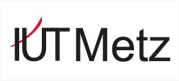
R3.05 - Programmation Système

Threads

Cyril Grelier

Université de Lorraine - IUT de Metz



Threads : Processus léger

- Un **thread** = fil d'exécution léger dans un processus.
- Threads partagent : code, données, heap.
- Chaque thread a sa **pile** (stack), son **TID** (*Thread ID*) et son **PC** (*Program Counter*).
- **Processus** = unité lourde, isolée, mémoire séparée.
- **Threads** = unités légères, mémoire partagée.

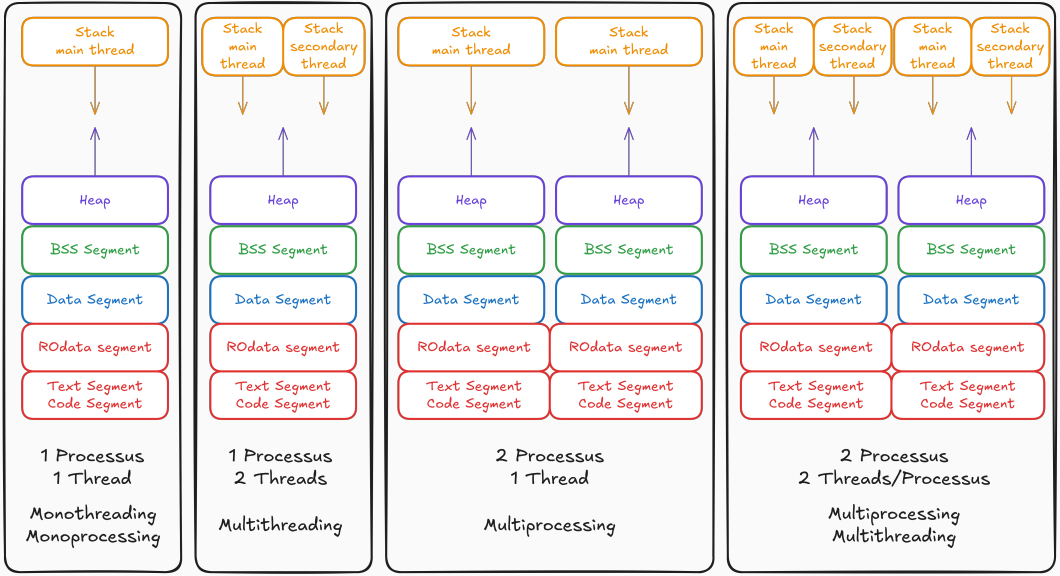
Avantages :

- Partage simple des données
- Création/destruction rapides
- Communication directe (sans IPC lourde, *Inter Process Communication*)

Inconvénients :

- Risque de *data race* $\Rightarrow$ synchronisation nécessaire
- Un crash $\Rightarrow$ tout le processus tombe

Multithreading vs Multiprocessing



Threads vs fork : cas d'usage

Threads (mémoire partagée)

- **Serveurs web** : un thread par requête HTTP, cache partagé.
- **Applications interactives** : UI fluide + calculs en parallèle.
- **Calcul parallèle** : partage de données lourdes en RAM.
- **Producteur/consommateur** : communication rapide par variables partagées.
- **Simulations** : entités légères dans le même univers mémoire.

Processus (fork) (mémoire isolée)

- **Isolation forte** : un crash n'impacte pas les autres.
- **Sécurité** : processus non fiables (ex. plugins, sandbox).
- **Services indépendants** : exécution d'outils externes.
- **Multi-utilisateurs** : chaque utilisateur a son espace séparé.

APIs disponibles

- **Pthreads** (<pthread.h>) : standard POSIX, complet.
- **C11 threads** (<threads.h>) : plus simple, portable, mais limité.

Modèle commun

- `create` : démarre un thread sur une fonction.
- `join` : attendre la fin d'un thread.
- Passage d'un seul argument (`void *`) $\Rightarrow$ souvent une structure.
- Retour : `void *` (pthreads) ou `int` (C11).

Pthreads : création et attente

Compilation : ajouter le flag -pthread

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *ma_fonction(void *arg) {
    int *val = arg; // <- cast de void * dans le type voulu
    printf("Thread: arg = %d\n", *val);
    return NULL;
}

int main(void) {
    pthread_t tid;
    int v = 42;
    if (pthread_create(&tid, NULL, ma_fonction, &v) != 0){ // <- création du thread qui va lancer ma_fonction
      perror("pthread_create");
      exit(1);
    }
    pthread_join(tid, NULL); // <- attente du thread dans le programme principal
    printf("Thread terminé !\n");
}
```

Output :

```
Thread: arg = 42
Thread terminé !
```

Retour de valeur avec pthreads

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *ma_fonction(void *arg) {
    int *val = arg;
    int *res = malloc(sizeof *res); // <- Allocation sur la heap
    *res = *val * 2;
    pthread_exit(res); // <- Ne jamais retourner l'adresse d'une variable déclarée sur le stack !
}

int main(void) {
    pthread_t tid;
    int v = 42;
    if (pthread_create(&tid, NULL, ma_fonction, &v) != 0)
        exit(1);
    int *resultat;
    pthread_join(tid, (void **)&resultat); // <- récupération du résultat avec le join
    printf("Résultat = %d\n", *resultat);
    free(resultat); // <- ne pas oublier de free la mémoire allouée sur la heap
}
```

