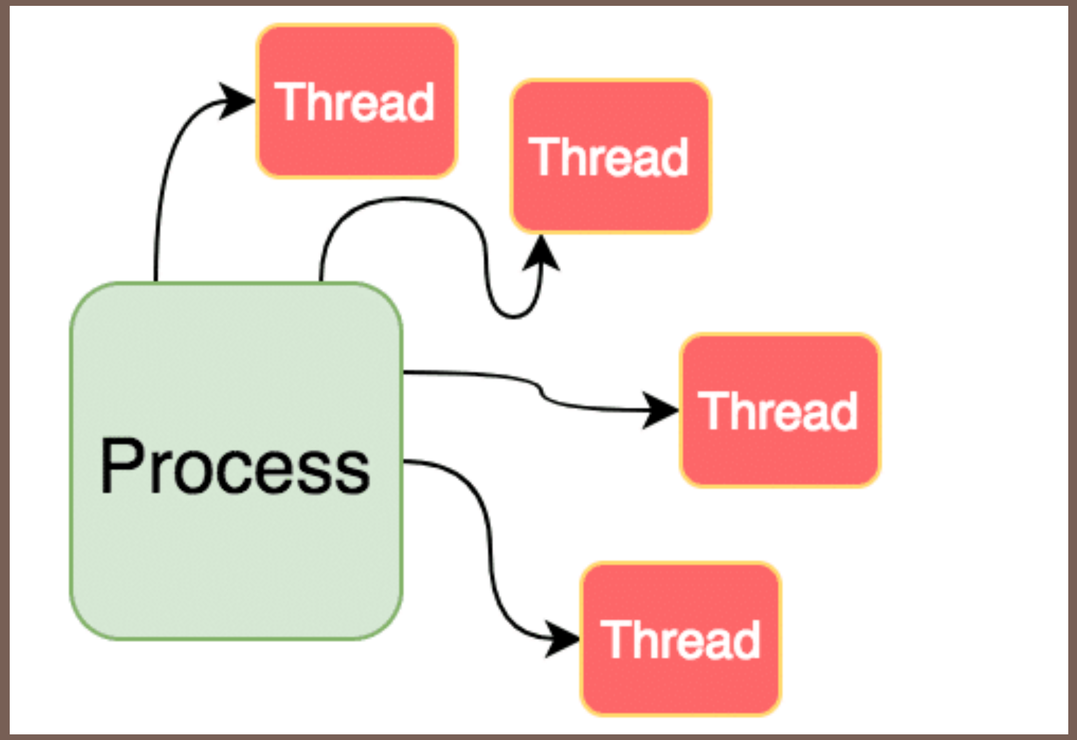
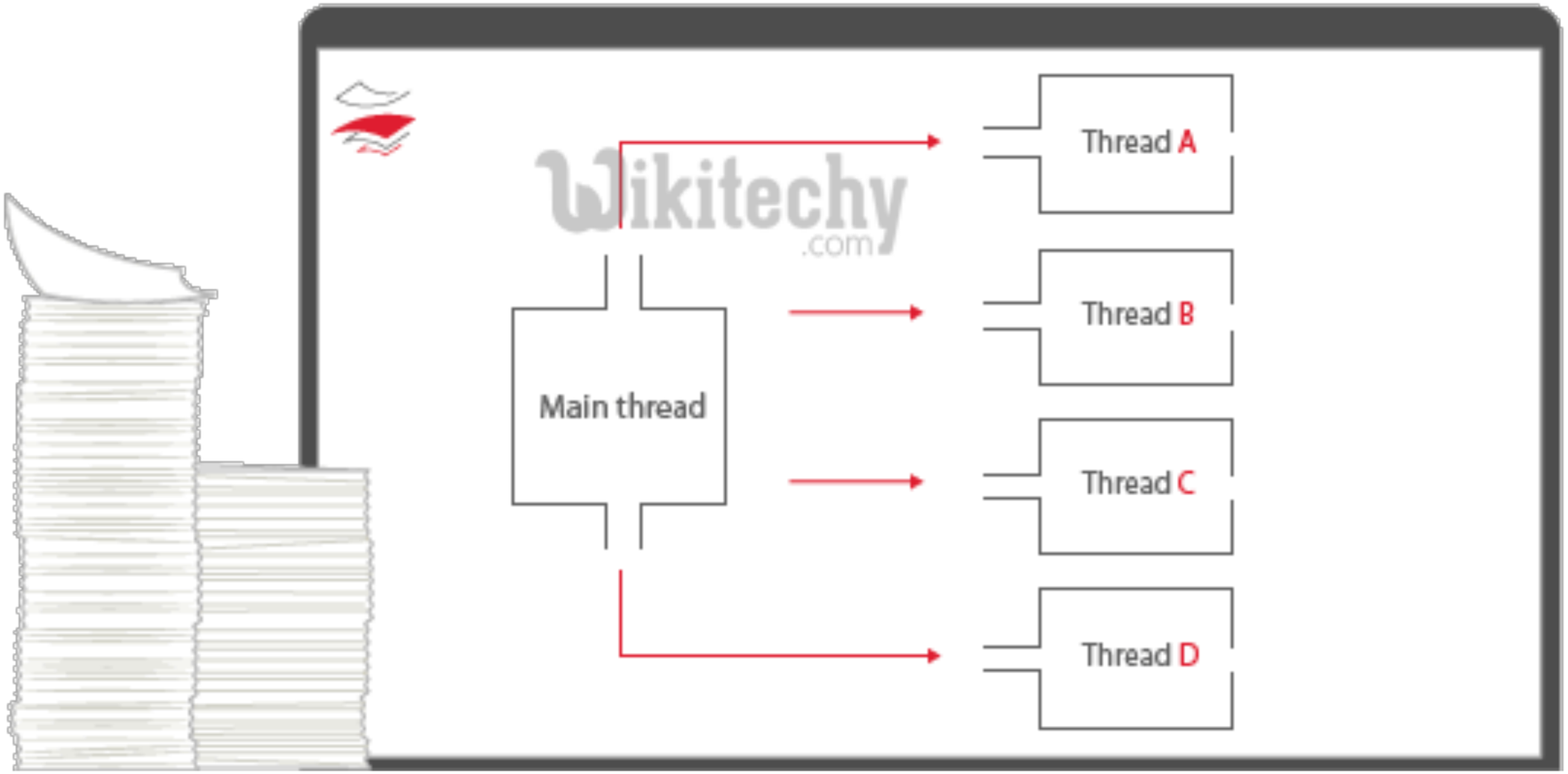


