Parallélisme et concurrence

Programmation avancée

Jacques-Henri Jourdan

17 mai 2024

Examen

L'examen est fixé le

vendredi 31 mai, de 11h15 à 13h15.

La concurrence et le parallélisme

Quelle est, pour vous, la définition de ces deux termes ?
Quelle différence faites-vous ?

La concurrence et le parallélisme

Quelle est, pour vous, la définition de ces deux termes ?
Quelle différence faites-vous ?

Le parallélisme, c'est quand on effectue plusieurs calculs en même temps, généralement pour gagner de la performance.

Exemples ?

La concurrence et le parallélisme

Quelle est, pour vous, la définition de ces deux termes ?
Quelle différence faites-vous ?

Le parallélisme,
pour gagner de

Exemples ?

- plusieurs cœurs,
- plusieurs unités de calcul dans un même cœur (exécution *out-of-order*),
- *instructions SIMD* permettant d'effectuer la même opération simultanément sur plusieurs opérandes (*vectorisation* du calcul),
- calcul massivement parallèle dans les GPUs,
- répartition du calcul sur plusieurs ordinateurs dans un *cluster*,
- ...

La concurrence et le parallélisme

Quelle est, pour vous, la définition de ces deux termes ?
Quelle différence faites-vous ?

Le parallélisme, c'est quand on effectue plusieurs calculs en même temps, généralement pour gagner de la performance.

Exemples ?

La concurrence c'est lorsque plusieurs tâches sont à gérer simultanément.
Il n'y a pas *a priori* de but de performance.
La difficulté, c'est que ces tâches partagent des ressources communes, et qu'il faut donc établir des protocoles pour permettre ce partage.

Exemples ?

La concurrence et le parallélisme

Quelle est, pour vous, la définition de ces deux termes ?
Quelle différence faites-vous ?

Le parallélisme,
pour gagner de

Exemples ?

La concurrence
Il n'y a pas *a pr*
La difficulté, c'e
établir des protocoles pour permettre ce partage.

Exemples ?

- exécution simultanée de plusieurs programmes sur une même machine (même avec un seul cœur !),
- dans un même programme, gestion simultanée de l'interface utilisateur et des calculs,
- dans un serveur, gestion simultanée de plusieurs connexions,
- implémentation de parallélisme avec plusieurs tâches.

La concurrence et le parallélisme

Quelle est, pour vous, la définition de ces deux termes ?
Quelle différence faites-vous ?

Le parallélisme, c'est quand on effectue plusieurs calculs en même temps, généralement pour gagner de la performance.

Exemples ?

La concurrence c'est lorsque plusieurs tâches sont à gérer simultanément.
Il n'y a pas *a priori* de but de performance.
La difficulté, c'est que ces tâches partagent des ressources communes, et qu'il faut donc établir des protocoles pour permettre ce partage.

Exemples ?
Exemples de ressources à partager et de solutions pour les partager ?

La concurrence et le parallélisme

Quelle est, pour vous, la définition de ces deux termes ?
Quelle différenc

Le parallélisme,
pour gagner de

Exemples ?

La concurrence
Il n'y a pas *a pr*
La difficulté, c'e
établir des proto

Exemples ?

Exemples de ressources à partager et de solutions pour les partager ?

- temps CPU,
  - solution : utilisation d'un ordonnanceur (*scheduler*), ...
- mémoire partagée (RAM, système de fichiers, base de donnée, structure de donnée, ...)
  - solution : verrous, communication par canaux, structures de donnés linéarisables
- connexion réseau partagé par plusieurs machines ou processus
  - solution : segmentation des communications en paquets pour multiplexer les connexions
- ...

La concurrence et le parallélisme

L'un sans l'autre

Bien que parfois liées, les deux notions ne vont pas nécessairement ensemble.

Exemples de concurrence sans parallélisme ?

La concurrence et le parallélisme

L'un sans l'autre

Bien que parfois liées, les deux notions ne vont pas nécessairement ensemble.

Exemples de concurrence sans parallélisme ?

- plusieurs logiciels sur une même machine avec un seul cœur,
- un même programme interagit avec plusieurs intervenants externes (ex : connexions réseau, fichiers disque, utilisateur...)
  - une tâche pour chaque intervenant

La concurrence et le parallélisme

L'un sans l'autre

Bien que parfois liées, les deux notions ne vont pas nécessairement ensemble.

Exemples de concurrence sans parallélisme ?

- plusieurs logiciels sur une même machine avec un seul cœur,
- un même programme interagit avec plusieurs intervenants externes (ex : connexions réseau, fichiers disque, utilisateur...)
  - une tâche pour chaque intervenant

Exemples de parallélisme sans concurrence ?

La concurrence et le parallélisme

L'un sans l'autre

Bien que parfois liées, les deux notions ne vont pas nécessairement ensemble.

