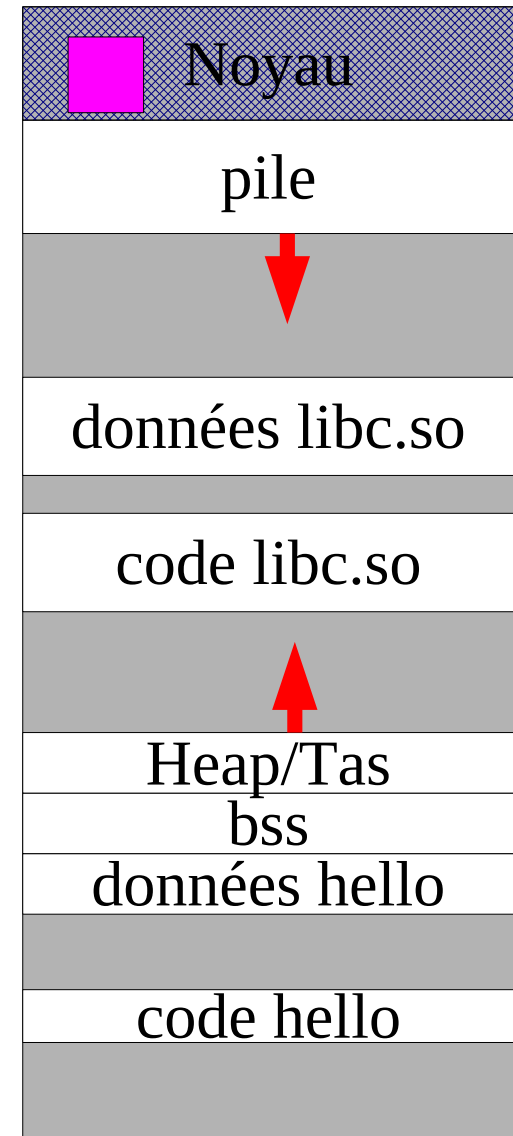


Informatique Embarquée M2 / 2014

Processus, Threads,
Synchronisation

Processus

- Code, données, tas, pile
 - Librairies
- Contexte système
 - Espace d'adressage,
 - Identificateurs,
 - Fichiers ouverts, répertoires
 - Utilisateur, groupe,
 - Signaux (handlers, reçus, ...)
 - Accounting (temps, I/O's,...)