LES THREADS

Programmation système sous Unix

Multithreading



Processus lourds

- Nous avons déjà vu qu'il est possible par logiciel de créer une 'copie' ou un 'fils' du processus courant. Cela revient à créer un nouvel espace mémoire contenant une copie des données (le code reste unique).
- Les valeurs des données du processus fils sont celles de l'instant de la copie (avec fork()). Si le processus fils modifie ses données, les données du père sont inchangées. De même, si le père modifie ses données, les données du fils sont inchangées. Les deux espaces mémoires sont complètement étanches. au point que si il fallait échanger des données entre les deux processus, il faudrait utiliser des mécanismes assez complexes (IPC : Inter Processus Communication, SHM : SHared Memory, sockets UNIX etc).
- C'est ce qu'on appelle des processus lourds

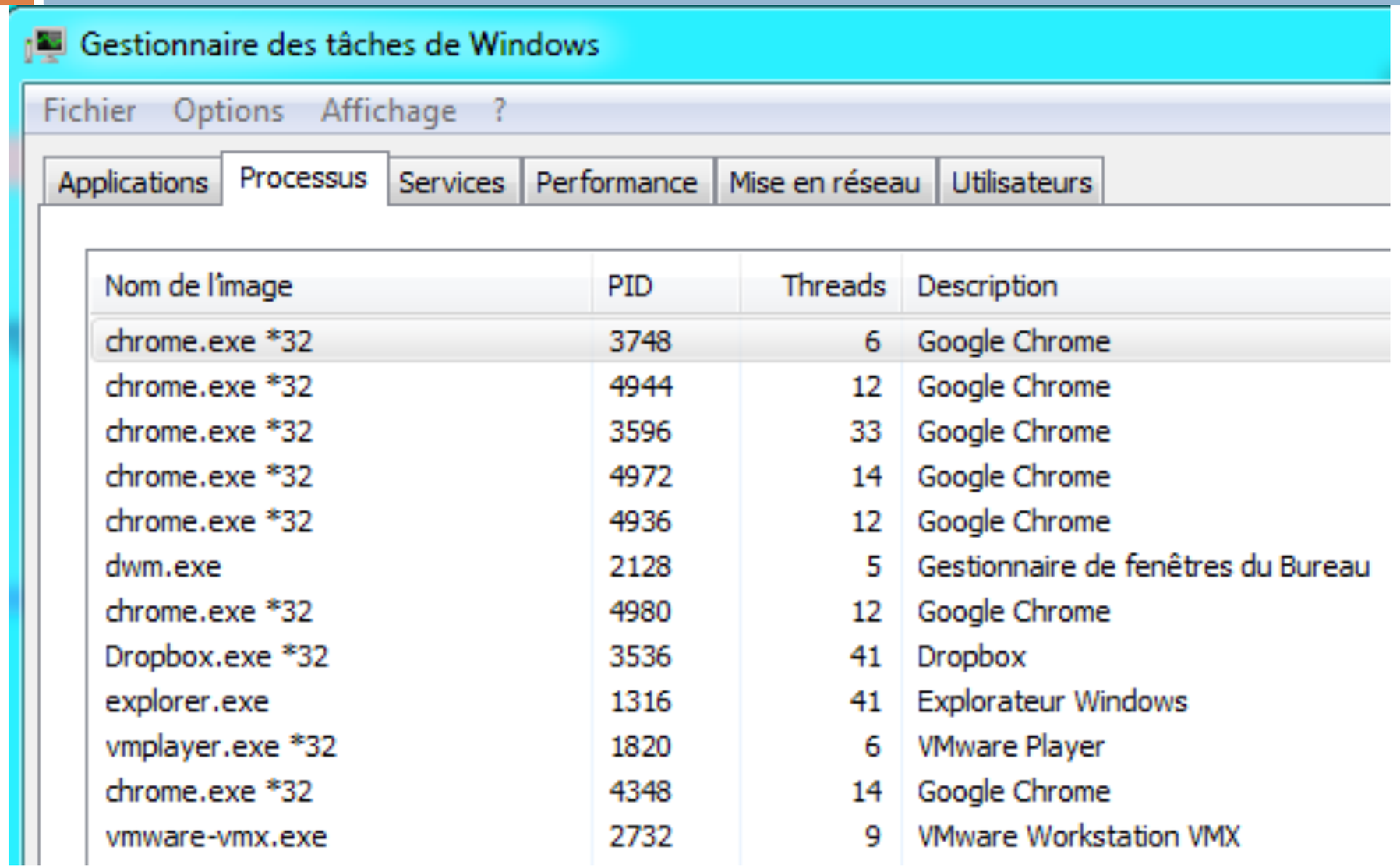
Processus légers

- Dans un processus donné, il est possible de créer un processus léger appelé aussi thread qui est un espace mémoire limité à un code exécutable (la fonction du thread). La mémoire est celle du processus courant. Si plusieurs threads sont créés, ils partagent le même espace de mémoire, mais ils ont chacun leur variables locales.
- Ce mécanisme est simple et permet d'écrire facilement des applications multitâches. Les données étant partagées, leur accès est direct.
- Revers de la médaille, les accès aux données étant directs, et des accès concurrents étant possibles, les données peuvent être incorrectes si des précautions ne sont pas prises. Il existe donc des mécanismes de protection qui permettent de réguler l'accès aux données (sémaphores, mutex).
- La gestion des threads a été normalisée par POSIX.1. C'est l'API pthreads.

Partagé entre threads	Propre à chaque thread
Code, variables globales, descripteurs de fichiers, PID	Pile (variables locales), TID

Le TID (Thread ID) est le numéro que le noyau Linux attribue à chaque thread, de la même façon qu'il attribue un PID à chaque processus. Il est unique sur tout le système à un instant donné.

Threads dans Windows



Création de 2 threads

```
#include <stdio.h>
#include <pthread.h>
static void *thread_a(void *arg) {
    (void)arg;