Exemples de concurrence sans parallélisme ?

- plusieurs logiciels sur une même machine avec un seul cœur,
- un même programme interagit avec plusieurs intervenants externes (ex : connexions réseau, fichiers disque, utilisateur...)
  - une tâche pour chaque intervenant

Exemples de parallélisme sans concurrence ?

Deux machines indépendantes qui calculent.
Instructions SIMD.

Le problème

On dispose d'un seul processeur, sur lequel on veut exécuter simultanément plusieurs tâches.

Comment fait-on ?

Première solution : les processus

Sur les systèmes d'exploitations modernes, on peut exécuter plusieurs programmes simultanément.

Moins de cœurs que de processus : le noyau interrompt (« préempte ») régulièrement chaque programme, pour donner la main au suivant.
Illusion que les programmes s'exécutent simultanément.
Préemption : $\sim$ 250Hz pour beaucoup de distributions Linux.

Première solution : les processus

Sur les systèmes d'exploitations modernes, on peut exécuter plusieurs programmes simultanément.

Moins de cœurs que de processus : le noyau interrompt (« préempte ») régulièrement chaque programme, pour donner la main au suivant.
Illusion que les programmes s'exécutent simultanément.
Préemption : $\sim$ 250Hz pour beaucoup de distributions Linux.

Chaque processus a un espace mémoire différent (*c.f.*, cours sur la gestion mémoire) pour garantir son indépendance

Première solution : les processus

Sur les systèmes d'exploitations modernes, on peut exécuter plusieurs programmes simultanément.

Moins de cœurs que de processus : le noyau interrompt (« préempte ») régulièrement chaque programme, pour donner la main au suivant.
Illusion que les programmes s'exécutent simultanément.
Préemption : $\sim$ 250Hz pour beaucoup de distributions Linux.

Chaque processus a un espace mémoire différent (*c.f.*, cours sur la gestion mémoire) pour garantir son indépendance

Problèmes des processus :

- La communication entre processus est lente et compliquée.
- Créer un processus est très coûteux (il faut allouer un nouvel espace mémoire virtuel...).

On les réserve donc aux cas où on a peu de tâches, relativement indépendantes.

Deuxième solution : les threads

Dans certains langages de programmation : threads (fils d'exécution en français).

Idée : plusieurs fonctions sont en cours d'exécution en même temps.

Point crucial : même espace mémoire.
Seule la pile est spécifique à chaque thread.

Deuxième solution : les threads

Dans certains langages de programmation : threads (fils d'exécution en français).

Idée : plusieurs fonctions sont en cours d'exécution en même temps.

Point crucial : même espace mémoire.
Seule la pile est spécifique à chaque thread.

API en OCaml :

```
module Thread = struct
  type t

  (* ‘create f x‘ exécute ‘f x‘ dans un nouveau thread. On peut continuer
     l’exécution du code appelant, même si ‘f x‘ n’est pas fini.
     Renvoie l’identifiant du thread créé.  *)
  val create : ('a -> 'b) -> 'a -> t

  (* ‘join t‘ attend la fin du thread identifié par ‘t‘. *)
  val join : t -> unit
end
```

Deuxième solution : les threads

Dans certains langages de programmation : threads (fils d'exécution en français)

Idée : plusieurs

Point crucial : m
Seule la pile est

API en OCaml

```
module Thread = 
  type t

  (* 'create f x
     l'exécution
     Renvoie l'i
  val create : (

  (* 'join t' at
  val join : t -
end
```

Attention, les threads ne sont pas forcément exécutés en parallèle !

- Certains langages ne le supportent pas (Python, OCaml $< 5.x$, ...).
- La machine n'a pas forcément suffisamment de cœurs.

Par exemple, en OCaml 4.x, il y a un mécanisme de préemption pour donner l'illusion que les threads s'exécutent simultanément.

Pour beaucoup d'applications : besoin de concurrence, pas de parallélisme.

Limite des threads « natifs »

- La création d'un thread « natif » est coûteuse.
  Il faut que le système d'exploitation et le runtime du langage alloue des structures de données internes, une nouvelle pile...

  On ne peut pas l'utiliser quand on a de nombreuses petites tâches.
  Par exemple, on ne veut pas créer un nouveau thread pour chaque connexion à un serveur Web.

- On a souvent besoin de s'assurer que certaines parties du code ne sont pas interrompues (ex. quand on accède à une structure de données globale).

  Or, soit les threads sont exécutés en parallèle, soit ils sont préemptés de manière imprévisible.

Threads collaboratifs

Une possibilité est de désactiver la préemption et l'exécution en parallèle.
Le programmeur peut alors manuellement déclencher le changement de tâche, grâce à une instruction `yield`.