Espace Adressage

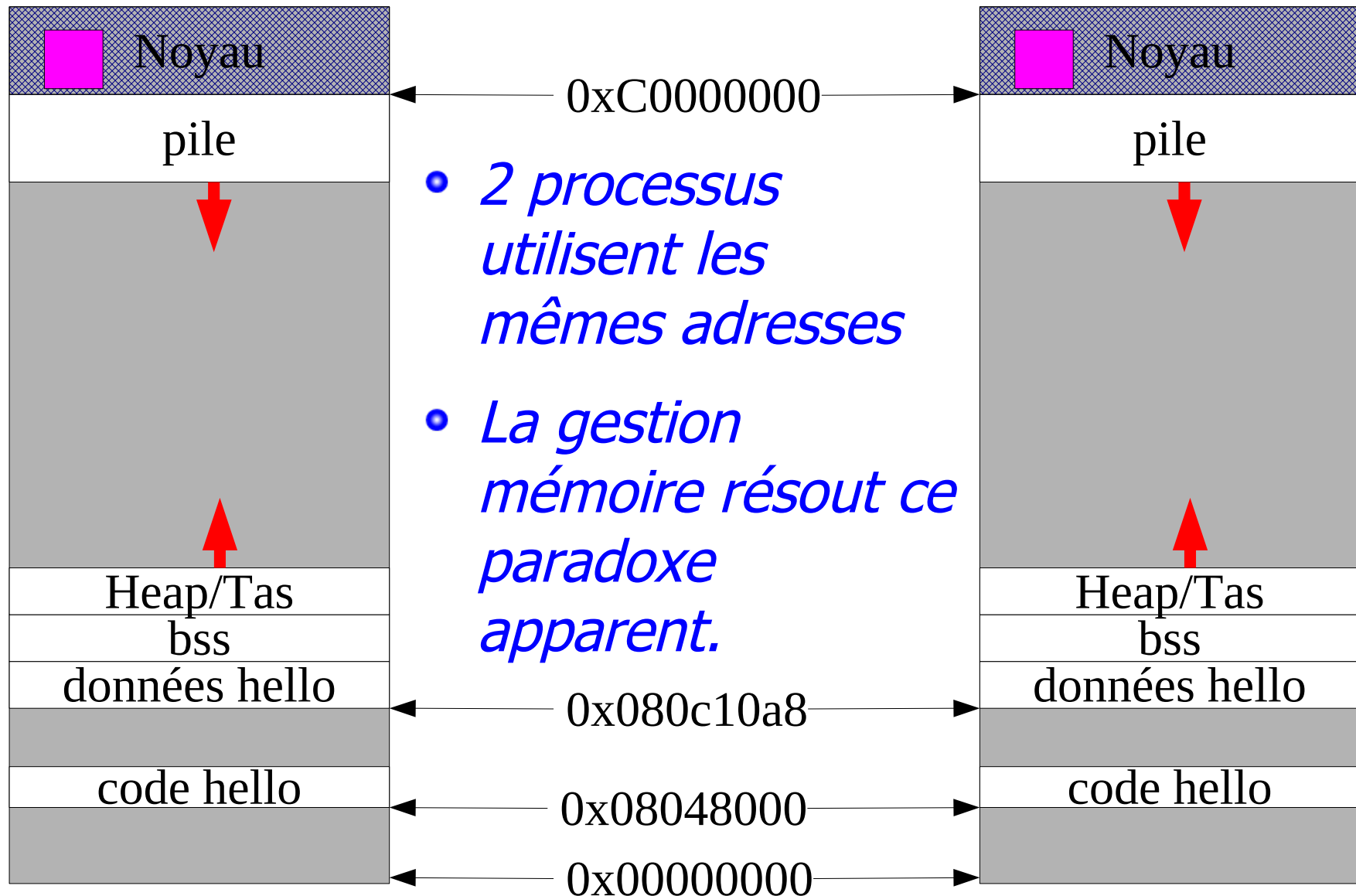
- Ensemble d'adresses « accessibles » par un processus. En général: adresses « octets »
 - Question: Pourrait-on facilement porter un programme écrit en C sur une machine ayant des adresses mots (2 octets)?
- Espace « théorique » atteignable par un processus en C?
 - Sur une machine 32 bits : $2^{32} \Rightarrow$ 4 Giga octets
 - Sur une machine 64 bits : dépend taille adresse!

Espace Adressage

- Varie suivant le système sous-jacent:
- Espace d'adressage virtuel
 - Nécessite une machine physique ayant une MMU
 - ▶ Memory Management Unit (MMU) est une unité de la machine physique qui gère l'adressage virtuel.