    puts("Bonjour, je suis le thread A");
    return NULL;
}
static void *thread_b(void *arg) {
    (void)arg;
    puts("Bonjour, je suis le thread B »);
    return NULL;
}
int main (void)
{
   pthread_t ta;
   pthread_t tb;
   puts ("main debut");
   pthread_create (&ta, NULL, thread_a, NULL);
   pthread_create (&tb, NULL, thread_b, NULL);
   puts ("main fin");
}
```

Compilation :

gcc -Wall -pthread votre_programme.c -o votre_programme

Ne fonctionne pas correctement car main se termine avant les threads

Création de 2 threads en C

```
#include <stdio.h>
#include <pthread.h>
static void *thread_a(void *arg) {
    (void)arg;
    puts("Bonjour, je suis le thread A");
    return NULL;
}
static void *thread_b(void *arg) {
    (void)arg;
    puts("Bonjour, je suis le thread B »);
    return NULL;
}
int main (void)
{
   pthread_t ta;
   pthread_t tb;
   puts ("main debut");
   pthread_create (&ta, NULL, thread_a, NULL);
   pthread_create (&tb, NULL, thread_b, NULL);
   pthread_join (ta, NULL);
   pthread_join (tb, NULL);
   puts ("main fin");
}
```

Compilation :

gcc -Wall -pthread votre_programme.c -o votre_programme

main attend que les threads se terminent

Voici un exemple minimal de découverte : un main et deux threads, qui affichent chacun leur PID et leur TID.

```c
#define _GNU_SOURCE      /* obligatoire pour utiliser gettid() */
#include <stdio.h>
#include <unistd.h>      /* getpid(), gettid(), sleep() */
#include <pthread.h>

void *mon_thread(void *arg)
{
    (void)arg;
    printf("Thread : PID = %d   TID = %d\n", getpid(), gettid());
    sleep(10);           /* le thread reste vivant 10 s pour l'observer avec ps -T */
    return NULL;
}

int main(void)
{
    pthread_t t1, t2;

    printf("main   : PID = %d   TID = %d\n", getpid(), gettid());

    pthread_create(&t1, NULL, mon_thread, NULL);
    pthread_create(&t2, NULL, mon_thread, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

Compilation :

gcc -Wall -pthread gettid.c -o gettid

main attend que les threads se terminent

Résultat obtenu :

```
main   : PID = 234   TID = 234
Thread : PID = 234   TID = 236
Thread : PID = 234   TID = 237
```

Pendant les 10 secondes d'attente, lancer ps -T -p 234 dans un second terminal :