Un avantage, c'est qu'on peut fournir le multithreading sous la forme d'une bibliothèque, sans support du système.
On ne paie donc pas le coût des threads natifs.

C'est ce que font par exemple :

- La bibliothèque `Lwt` (« LightWeight Threads ») en OCaml.
- La crate `futures` en Rust.

La monade de Lwt

Lwt repose sur l'utilisation d'une monade.
Voici l'idée de l'implémentation :

```
module Lwt = struct
  type 'a t = 'a state ref                (* An Lwt promise can be in two states: *)
  and 'a state =
  | Fulfilled of 'a                       (*   - it is "fulfilled" (i.e., computation finished) *)
  | Suspended of 'a cont_list             (*   - it is suspended, and *)
  and 'a cont_list = ('a -> unit) list    (*     has a list of things to do when fulfilled. *)

  (* INTERNAL: creates a new suspended promise *)
  (* unit -> 'a t *)
  let new_suspended () =
    ref (Suspended [])

  ...
```

La monade de Lwt

Queue d'ordonnancement

```
...

(* INTERNAL: Scheduling queue *)
(* (unit -> unit) Queue.t *)
let todo = Queue.create ()

(* INTERNAL: Schedule something to do *)
(* ('a -> unit) -> 'a -> unit *)
let schedule f x  =
  Queue.push (fun () -> f x) todo

(* Run the queue until a promise is fulfilled *)
(* Promises are setup in such a way `run` always finishes before the queue is empty. *)
(* 'a t -> 'a  *)
let rec run p =
  match !p with
  | Fulfilled a -> a
  | Suspended _ ->
    (Queue.pop todo) (); run p

...
```

La monade de Lwt

Opérations internes sur les promesses

```
...

(* INTERNAL: Add a continuation to a promise. *)
(* 'a t -> ('a -> unit) -> unit *)
let add_cont x k : unit =
  match !x with
  | Fulfilled a -> schedule k a
  | Suspended kl -> x := Suspended (k :: kl)

(* INTERNAL: Fulfill a promise and schedule its continuations. *)
(* 'a t -> 'a -> unit *)
let fulfill p a =
  match !p with
  | Fulfilled _ -> assert false
  | Suspended kl ->
    p := Fulfilled a;
    List.iter (fun k -> schedule k a) kl

...
```

La monade de Lwt

bind et return

```
...

(* Monadic return *)
(* 'a -> 'a t *)
let return x = ref (Fulfilled x)

(* Monadic bind *)
(* 'a t -> ('a -> 'b t) -> 'b t *)
let bind x f =
  let r = new_suspended () in
  add_cont x (fun y -> add_cont (f y) (fun z -> fulfill r z));
  r

...
```

La monade de Lwt

Concurrence

```
  (* Yield to other scheduled promises. *)
  (* unit -> unit t *)
  let pause () =
    let r = new_suspended () in
    schedule (fun () -> fulfill r ()) ();
    r

  (* Wait for both promises to be fulfilled. *)
  (* 'a t -> 'b t -> ('a * 'b) t *)
  let both pa pb =
    bind pa (fun a -> bind pb (fun b -> return (a, b)))

  ...
end
```

1 Threads et processus
- Threads et processus système
- Promesses/`Lwt`

2 Mémoire partagée
- Data race
- Modèles mémoire faibles
- Synchronisation
- Structures de données concurrentes
- Concurrence en Rust

3 Parallélisme

Exemple : race sur un `vector` en C++

```
void add_10_elem(vector<int> *vec) {
  for(int i = 0; i < 10; i++) vec->push_back(i);
}

void f() {
  vector<int> v;
  thread t(add_10_elem, &v);
  add_10_elem(&v);
  t.join();
  if(v.size() != 20) {
    cout << "ERROR" << endl;
  }
}
```

Que se passe-t-il ? (Démo `race_on_vec.cpp`)

Exemple : race sur un `vector` en C++

```
void add_10_elem(vector<int> *vec) {
  for(int i = 0; i < 10; i++) vec->push_back(i);
}

void f() {
  vector<int> v;
  thread t(add_10_elem, &v);
  add_10_elem(&v);
  t.join();
  if(v.size() != 20) {
    cout << "ERROR" << endl;
  }
}
```

Que se passe-t-il ? (Démo `race_on_vec.cpp`) Nous avons créé une data race !

I.e., deux threads tentent de modifier simultanément le vecteur.
L'opération `push_back` passe `transitoirement` par un état invalide. Un autre accès observe cet état invalide, et ne peut donc pas s'exécuter normalement.