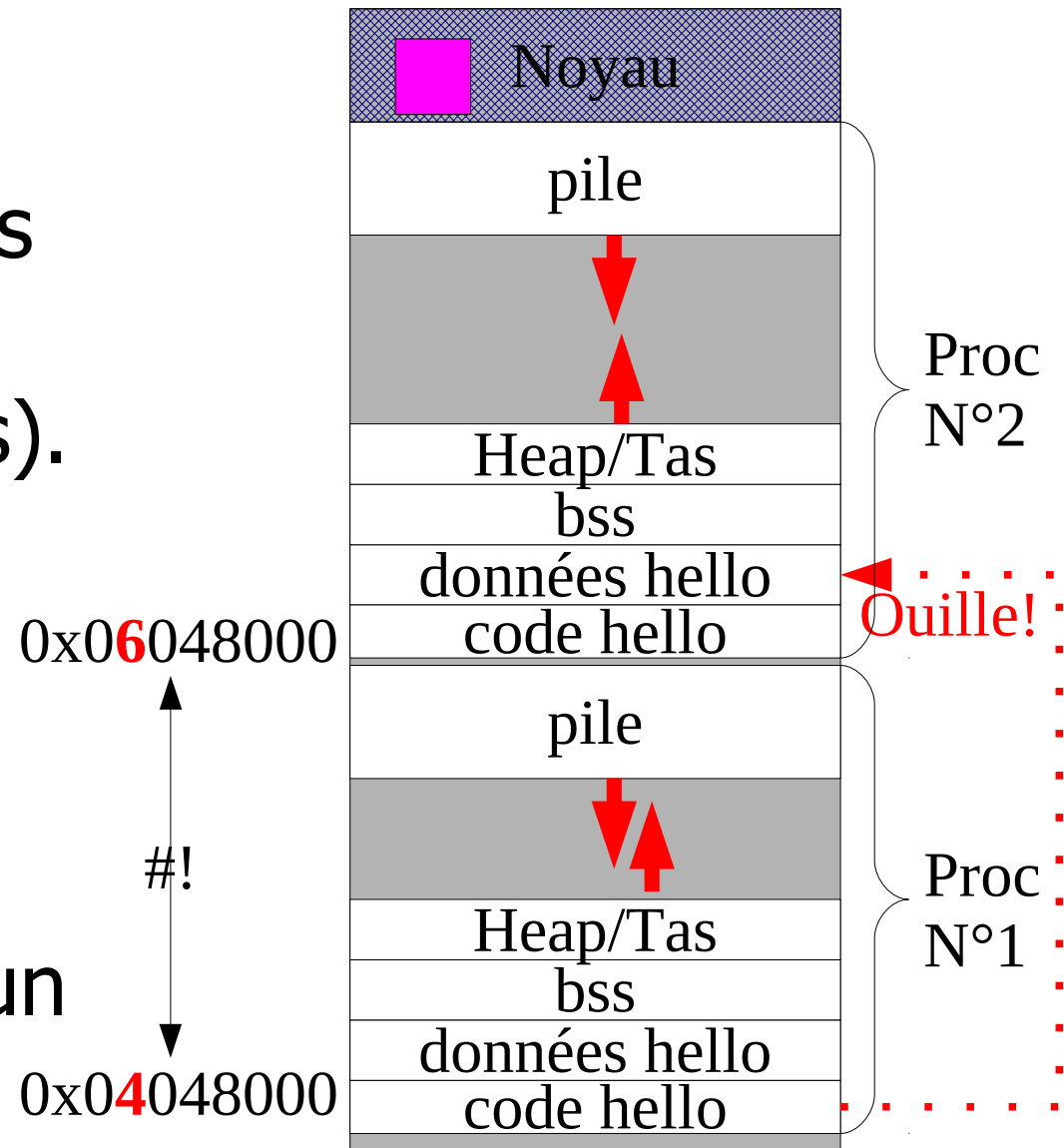
Espace d'adressage Esp. virt. → Esp. phys.

Mémoire virtuelle



Mémoire physique

- Un seul espace d'adressage, partagé par tous les processus (et leurs zones de mémoires respectives).
- Souvent, il n'y a pas d'isolation entre processus. Un processus peut lire / écrire les données d'un autre...



Espace d'adressage Linux

- « Voir » l'espace d'un processus
 - `# cat /proc/pid/maps`
 - ▶ Voir la page « man » : `man proc`
- « Voir l'espace requis par un programme »
 - `# readelf -l hello`
- Changer les adresses par défaut
 - Options de l'éditeur de liens
 - ▶ Script pour des opérations complexes