```
  PID  SPID CMD
  234   234 decouverte_gett
  234   236 decouverte_gett
  234   237 decouverte_gett
```

Les trois choses à observer :

- Le PID est le même pour tout le monde : les trois threads appartiennent au même processus.
- Le TID est différent pour chaque thread : c'est son numéro propre, attribué par le noyau.
- Pour le main, TID = PID : le thread principal porte le numéro du processus. Les TID affichés par le programme se retrouvent dans la colonne SPID de ps -T.

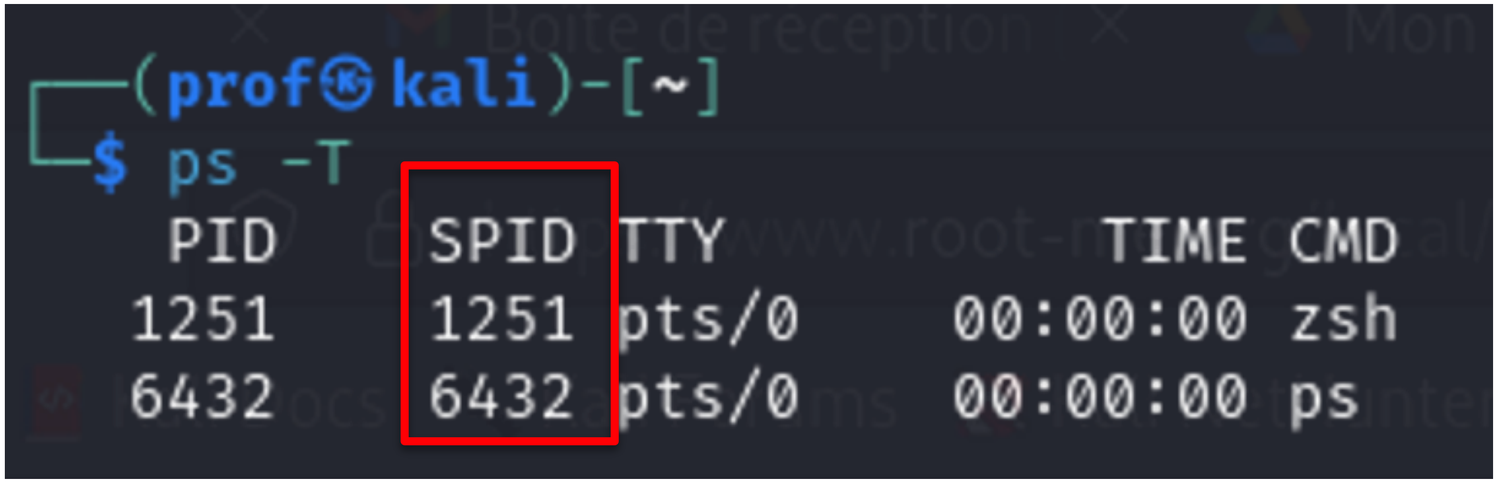
Compte-rendu

- Réaliser un compte-rendu des exercices suivants en joignant les sources et en faisant des copies d'écran des résultats obtenus dans le document.



Option à ps : -T

- Pour visualiser les threads, il y a la commande : ps -T
- N'oubliez pas de l'inclure avec : #include <stdlib.h>



```
└─$ ps -Tel
F S   UID     PID    SPID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0       1       1       0  0  80   0 -  5777 -      ?        00:00:00 systemd
1 S     0       2       2       0  0  80   0 -     0 -      ?        00:00:00 kthreadd
1 S     0       3       3       2  0  80   0 -     0 -      ?        00:00:00 pool_workqueue_release
1 S  1000     814     828     797  0  39   - -  25863 do_epo ?        00:00:00 data-loop.0
0 S  1000     816     816     797  0  80   0 - 21341 do_epo ?        00:00:00 pipewire
1 S  1000     816     821     797  0  80   0 - 21341 do_epo ?        00:00:00 module-rt
0 S  1000     817     817     797  0  69 -11 - 121302 do_sys ?       00:00:00 wireplumber
1 S  1000     817     822     797  0  80   0 - 121302 do_epo ?       00:00:00 module-rt
1 S  1000     817     823     797  0  80   0 - 121302 do_sys ?       00:00:00 gmain
1 S  1000     817     829     797  0  80   0 - 121302 futex_ ?       00:00:00 pool-spawner
1 S  1000     817     839     797  0  80   0 - 121302 do_sys ?       00:00:00 gdbus
1 S  1000     817     840     797  0  39   - - 121302 do_epo ?       00:00:00 data-loop.0
0 S  1000     818     818     797  0  69 -11 - 25278 do_epo ?        00:00:00 pipewire-pulse
1 S  1000    1534    1534    1087  0  80   0 - 25278 futex_ ?        00:00:00 ProfilerChild
4 S  1000    1541    1541    1088  0  80   0 - 601189 do_sys ?       00:00:00 Web Content
1 S  1000    1541    1550    1088  0  80   0 - 601189 do_epo ?       00:00:00 IPC I/O Child
1 S  1000    1541    1553    1088  0  80   0 - 601189 futex_ ?       00:00:00 Socket Thread
1 S  1000    1541    1554    1088  0  80   0 - 601189 futex_ ?       00:00:00 HTML5 Parser
1 S  1000    1541    1555    1088  0  80   0 - 601189 futex_ ?       00:00:00 JS Watchdog
1 S  1000    1541    1557    1088  0  80   0 - 601189 futex_ ?       00:00:00 Timer
1 S  1000    1541    1558    1088  0  80   0 - 601189 futex_ ?       00:00:00 RemVidChild
1 S  1000    1541    1566    1088  0  80   0 - 601189 futex_ ?       00:00:00 ImageIO
1 S  1000    1541    1567    1088  0  80   0 - 601189 futex_ ?       00:00:00 TaskCon~ller #0
1 S  1000    1541    1568    1088  0  80   0 - 601189 futex_ ?       00:00:00 TaskCon~ller #1
1 S  1000    1541    1569    1088  0  80   0 - 601189 futex_ ?       00:00:00 ImageBridgeChld
1 S  1000    1541    1570    1088  0  80   0 - 601189 futex_ ?       00:00:00 ProcessHangMon
1 S  1000    1541    1571    1088  0  80   0 - 601189 futex_ ?       00:00:00 ProfilerChild
```

Commande system

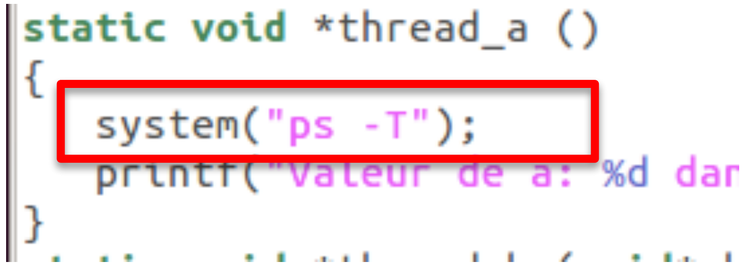
- Pour visualiser les threads dans votre programme en cours d'exécution, il y a la commande system : ps -T