Exemple : incrémentation d'une variable partagée

```
void incr(int *x) {
  x += 1
}

void f() {
  int x = 0;

  thread t(incr, &x);
  incr(&x);
  t.join();

  if(x != 2) cout << "ERREUR" << endl;
}
```

Que se passe-t-il ? (Démo `race_on_int.cpp`)

Exemple : incrémentation d'une variable partagée

```
void incr(int *x) {
  x += 1
}

void f() {
  int x = 0;

  thread t(incr, &x);
  incr(&x);
  t.join();

  if(x != 2) cout << "ERREUR" << endl;
}
```

Que se passe-t-il ? (Démo `race_on_int.cpp`) Il s'agit d'une autre data race.

Si les deux threads incrémentent la même variable en même temps, alors il se peut qu'on ne voit que l'une des deux incrémentations.
On dit que l'incrémentation n'est pas atomique.

Exemple : comportement non séquentiellement consistent

Considérons le programme suivant :

```
struct Data { int a, b, a_read, b_read; };

void read_write_1(Data *d) { // Thread 1
  d->b = 1;
  d->a_read = d->a;
}

void read_write_2(Data *d) { // Thread 2
  d->a = 1;
  d->b_read = d->b; // Si on lit 0 ici, alors le thread 1 n'a pas commencé, et devrait voir
                    // l'écriture d->a = 1 ???
}

void f() {
  struct Data d = {0, 0, 0, 0};

  thread t(read_write_1, &d);
  read_write_2(&d);
  t.join();

  if(d.a_read == 0 && d.b_read == 0) cout << "ERREUR" << endl;
}
```

Que se passe-t-il ? (Démo `race_on_int_2.cpp`)

Exemple : comportement non séquentiellement consistent

Considérons le programme suivant :

```
struct Data { int a, b, a_read, b_read; };

void read_write_1(Data *d) { // Thread 1
  d->b = 1;
  d->a_read = d->a;
}

void read_write_2(Data *d) { // Thread 2
  d->a = 1;
  d->b_read = d->b; // Si on lit 0 ici, alors le thread 1 n'a pas commencé, et devrait voir
                    // l'écriture d->a = 1 ???
}

void f() {
  struct Data d = {0, 0, 0, 0};

  thread t(read_write_1, &d);
  read_write_2(&d);
  t.join();

  if(d.a_read == 0 && d.b_read == 0) cout << "ERREUR" << endl;
}
```

Que se passe-t-il ? (Démo `race_on_int_2.cpp`) Encore une data race !
Mais celle-ci exhibe un comportement non séquentiellement consistent !

Races

Une race a lieu lorsque :

- deux accès simultanés ont lieu sur le même emplacement mémoire, dans deux threads différents,
- l'un d'eux (au moins) est une écriture.

C'est en général le signe d'un bug dans le programme.

En C/C++/Rust (unsafe), les race sur des variables ordinaires s'appellent des data races. Ce sont des comportements indéfinis.
I.e., le compilateur a le droit de tout faire.
Si on a vraiment besoin de faire des races, alors il faut utiliser des variables atomiques.

Modèles mémoire faible

```
struct Data { int a, b, a_read, b_read; };

void read_write_1(Data *d) { // Thread 1
  d->b = 1;
  d->a_read = d->a;
}

void read_write_2(Data *d) { // Thread 2
  d->a = 1;
  d->b_read = d->b; // Si on lit 0 ici, alors le thread 1 n'a pas commencé, et devrait voir
                    // l'écriture d->a = 1 ???
}
```

L'état de sortie `d.a_read == 0 && d.b_read == 0` ne devrait jamais être observé si le comportement de la mémoire était simplement l'entrelacement des accès des différents threads.

Pourtant, on l'observe :

- même si on utilise des variables atomiques,
- même si on écrit le programme en assembleur directement.

Pourquoi ?

Modèles mémoire faible

Pourquoi ?

Ces comportement dits faiblement cohérents sont le résultat de :

- certaines optimisations du compilateur,
  - Le compilateur pourrait réordonner les accès à la mémoire, sachant qu'ils sont *a priori* indépendants.
- l'architecture des processeurs modernes :
  - ils peuvent réordonner certaines instructions « indépendantes » pour plus de performances,
  - ils peuvent utiliser un cache et ne pas accéder à la mémoire directement,
  - les ordres d'écriture n'atteignent pas la mémoire immédiatement.

Bref, c'est compliqué !

Modèles mémoire faible

Pourquoi ?

Ces comportement dits faiblement cohérents sont le résultat de :

- certaines optimisations du compilateur,
  - Le compilateur pourrait réordonner les accès à la mémoire, sachant qu'ils sont *a priori* indépendants.
- l'architecture des processeurs modernes :
  - ils peuvent réordonner certaines instructions « indépendantes » pour plus de performances,
  - ils peuvent utiliser un cache et ne pas accéder à la mémoire directement,
  - les ordres d'écriture n'atteignent pas la mémoire immédiatement.

Bref, c'est compliqué !

Donc c'est intéressant !