Création d'un processus

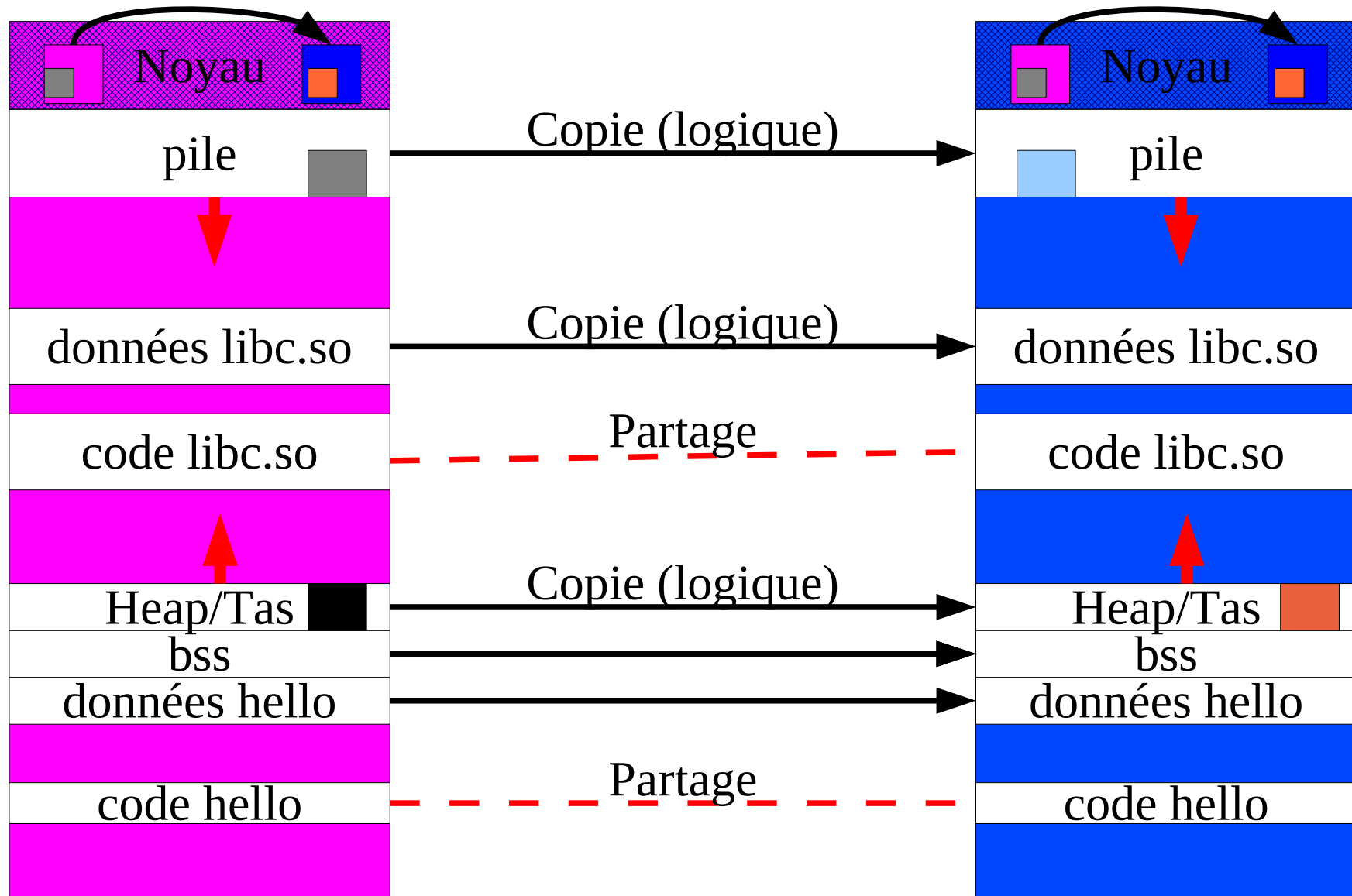
- Unix / Linux :
 - Par duplication d'un processus existant
 - `fork()` // 1 appel, 2 retours!

```
if ((pid = fork()) == 0) {  
    // le nouveau processus: fils  
    execlp(file, ...); // nouveau prog  
} else {  
    // le processus pré-existant: père  
    airtid(pid, ...);  
}
```

Création d'un processus

- Autre possibilité :
 - `vfork()` // sera étudié plus tard
- Posix :
 - `posix_spawn(file, ...)` // peu utilisé
 - Sorte de `fork` + `exec` en une étape
 - Arguments permettent de manipuler le contexte comme on peut le faire « manuellement » entre la sortie du `fork` et l'appel à `exec` dans le processus fils.