```
Valeur de a dans main : 45
main debut
Valeur de a: 99999 dans thread B
  PID  SPID TTY          TIME CMD
 2114  2114 pts/1    00:00:00 sudo
 2115  2115 pts/1    00:00:02 bash
 2923  2923 pts/1    00:00:00 cours_thread2
 2923  2926 pts/1    00:00:00 cours_thread2
 2928  2928 pts/1    00:00:00 sh
 2929  2929 pts/1    00:00:00 ps
Valeur de a: 99999 dans thread A
main fin
Valeur de a dans main : 99999
```

- Si vous ne voyez pas les threads, essayez en ajoutant un sleep(1) dans chacun

Commande system ps -T

- Pour visualiser les threads, il y a la commande system : ps –T



- Ici, elle a été intégrée dans le thread A pour être exécuté quand celui existe.

Création de 2 threads en C++

```
#include <iostream>
#include <pthread.h>
using namespace std;

static void *thread_a(void *)
{
    cout << "Bonjour, je suis le thread A" << endl;
    return NULL;
}

static void *thread_b(void *)
{
    cout << "Bonjour, je suis le thread B" << endl;
    return NULL;
}

int main(void)
{
    pthread_t ta;
    pthread_t tb;

    cout << "main debut" << endl;

    pthread_create(&ta, NULL, thread_a, NULL);
    pthread_create(&tb, NULL, thread_b, NULL);

    pthread_join(ta, NULL);
    pthread_join(tb, NULL);

    cout << "main fin" << endl;
    return 0;
}
```

Compilation :
g++ -Wall -Wextra -pthread thread.cpp -o thread

main attend que les threads se terminent

Création de 2 threads en C++

Compilation : pour compiler il ajouter l'option -pthread

g++ -Wall -Wextra -pthread votre_programme.cpp -o votre_programme

Variables communes

Mise en évidence

```c
#include <stdio.h>
#include <stdlib.h>      /* pour system() */
#include <pthread.h>

int a = 45;              /* variable globale, partagée par les th

static void *thread_a(void *arg)
{
    (void)arg;
    printf("Valeur de a : %d dans thread A\n", a);
    return NULL;
}

static void *thread_b(void *arg)
{
    (void)arg;
    a = 0;
    printf("Valeur de a : %d dans thread B\n", a);
    return NULL;
}

int main(void)
{
    pthread_t ta;
    pthread_t tb;

    system("clear");
    printf("Valeur de a dans main : %d\n", a);
    puts("main debut");

    pthread_create(&ta, NULL, thread_a, NULL);
    pthread_create(&tb, NULL, thread_b, NULL);

    pthread_join(ta, NULL);
    pthread_join(tb, NULL);

    puts("main fin");
    printf("Valeur de a dans main : %d\n", a);
    return 0;
}
```

Variable globale

Variable modifiée

Résultat à l'écran

Valeur de a dans main : 45
main debut
Valeur de a: 0 dans thread B
Valeur de a: 0 dans thread A
main fin
Valeur de a dans main : 0

on voit bien ici que c'est la même variable dans main et les threads

Note : pour compiler il ajouter l'option -pthread

g++ -Wall -Wextra -pthread votre_programme.cpp -o votre_programme

Passage d'arguments

Mise en évidence