Modèles mémoire faible sans parallélisme

Si on écrit un programme avec plusieurs threads, mais que l'on dispose que d'un cœur, observe-t-on ces comportements ?

Modèles mémoire faible sans parallélisme

Si on écrit un programme avec plusieurs threads, mais que l'on dispose que d'un cœur, observe-t-on ces comportements ?

Oui, car ces comportements résultent aussi d'optimisations du compilateur.

Par contre, si on utilise des threads collaboratifs, le compilateur n'a pas le droit d'échanger un accès à la mémoire avec l'instruction `yield`.
Donc il n'y a pas de comportement faiblement cohérent.

Modèles mémoire faible

Solutions

Le premier problème est de décrire les comportements possibles :

- au niveau d'un processeur,
- au niveau d'un langage de programmation.

Cela nécessite en général la définition d'une relation *happens-before* :

- quels sont les événements dont on sait qu'ils sont exécutés avant tel autre ?
- dans certains cas, permet de garantir le résultat d'une lecture.

Modèles mémoire faible

Solutions

Le premier problème est de décrire les comportements possibles :

- au niveau d'un processeur,
- au niveau d'un langage de programmation.

Cela nécessite en général la définition d'une relation *happens-before* :

- quels sont les événements dont on sait qu'ils sont exécutés avant tel autre ?
- dans certains cas, permet de garantir le résultat d'une lecture.

Le deuxième problème est de raisonner sur les programmes présentant des races.

- parce que ce n'est pas toujours un bug,
- et que ça permet de s'assurer de la correction d'un programme.

Théorème DRF

Heureusement, le modèles mémoire faible ont toujours un propriété proche de la suivante :

Théorème DRF

Si le programme s'exécute sans data race dans un modèle séquentiellement consistent, alors il a le même comportement dans un modèle séquentiellement consistent.

Du coup, si on ne joue pas avec le feu avec des data race, alors on peut raisonner de manière séquentiellement consistante.

Besoin de synchronisation

S'il ne faut pas faire de *race* sur des variables (atomiques ou pas), comment faire pour communiquer entre threads ?

Besoin de synchronisation

S'il ne faut pas faire de *race* sur des variables (atomiques ou pas), comment faire pour communiquer entre threads ?

Il faut utiliser des primitives de synchronisation.

Il en existe de nombreux types : verrous (mutex), barrières, sémaphores, ...

Mutex

Mutex est l'abréviation de *mutual exclusion*.
Un Mutex (ou lock, verrou) a en général deux méthodes :

- `lock` permet d'acquérir (ou *verrouiller*) le verrou,
- `unlock` fait l'opération inverse : libérer (ou *déverrouiller*).
  - Dans le même thread que l'acquisition.

Propriété essentielle : le verrou ne peut être acquis qu'une seule fois en même temps.

Si on appelle `lock` alors que le verrou est déjà verrouillé, alors on attend.

Mutex

Utilisation

Un mutex permet d'éviter les data races à l'accès à une ressource.

Par exemple, si on a une table de hachage partagée, on peut décider que son accès est protégé par un mutex.
On s'assure que tous les accès se font entre un appel à `lock` et un appel à `unlock`.

Ainsi, si deux threads veulent accéder à la table de hachage en même temps, l'un des deux attendra lors de son appel à `lock`.

Mutex<T> : exclusion mutuelle en Rust

En Rust, le type `Mutex<T>` est un verrou, protégeant explicitement une variable de type `T`.

Dans beaucoup de langages de programmation, les mutex ne sont pas associés explicitement à la ressource qu'ils protègent.

- En Rust, un `Mutex<T>` détient une valeur de type `T`.
- Un exemple de mutabilité intérieure, parce que cela permet de modifier une variable partagée une fois le verrou acquis.

Mutex<T> : API

```
pub struct Mutex<T> { ... }
pub struct MutexGuard<'a, T> { ... }

impl<T> Mutex<T> {
    pub fn new(t: T) -> Mutex<T> { ... }

    // Permet d'acquérir le verrou, renvoie un MutexGuard qui permet d'accéder au contenu du verrou
    pub fn lock<'a>(&'a self) -> MutexGuard<'a, T> { ... }

    // Ces fonctions permettent d'accéder le contenu du mutex sans verrouiller.
    // C'est sûr, car on sait ici que nous sommes le propriétaire exclusif du verrou.
    pub fn into_inner(self) -> T { ... }
    pub fn get_mut(&mut self) -> &mut T { ... }
}
```

Petit mensonge : un verrou peut être « empoionné » si un thread s'est arrêté avec un verrou acquis...

Ce détail ajoute du bruit à l'API, qu'on omet ici. Vous verrez la vraie API en TD.