Fork



Fork et espace d'adressage

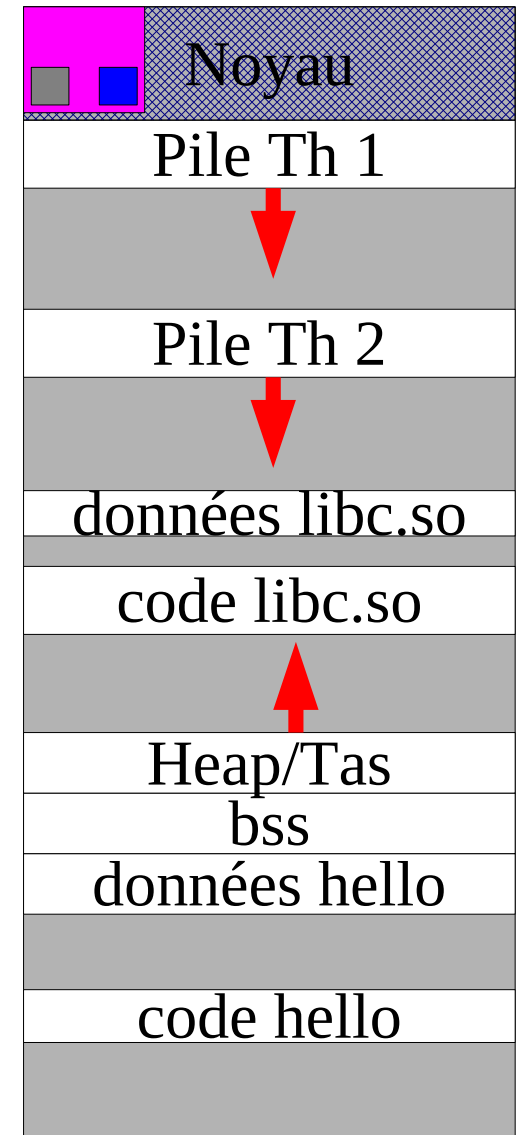
- Le nouveau processus reçoit une copie
 - Mêmes adresses pour code, données, tas, pile...
- Donc....
- Pas possible si pas d'espace d'adressage virtuel...
- Pas possible si la machine ne dispose pas de MMU.
 - Dans ces deux cas, il n'y a pas d'indirections entre les adresses vues par le processus et les adresses en mémoire physique.
- Sur les machines sans MMU, il faut pouvoir créer les processus autrement que par « clonage »
 - Par exemple: `posix_spawn`

Processus « trop lourds »

- Il faut d'autres moyens de créer des « activités » permettant de traiter de requêtes :
 - Un processus par requête, trop coûteux, trop gourmand en ressources
 - ▶ A la création
 - ▶ Lors de l'exécution : changement de contexte
 - Trouver un moyen d'avoir plusieurs exécutions en parallèle au sein du même processus.

pthread_create

- Espace adressage partagé
- Les piles ne sont pas protégées les unes des autres
- Handler de signaux peuvent être exécutés par n'importe quelle thread
 - Masquer les signaux dans les threads qui ne veulent pas être perturbées

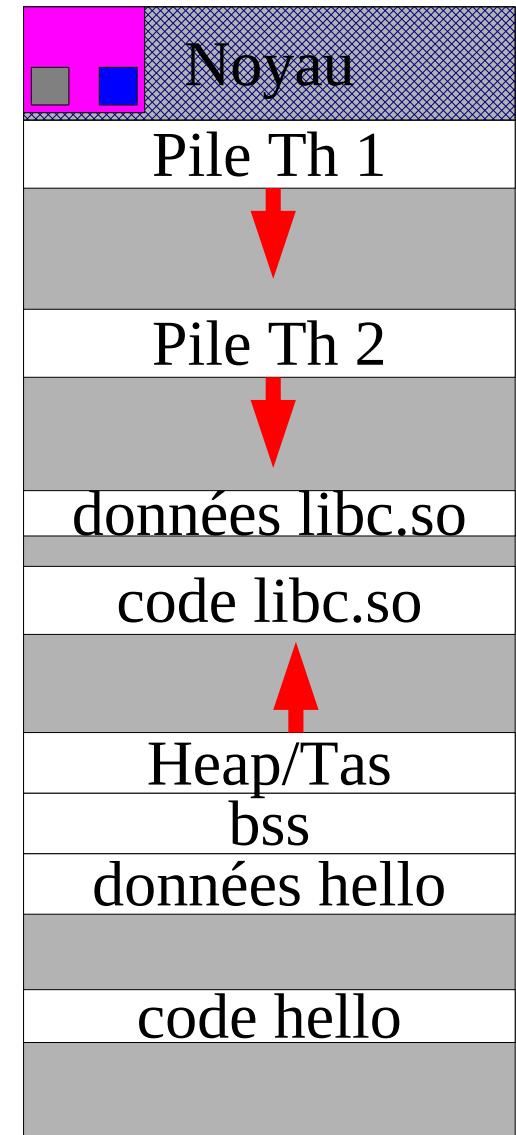


Manipuler les threads

- `pthread_create(thout, attrin, fnin, argin)`
 - `pthread_attr_init`: Scheduling policy, parameters...
- `pthread_exit(void *valin)`
 - Attention, `exit()` termine le processus (toutes les threads)
- `pthread_join(thin, void **valout)`
 - Attendre la fin d'une thread en particulier (\sim `wait`)
- `pthread_self()`
 - Qui suis-je? Similaire à `getpid()`

pthread_create

- Espace adressage partagé
- Les piles ne sont pas protégées les unes des autres
- Handler de signaux peuvent être exécutés par n'importe quelle thread
 - Masquer les signaux dans les threads qui ne veulent pas être perturbées