```c
#include <stdio.h>
#include <stdlib.h>      /* system() */
#include <stdint.h>      /* intptr_t */
#include <pthread.h>

int a = 45;              /* variable globale partagée */

static void *thread_a(void *arg)
{
    (void)arg;
    printf("Valeur de a : %d dans thread A\n", a);
    return NULL;
}

static void *thread_b(void *b)
{
    a = (int)(intptr_t)b;   /* récupère l'argument passé au thread */
    printf("Valeur de a : %d dans thread B\n", a);
    return NULL;
}

int main(void)
{
    pthread_t ta;
    pthread_t tb;

    system("clear");
    printf("Valeur de a dans main : %d\n", a);
    puts("main debut");

    pthread_create(&ta, NULL, thread_a, NULL);
    pthread_create(&tb, NULL, thread_b, (void *)(intptr_t)99999);

    pthread_join(ta, NULL);
    pthread_join(tb, NULL);

    puts("main fin");
    printf("Valeur de a dans main : %d\n", a);
    return 0;
}
```

Variable globale

Variable modifiée

Résultat à l'écran

Valeur de a dans main : 45

main debut

Valeur de a: 99999 dans thread B

Valeur de a: 99999 dans thread A

main fin

Valeur de a dans main : 99999

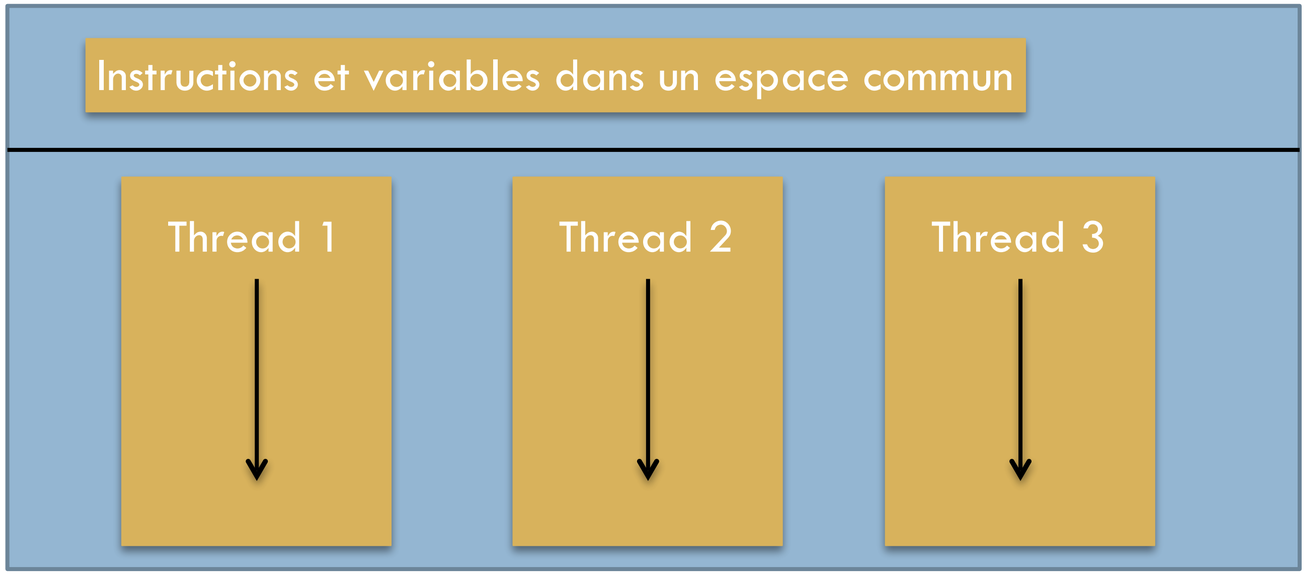
on voit que la variable globale a été modifié par l'argument

Espace mémoire

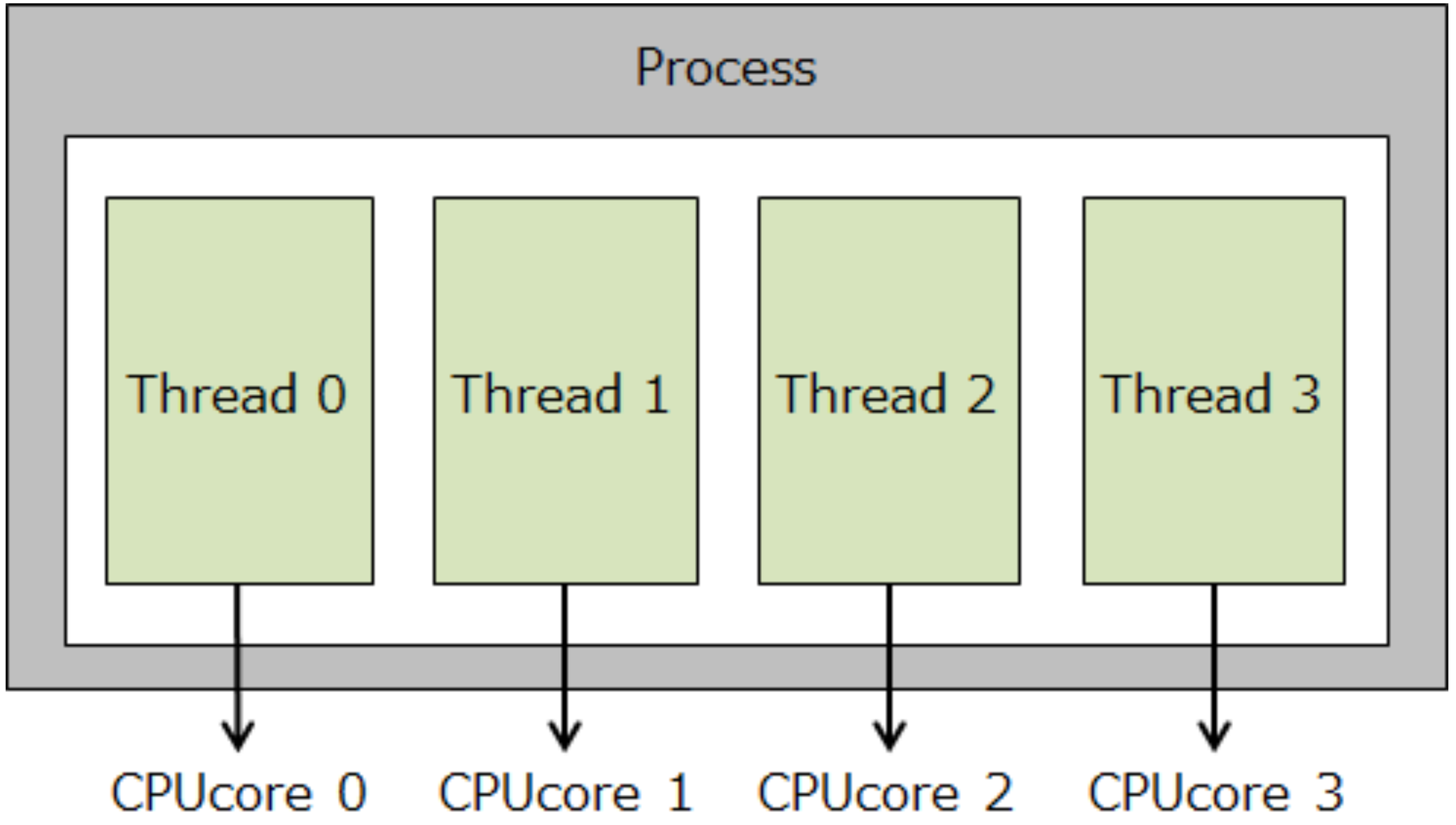
- Dans le cas du multi-processus :



- Dans le cas du multi-threads :



Threads avec multi-core



Affichage du nombre de coeurs dédiés à la machine virtuelle