Mutex<T> : API

MutexGuard<a, T>

```
// Les MutexGuards peuvent être utilisées comme un emprunt (partagé ou unique)
impl<'a, T> Deref for MutexGuard<'a, T> {
    type Target = T;
    fn deref(&self) -> &T { ... }
}
impl<'a, T> DerefMut for MutexGuard<'a, T> {
    fn deref_mut(&mut self) -> &mut T { ... }
}

// Détruire une MutexGuard déverrouille le  mutex
impl<'a, T> Drop for MutexGuard<'a, T> {
    fn drop(&mut self) { ... }
}
```

Autres primitives de synchronisation

Il y a de nombreux autres systèmes de synchronisation :

- Verrous lecteur/écrivain `RwLock` : deux méthodes de verrouillage différentes.
  - Si verrouillage en lecture : plusieurs possible simultanément.
  - Si verrouillage en écriture : exclusif, pas de lecture simultanée possible.
  - En fait, c'est comme `RefCell`, mais en autorisant la concurrence.
- Barrières : attendre que tous les threads aient atteint un certain point de programme avant de redémarrer
- sémaphores, variables de condition...

Interblocage

Considérons le programme Rust suivant :

```rust
fn thread_1(a: &Mutex<i32>, b: &Mutex<i32>) {
    let ga = a.lock();
    let gb = b.lock();
    *gb += *ga
}
```

```rust
fn thread_2(a: &Mutex<i32>, b: &Mutex<i32>) {
    let gb = b.lock();
    let ga = a.lock();
    *gb += *ga
}
```

Problème : si le thread 1 acquiert le verrou `a` et le thread 2 acquiert le verrou `b`, alors les deux threads restent bloqués pour verrouiller l'autre verrou.

Cela s'appelle un interblocage (*deadlock*). C'est un bug classique en programmation concurrente.

Interblocage

Considérons le programme Rust suivant :

```
fn thread_1(a: &Mutex<i32>, b: &Mutex<i32>) {
    let ga = a.lock();
    let gb = b.lock();
    *gb += *ga
}
```

```
fn thread_2(a: &Mutex<i32>, b: &Mutex<i32>) {
    let gb = b.lock();
    let ga = a.lock();
    *gb += *ga
}
```

Problème : si le thread 1 acquiert le verrou `a` et le thread 2 acquiert le verrou `b`, alors les deux threads restent bloqués pour verrouiller l'autre verrou.

Cela s'appelle un interblocage (*deadlock*). C'est un bug classique en programmation concurrente.

Un bon moyen de s'en prémunir est d'établir un ordre partiel sur les verrous d'un programme, et de s'assurer qu'ils sont toujours acquis dans cet ordre.

Structures de données concurrentes

Pour communiquer entre threads, on peut aussi utiliser des bibliothèques qui fournissent des structures de données qui sont correctes même en cas d'accès concurrents.

Exemples habituels :

- queue (producteur unique ou pas, consommateur unique ou pas),
- table de hachage concurrente,
- comptage de référence *atomique* (`Arc` en Rust).

Linéarisabilité

Le plus souvent, ces structures de données sont linéarisables.

C'est-à-dire qu'il existe, pour chaque opération sur la structure de donnée, un instant appelé point de linéarisation, tel que :

- le point de linéarisation a lieu pendant l'exécution de l'opération,
- une modification du contenu de la structure de donnée n'est visible que par les opérations dont le point de linéarisation a lieu après.

En d'autres termes, tout se passe comme si les accès à la structure de données étaient atomiques.

Un fonction pour créer un nouveau thread

Pourriez-vous proposer un type pour une fonction de création de thread en Rust ?

Un fonction pour créer un nouveau thread

Proposition 1 :

```
fn spawn<F>(f: F) where F: FnOnce() -> ()
```

C'est incorrect. Pourquoi ?

Un fonction pour créer un nouveau thread

Proposition 1 :

```
fn spawn<F>(f: F) where F: FnOnce() -> ()
```

C'est incorrect parce que `F` pourrait contenir des références vers le tableau d'activation (stack frame) courant :

```
fn f() {
    let mut x = 0;
    spawn(||{
        let dt = std::time::Duration::from_millis(100);
        std::thread::sleep(dt);

        // Cette clôture contient un emprunt vers x dans le tableau d'activation de l'appelant.
        // On se retrouve donc à déréférencer un emprunt invalide !
        x += 1
    });
}
```

Un fonction pour créer un nouveau thread

Proposition 2 :

```
fn spawn<F>(f: F) where F: FnOnce() -> (),
                        F: 'static
```

- La contrainte `F: 'static` signifie que le *type* de la clôture `F` doit survivre à `'static`.
- `'static` est la lifetime qui ne termine jamais.
- Donc `F: 'static` signifie que le type `F` doit toujours être valide.
- En particulier, `F` ne peut pas contenir de « vrai » emprunt.