Une fonction incrémente une variable

```
#define MAXLOOP 1000000000
static int value;

void* my_func(void *arg)
{
    int i;
    for (i =0; i < MAXLOOP; i++) {
        value++;
    }
    return NULL;
}
```

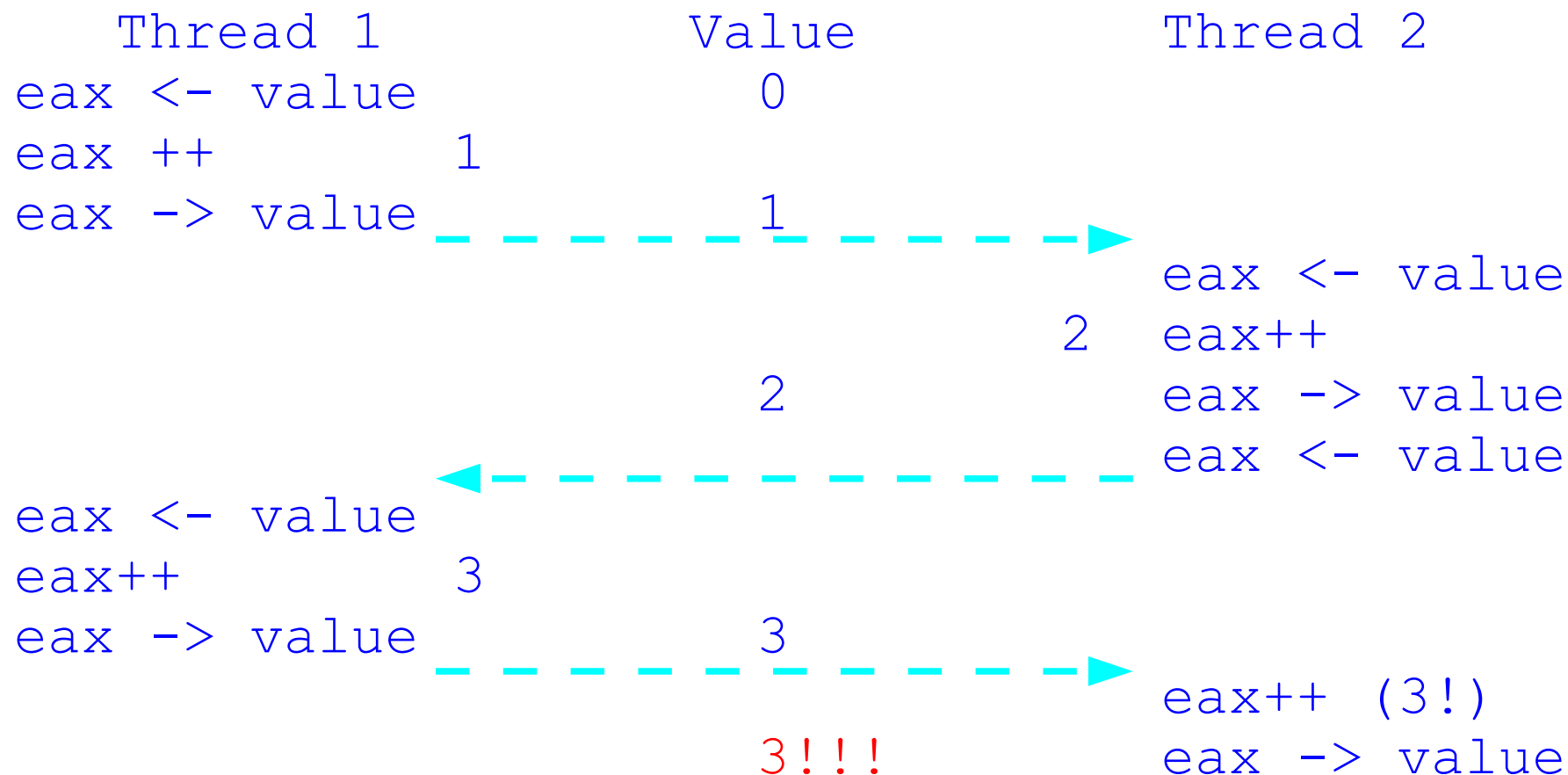
Code généré... (sans -O2)

```
my_func:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $16, %esp
    movl     $0, -4(%ebp) # i =0
    jmp      .L2

.L3:                                # Code associé à value++;
    movl     value, %eax # eax <- value
    addl     $1, %eax    # eax ++
    movl     %eax, value # eax -> value
    addl     $1, -4(%ebp) # i++

.L2:
    cmpl     $999999999, -4(%ebp)
    jle      .L3
```