```
#include <iostream>
#include <thread>

int main()
{
    unsigned int m = std::thread::hardware_concurrency();

    if (m == 0)
        std::cout << "Nombre de coeurs : \t Information non disponible.\n";
    else
        std::cout << "Nombre de coeurs : \t" << m << ".\n";

    return 0;
}
```

le nombre renvoyé est le nombre de fils d'exécution matériels que le processeur peut gérer en parallèle — ce qui inclut l'hyperthreading. Sur un processeur 4 cœurs avec hyperthreading, la fonction renvoie souvent 8, pas 4. Le terme « cœurs » dans le message est donc un léger abus de langage ; « unités d'exécution » ou « threads matériels » serait plus exact. C'est d'ailleurs cette valeur qui sert à dimensionner un pool de threads pour ne pas en créer plus que ce que la machine peut vraiment exécuter simultanément.

Compte-rendu

- Réaliser un compte-rendu des exercices précédents en joignant les sources et en faisant des copies d'écran des résultats obtenus dans le document.



Les mutex

Exclusions mutuelles

Problématique

- Avec les threads, toutes les variables sont partagées : c'est la **mémoire partagée**.
  Mais cela pose des problèmes. En effet, quand deux threads cherchent à modifier deux variables en même temps, que se passe-t-il ? Et si un thread lit une variable quand un autre thread la modifie ?
  C'est assez problématique. Par conséquent, nous allons voir une mécanisme de synchronisation : les **mutex**, un des outils permettant l'**exclusion mutuelle**.

Les mutex

- Concrètement, un **mutex** est en C une variable de type pthread_mutex_t.
- Elle va nous servir de verrou, pour nous permettre de protéger des données.
- Ce verrou peut donc prendre deux états: **disponible** et **verrouillé**.
- Quand un thread a accès à une variable protégée par un mutex, on dit qu'*il tient le mutex*.

Exemple avec mutex

- std::mutex est le plus simple d'entre eux. Il peut être verrouillé une seule et unique fois, chaque appel à lock supplémentaire attendra que le mutex soit déverrouillé via un appel à unlock avant de retourner.
- Il existe également la méthode try_lock qui permet, si le mutex est déjà verrouillé, et un appel à lock serait alors bloquant de retourner false directement. Si le mutex peut être acquis, il est verrouillé et try_lock retourne true.

La classe `mutex` est une primitive de synchronisation qui peut être utilisée pour protéger des données partagées simultanément par plusieurs threads.

Le `mutex` propose une sémantique de propriété :

- Un thread appelant possède un `mutex` quand il réussit l'appel à `lock` ou `try_lock` et cela jusqu'à ce qu'il appelle `unlock`.
- Quand un thread possède un `mutex`, tous les autres threads bloqueront (pour les appels à `lock`) ou recevront une valeur de retour `false` (pour `try_lock`) s'ils tentent de revendiquer la propriété du `mutex`.
- Un thread appelant ne doit pas posséder un `mutex` avant d'appeler `lock` ou `try_lock`.

Le comportement d'un programme n'est pas défini si un `mutex` est détruit alors qu'il est toujours détenu par un autre thread.