Pour s'assurer que le compilateur n'utilise pas d'emprunt dans les clôtures, on a le mot-clé `move` :

```
spawn(move ||{ .... })
```

Les variables capturées sont alors déplacées ou copiées vers la clôture plutôt qu'empruntées.

Un fonction pour créer un nouveau thread

Proposition 2 :

```
fn spawn<F>(f: F) where F: FnOnce() -> (),
                        F: 'static
```

Est-ce correct ?

Un fonction pour créer un nouveau thread

Proposition 2 :

```
fn spawn<F>(f: F) where F: FnOnce() -> (),
                        F: 'static
```

Ceci est incorrect.
À cause de la mutabilité intérieure, on peut créer data race sur des variables non-atomiques :

```
fn f() {
    let x = Rc::new(Cell::new(42));
    let y = x.clone();
    // x et y sont des pointeurs vers le même objet Cell
    std::thread::spawn(move ||{
        y.set(12);
    });
    x.set(13);
}
```

On ne devrait pas être autorisé à partager des pointeurs vers `Cell` entre plusieurs threads !

Le trait Sync

Partager des pointeurs (e.g., `&T` ou `Rc<T>`) vers un type n'est pas anodin.
Cela ne devrait pas être autorisé pour certains types (e.g., `Cell`).

Le trait Sync

Partager des pointeurs (e.g., `&T` ou `Rc<T>`) vers un type n'est pas anodin.
Cela ne devrait pas être autorisé pour certains types (e.g., `Cell`).

Il y a un trait pour ceci : `Sync`.

- `T: Sync` signifie qu'il est sûr de partager un pointeur vers `T` entre threads.
- Il ne déclare aucune méthode, mais affirme simplement une propriété d'un type.
- C'est un trait unsafe : on ne peut pas l'implémenter à la main sans risque.
- C'est un « auto trait » : dans les cas simples, Rust l'implémente automatiquement.
  - Exemple : `struct S<T, U> { t: T, u: U }` est `Sync` ssi à la fois `T` et `U` sont `Sync`.
  - Les clôtures sont `Sync` ssi leur composants sont `Sync`.

Exemples ?

Le trait Sync

Partager des pointeurs (e.g., `&T` ou `Rc<T>`) vers un type n'est pas anodin.
Cela ne devrait pas être autorisé pour certains types (e.g., `Cell`).

Il y a un trait pour ceci : `Sync`.

- `T: Sync` signifie qu'il est sûr de partager un pointeur vers `T` entre threads.
- Il ne déclare aucune méthode, mais affirme simplement une propriété d'un type.
- C'est un trait unsafe : on ne peut pas l'implémenter à la main sans risque.
- C'est un « auto trait » : dans les cas simples, Rust l'implémente automatiquement.
  - Exemple : `struct S<T, U> { t: T, u: U }` est `Sync` ssi à la fois `T` et `U` sont `Sync`.
  - Les clôtures sont `Sync` ssi leur composants sont `Sync`.

Exemples :

- `i32`, `Box<i32>`, `&mut i32` sont `Sync`.
  - Un pointeur partagé vers ces types ne donne qu'un accès en lecture.
- `Cell<i32>` n'est pas `Sync`.
  - Un pointeur partagé peut être utilisé pour écrire.
- `Rc<i32>` et `RefCell<i32>` ne sont pas `Sync`.
  - Les accès au compteur interne seraient des data races.

Le trait Send

De manière similaire, il n'est pas toujours sûr de déplacer une valeur vers d'autres threads.

Il y a un autre trait pour ceci : `Send`.

- `T: Send` signifie qu'il est correcte d'envoyer une valeur de type `T` vers un autre thread.
- Comme `Sync`, c'est un auto trait, unsafe.

Exemples ?

Le trait Send

De manière similaire, il n'est pas toujours sûr de déplacer une valeur vers d'autres threads.
Il y a un autre trait pour ceci : `Send`.

- `T: Send` signifie qu'il est correcte d'envoyer une valeur de type `T` vers un autre thread.
- Comme `Sync`, c'est un auto trait, unsafe.

Exemples :

- On a `T: Sync` ssi `&T: Send`.
  - C'est une instance spécifique de `Send` pour `&T` dans la bibliothèque standard.
  - Donc `&Cell<T>` n'est pas Send.
- `i32`, `Box<i32>`, `&mut i32` sont `Send`.
  - Il n'ont rien de spécifique à un thread.
- `Rc<T>` n'est jamais `Send`.
  - L'accès au compteur interne serait une data race.
- `Cell<T>` et `RefCell<T>` sont `Send` quand `T: Send`.
  - Quand on a l'ownership, personne ne peut accéder au compteur.

Send et Sync : récapitulons

`T: Send` signifie qu'on peut envoyer un object de type `T` à un autre thread.

`T: Sync` signifie qu'on peut partager un object de type `T` à un autre thread.