Dépendance sur l'ordonnement



On appelle cette fonction depuis 2 threads simultanément

```
main()
{
    pthread_t th1, th2;
    int res;

    res = pthread_create(&th1, NULL, my_func, NULL);
    res = pthread_create(&th2, NULL, my_func, NULL);

    res = pthread_join(th1, NULL);
    res = pthread_join(th2, NULL);

    printf("Value: %d\n", value);
}
```

Résultats à l'exécution ?

```
P# for i in 1 2 3 4 5 6 7 8 9 10  
> do  
> ./2threads  
> done
```

Value: 162316776

Value: 168188618

Value: 166543199

Value: 157071774

Value: 164020788

Value: 151951843

Value: 157690216

Value: 163886857

Value: 180591873

Value: 159500167

200 000 000?

```
# #Système Debian / VirtualBox monoprocasseur!
```

Synchronisation : mutex

- Exclusion mutuelle:
 - Protège l'accès à une section de code
 - Section critique
 - ▶ Acquisition du mutex (bloquant si occupé)
 - ▶ Exécution code section critique
 - ▶ Relâche le mutex (débloque ceux qui attendent)
 - Acquisition / Relâchement
 - ▶ par le même processus
 - ▶ Ou la même thread

Synchronisation : mutex

- Services:
 - `pthread_mutex_init`, `pthread_mutex_lock`
 - `pthread_mutex_unlock`, `pthread_mutex_try_lock`
- Mutex:
 - Ressource (structure mémoire) dans l'espace d'adressage du processus,
 - En mémoire partagée entre 2 processus
- Ne pas confondre
 - Avec les sémaphores (voir plus loin)
 - Avec les sémaphores System V (`semget`)

Synchronisation : mutex

- Propriétés diverses:
 - Intra / Inter processus
 - Récursif, Sur, Détection de dépendance circulaire
 - Héritage de priorité...
 - ▶ Voir les problèmes d'inversion de priorité
 - ▶ Plus loin dans ce cours
- Plus il y a de propriétés, plus l'acquisition coûte cher!
 - On essaye de faire que l'acquisition d'un mutex libre n'entraîne pas d'appel au noyau...
 - Utilisation de « Test and Set » ou supports similaires

Synchronisation : sémaphores

- Le mutex protège une section critique contre des exécutions concurrentes simultanées
 - Permet une sérialisation des traitements.
- Quid si on veut synchroniser des jobs d'impression vers 2 imprimantes?
 - Utilisation de sémaphore

Sémaphore

- Un sémaphore est associé à une valeur
 - Lors de son initialisation,
 - Dans notre cas : 2 (2 imprimantes)
- Pour pouvoir imprimer, il faut faire une opération **P()**, *prendre, puis-je..*, sur le sémaphore...
- Quand on a fini d'imprimer, on fait une opération **V()**, *vendre, vas-y..*, sur le sémaphore

P() et V()

- P()
 - Si $\text{cpt} \leq 0$
 - ▶ Attendre
 - Sinon
 - ▶ Cpt --, continuer
- V()
 - Cpt ++
 - Si quelqu'un en attente, le réveiller (un seul!)
- Voir la page wikipédia en français...

Mutex Posix

```
#include <semaphore.h>
int sem_init(sem_t*s, int psh, uint v);
int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_timedwait(sem_t *sem, struct timespec *t);
```

Named semaphores

- Existent dans l'espace de nommage (« fichiers »)

```
sem_t *sem_open(char *name, int oflag, ...);  
int sem_close(sem_t *sem);  
int sem_unlink(const char *name);
```

Semaphores IPC System V

```
#include <sys/sem.h>

int semget(key_t key, int n, int semflg);

int semop(int semid, struct sembuf *sops,
          size_t nsops);
```

Cas particuliers

- Sémaphore « binaire » : 0 ou 1
 - Ne laisse travailler qu'un seul processus