```
#include <iostream>       // std::cout
#include <thread>         // std::thread
#include <mutex>          // std::mutex
#include <unistd.h>

std::mutex mtx;           // mutex for critical section

void print_block (int n, char c) {
  // critical section (exclusive access to std::cout signaled by locking mtx):
  mtx.lock();
  for (int i=0; i<n; ++i) { std::cout << c; usleep(10); }
  std::cout << '\n';
  mtx.unlock();
}

int main ()
{
  std::thread th1 (print_block,50,'*');
  std::thread th2 (print_block,50,'$');

  th1.join();
  th2.join();

  return 0;
}
```

Essayez cet exemple
D'abord sans le mutex et ensuite avec et observez le comportment des threads

Quel est l'effet des mutex ?

Exercice 1 : Création et attente de threads

Écrivez un programme qui :

- crée NB_THREADS threads (constante définie par #define) ;
- fait afficher à chaque thread le message « Hello BTS !!! » ;
- fait attendre au thread principal la fin de tous les threads avec pthread_join().

Compilation : gcc -Wall -Wextra -pthread ex1.c -o ex1

Exercice 2 : Identification des threads

Modifiez le programme précédent pour que chaque thread affiche :

- son PID, avec getpid() ;
- son TID, avec gettid() (ajoutez #define _GNU_SOURCE en première ligne) ;
- son identifiant pthread, avec printf("%p\n", (void *)pthread_self());.

Pendant que les threads tournent, le thread principal exécute ps -T -p <PID> avec system().

Comparez les valeurs : lesquelles retrouvez-vous dans la colonne SPID ? Laquelle n'y apparaît pas, et pourquoi ?

FONCTION pthread_self()

PTHREAD_SELF() Linux Programmer's Manual PTHREAD_SELF()

NAME

pthread_self - obtain ID of the calling thread

SYNOPSIS

#include <pthread.h>

pthread_t pthread_self(void);

Compile and link with *-pthread*.

DESCRIPTION

The **pthread_self**() function returns the ID of the calling thread.

This is the same value that is returned in **thread* in the pthread_create() call that created this thread.

Fonction malloc

- **Entête à inclure**

  #include <stdlib.h> // <cstdlib> en C++

- **void * malloc(size_t memorySize);**

  Cette fonction permet d'allouer un bloc de mémoire dans le tas (le heap en anglais). Attention : la mémoire allouée dynamiquement n'est pas automatiquement relachée. Il faudra donc, après utilisation, libérer ce bloc de mémoire via un appel à la fonction free.

- **Paramètres**

  memorySize : permet de spécifier la taille (en octets) du bloc de mémoire à allouer.

- **Valeur de retour**

  Si tout ce passe bien, la fonction vous renvoie un pointeur sur la zone nouvellement allouée. Attention, ce pointeur est de type pointeur générique (void *) : si nécessaire, il est donc de votre responsabilité de "caster" votre pointeur vers un autre type

Travaux pratiques suite

★ **Exercice 3 : Passage de paramètres et exclusion mutuelle**

▷ **Question 1.** Modifiez le programme de l'exercice précédent pour passer son numéro d'ordre à chaque thread. Chaque thread doit ensuite l'afficher. Vérifiez que le numéro d'ordre affiché par chaque thread est bien différent (corrigez votre programme le cas échéant).

▷ **Question 2.** Déclarez une variable globale `somme` initialisée à 0. Chaque thread doit, dans une boucle, ajouter 1 000 000 fois son numéro d'ordre à cette variable globale (on veut bien faire 1 000 000 additions par thread, pas juste une). Affichez la valeur obtenue après la terminaison de tous les threads.

▷ **Question 3.** Avec 5 threads (numérotés de 0 à 4), on devrait obtenir (0 + 1 + 2 + 3 + 4) × 1 000 000 = 10 000 000. Corrigez votre programme s'il n'affiche pas systématiquement ce résultat.

▷ **Question 4.** Modifiez votre programme pour ne plus utiliser de variables globales. Il faut donc passer les adresses des différentes variables nécessaires en argument aux threads, dans une structure.

Compte-rendu

- Réaliser un compte-rendu des exercices précédents en joignant les sources et en faisant des copies d'écran des résultats obtenus dans le document.