- C'est équivalent à `&T: Send`.

Send et Sync : récapitulons

`T: Send` signifie qu'on peut envoyer un object de type `T` à un autre thread.

`T: Sync` signifie qu'on peut partager un object de type `T` à un autre thread.

- C'est équivalent à `&T: Send`.

Il faut adapter l'API de `Mutex` :

```
unsafe impl<T: Send> Send for Mutex<T> {}
unsafe impl<T: Send> Sync for Mutex<T> {} // Permet de partager le mutex !

unsafe impl<'a, T: Sync> Sync for MutexGuard<'a, T> {}
// MutexGuard n'est pas Send: on ne peut pas déverrouiller dans un thread différent.
```

Une fonction pour créer un nouveau thread

Proposition 3 :

```
fn spawn<F>(f: F) where F: FnOnce() -> (),
                        F: Send + 'static
```

On demande à ce que la clôture soit `Send` et survive à `'static` de telle façon que les valeurs capturées :

- ne puissent pas référer au tableau d'activation de l'appelant ;
- ne puisse pas contenir d'emprunt qui terminerait un jour (avant la fin du thread) ;
- puissent être déplacées sans risque à un autre thread ;
  - En particulier, `Rc<RefCell<i32>>` n'est pas `Send` !

Une fonction pour créer un nouveau thread

Proposition 3 :

```
fn spawn<F>(f: F) where F: FnOnce() -> (),
                        F: Send + 'static
```

On demande à ce que la clôture soit `Send` et survive à `'static` de telle façon que les valeurs capturées :

- ne puissent pas référer au tableau d'activation de l'appelant ;
- ne puisse pas contenir d'emprunt qui terminerait un jour (avant la fin du thread) ;
- puissent être déplacées sans risque à un autre thread ;
  - En particulier, `Rc<RefCell<i32>>` n'est pas `Send` !

C'est sûr, mais il manque une fonctionnalité importante : récupérer le résultat du calcul concurrent.

Le vrai type de spawn

```
struct JoinHandle<T> { ... }

pub fn spawn<F, T>(f: F) -> JoinHandle<T> where F: FnOnce() -> T,
                                                F: Send + 'static,
                                                T: Send + 'static { ... }

impl<T> JoinHandle<T> {
      /* Result<T> est analogue à Option<T>: une erreur est renvoyée si le thread panique. */
      pub fn join(self) -> Result<T>
      { ... }
}
```

Parallélisme

Le but du parallélisme, c'est d'améliorer les performances d'un programme en effectuant les calculs sur plusieurs unités de calculs en même temps.

Un possibilité : utilisation de threads, que le système d'exploitation va automatiquement exécuter sur des processeurs séparés.

Mais cela ne passe pas forcément par l'utilisation de concurrence avec des threads.

En fait, le processeur fait déjà des calculs en parallèle en essayant d'exécuter plusieurs instructions simultanément lorsque c'est possible.

Parallélisme par SIMD

Certaines instructions du processeur peuvent effectuer simultanément la même opération sur plusieurs opérandes différentes.
Ce sont les instructions SIMD (*single instruction multiple data*).

Il s'agit donc d'un cas où on peut profiter de parallélisme sans avoir de problèmes de concurrence.

Démo (dossier `simd`) !

Parallélisme multithread

Une autre manière d'obtenir des performances avec le parallélisme est d'utiliser plusieurs cœurs.

Cependant, obtenir de réels gains est parfois difficile.
En effet :

- Il est souvent nécessaire de communiquer entre threads.
  - Ces communications requièrent de la synchronisation (mutex, ...) qui est coûteuse en performance.
  - On est vite tenté de protéger des structures de données partagées par des mutex, mais cela limite les performances car un seul thread peut y accéder simultanément. On appelle ce problème la contention.
- Il est parfois difficile d'équilibrer le travail entre threads.
  - On a parfois recours à des systèmes de *work-stealing* permettant de transférer du travail d'un thread à l'autre.
  - Ces bibliothèques permettent de créer beaucoup ($> 10^6$) de threads légers, et se chargent elles-mêmes de les répartir entre cœurs.
  - Exemple : bibliothèque `rayon` en Rust.

La loi d'Amdahl

Un calcul = une part que l'on peut paralléliser, et une part qui doit rester séquentielle.

La loi d'Amdahl permet de quantifier en théorie le gain maximal d'une parallélisation lorsqu'une partie du calcul ne peut être que séquentiel.
L'accélération théorique est :

$$\frac{1}{1-p+\frac{p}{s}}$$

où p est la fraction du calcul que l'on peut paralléliser, et s le nombre de cœurs.

Il faut la garder en tête, car elle limite beaucoup les gains qu'on peut obtenir.
Si $p = 90\%$ et $s = 8$, alors le gain maximal est $x4.7$. Décevant !