Problème de *data race*

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

int compteur = 0;

void *incremente(void *arg) {
    for (int i = 0; i < 1000000; i++) {
        compteur++;
    }
    return NULL;
}

int main(void) {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, incremente, NULL);
    pthread_create(&t2, NULL, incremente, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Compteur final = %d\n", compteur);
}
```

Quel résultat pour le compteur final ?

Résultat attendu :

Problème de *data race*

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

int compteur = 0;

void *incremente(void *arg) {
    for (int i = 0; i < 1000000; i++) {
        compteur++;
    }
    return NULL;
}

int main(void) {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, incremente, NULL);
    pthread_create(&t2, NULL, incremente, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Compteur final = %d\n", compteur);
}
```

Quel résultat pour le compteur final ?

Résultat attendu : 2 000 000

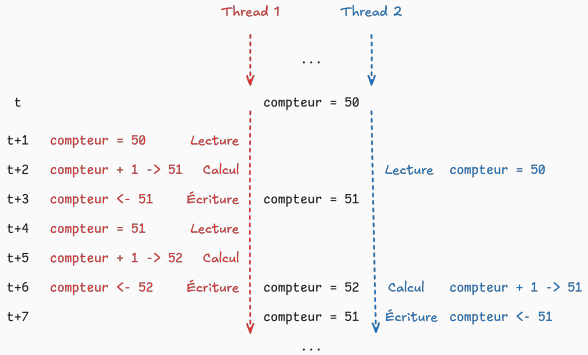
Problème de *data race*

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

int compteur = 0;

void *incremente(void *arg) {
    for (int i = 0; i < 1000000; i++) {
        compteur++;
    }
    return NULL;
}

int main(void) {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, incremente, NULL);
    pthread_create(&t2, NULL, incremente, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Compteur final = %d\n", compteur);
}
```



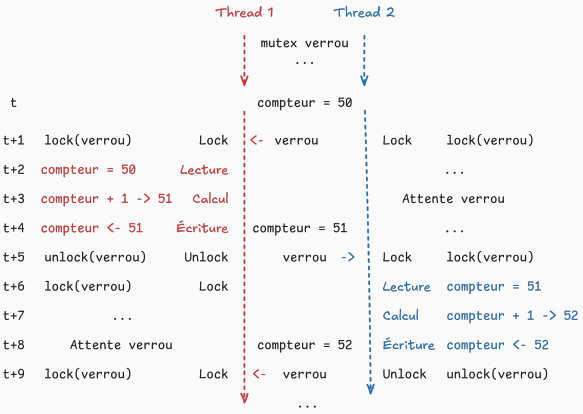
Résultat attendu : 2 000 000

Résultat obtenu : varie ; 973 323, 1 014 943, 1 124 725, . . . ⇒ *data race*

Mutex pour protéger les accès

```c
pthread_mutex_t verrou = PTHREAD_MUTEX_INITIALIZER;
int compteur = 0;

void *incremente(void *arg){
    for (int i=0;i<1000000;i++){
        pthread_mutex_lock(&verrou);
        compteur++;
        pthread_mutex_unlock(&verrou);
    }
    return NULL;
}
```



Grâce au **mutex**, compteur final = 2 000 000.

Threads en C11

API portable (<threads.h>), plus simple (peut tout de même nécessiter le flag -pthread à la compilation sous Linux et minimum -std=c11).

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

int ma_fonction(void *arg){
    int v = *(int*)arg;
    printf("Thread C11, arg=%d\n", v);
    return 0;
}

int main(void) {
    thrd_t t; int val = 42;
    if (thrd_create(&t,ma_fonction,&val)!=thrd_success){ // <- création
        return 1;
    }
    thrd_join(t,NULL); // <- attente
}
```

Synchronisation en C11

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

mtx_t verrou;
int compteur=0;

int incremente(void *arg) {
    for (int i=0;i<1000000;i++){
        mtx_lock(&verrou);
        compteur++;
        mtx_unlock(&verrou);
    }
    return 0;
}

int main(void) {
    thrd_t t1,t2;
    mtx_init(&verrou, mtx_plain);
    thrd_create(&t1,incremente,NULL);
    thrd_create(&t2,incremente,NULL);
    thrd_join(t1,NULL); thrd_join(t2,NULL);
    printf("Compteur=%d\n",compteur);
    mtx_destroy(&verrou);
}
```

Incrément atomique sans mutex

```c
#include <stdatomic.h>

atomic_int compteur = 0;

void *incremente(void *arg){
    for(int i=0;i<1000000;i++){
        atomic_fetch_add(&compteur, 1);
    }
    return NULL;
}
```

Utile pour de petits compteurs ; pour des sections critiques plus grandes $\Rightarrow$ mutex.

Erreurs fréquentes

- **Data race** $\Rightarrow$ mutex / atomiques
- **Deadlock** (ordre d'acquisition incohérent), **livelock**, **starvation**
- **False sharing** (caches) : séparer les données très modifiées par thread
- **printf** : sorties mélangées entre threads
- **join oublié** : ressources du thread non libérées (le thread est terminé donc n'utilise plus le CPU mais reste en mémoire dans le noyau) - sinon détacher le thread (`thrd_detach` / `pthread_detach`) pour qu'il libère les ressources automatiquement mais plus d'accès à sa valeur de retour