Nous avons appris que **Rust** permettait d'écrire des programmes à concurrence rapides et sûrs. C'est lors de cette étude que nous allons découvrir comment y parvenir. Nous verrons tour à tour trois façons d'utiliser les **exétrons** dans Rust : parallélisme **Fork-Join**, les **canaux**, **états mutables partagés**.

*Il va sans dire que ce sera l'occasion de vous servir de tout ce que vous avez appris jusqu'ici au sujet de **Rust**. Le soin avec lequel le langage considère les références, la mutabilité et les durées de vie est précieux dans un programme **mono-exétron**, mais encore plus en programmation concurrente.*

*Les règles correspondantes permettent d'enrichir votre boîte à outils pour exploiter différents styles de codage **multi-exétrons** rapidement et proprement.*

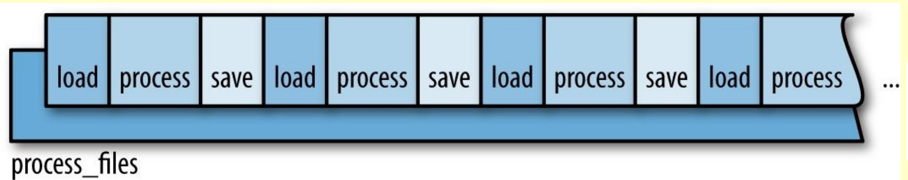
Le parallélisme Fork-Join

Le domaine d'application le plus simple pour dompter plusieurs **exétrons** correspond au lancement de plusieurs tâches totalement indépendantes, que nous voulons voir progresser en même temps.

À titre d'exemple, partons du besoin de traiter du langage humain en exploitant un vaste stock de documents. Nous pourrions commencer par une boucle. Le chronogramme d'exécution ressemblerait alors au visuel ci-dessous :

Processus mono-exétron

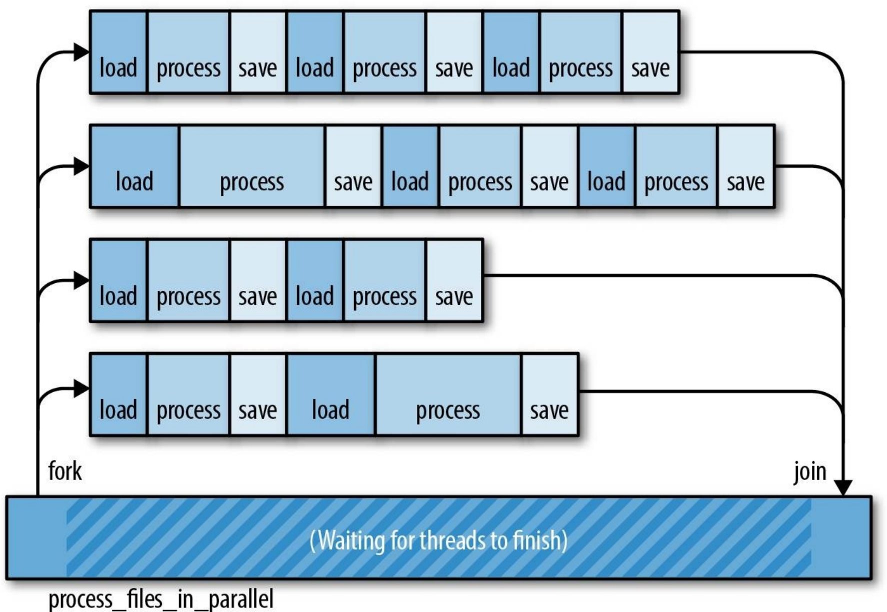
```
fn process_files(nomfics: Vec<String>) -> io::Result<()> {
  for document in nomfics {
    let texte = load(&document)?;          // lecture fichier source
    let resultats = = process(texte);     // Calcul statistiques
    save(&document, resultats)?;          // Écriture fichier cible
  }
  Ok(())
}
```



*Une meilleure approche consisterait à traiter chaque document tour à tour, ce qui permettrait d'accélérer le traitement en répartissant le stock d'entrée en plusieurs blocs, chaque bloc étant traité par un **exétron**.*

*Cette approche correspond au parallélisme **Fork-Join** (bifurcation-convergence).*

*L'étape **Fork** consiste à démarrer un ou plusieurs nouveaux **exétrons**, sans arrêter celui en cours ; l'étape **Join** désigne la phase de regroupement pendant laquelle les **exétrons** qui ont terminé attendent que tous les collègues aient terminés.*



*Le parallélisme **Fork-Join** est d'un grand intérêt pour plusieurs raisons :*

- *Il est extrêmement simple. Il est très facile à mettre en place et **Rust** vous aide à l'utiliser correctement.*
- *Il évite les interblocages. Aucune ressource partagée ne risque d'être verrouillée dans cette approche. Le seul moment où un **exétron** doit attendre un autre est lorsqu'il a terminé avant. Chaque **exétron** progresse au mieux qu'il peut. Le surcoût dû à la commutation entre **exétrons** reste faible.*
- *Les estimations de performances sont aisées. Dans le meilleurs des cas, si nous démarrons quatre **exétrons**, nous devrions achever le traitement quatre fois plus vite. La figure juste ci-dessus montre pourquoi cette estimation n'est pas réaliste, puisque nous ne pouvons pas distribuer le travail de façon équilibrée.*

  *Cette approche **Fork-Join** peut également obliger à consacrer du temps en fin de travail, une fois tous les **exétrons** terminés pour consolider les résultats entre eux. Autrement dit, isoler les tâches crée un travail de collation final supplémentaire.*

  En dehors de ces deux remarques, tout programme répartissant les traitements ainsi profitera d'une accélération significative.

*L'inconvénient principal de **Fork-Join** est que les unités de traitement doivent pouvoir travailler en isolation les unes des autres. Nous découvrirons dans cette étude que certains problèmes ne peuvent pas être distribués aussi nettement.*

Créer une nouvelle tâche avec spawn

Pour créer une nouvelle tâche, nous appelons la fonction **thread::spawn()** et nous lui passons une **clôture** qui contient le code que nous souhaitons exécuter dans la nouvelle tâche. L'exemple suivant affiche du texte à partir de la tâche principale et un autre texte également affiché sur la nouvelle tâche :

Deux tâches simultanées

```
use std::thread::{spawn, sleep};
use std::time::Duration;

fn main() {
  spawn(|| {
    for i in 1..10 {
      println!("Bonjour n°{} à partir de la nouvelle tâche !", i);
      sleep(Duration::from_millis(1));
    }
  });

  for i in 1..5 {
    println!("Bonjour n°{} à partir de la tâche principale !", i);
    sleep(Duration::from_millis(1));
  }
}
```

Résultat

```
Bonjour n°1 à partir de la tâche principale !
Bonjour n°1 à partir de la nouvelle tâche !
Bonjour n°2 à partir de la tâche principale !
Bonjour n°2 à partir de la nouvelle tâche !
Bonjour n°3 à partir de la tâche principale !
Bonjour n°3 à partir de la nouvelle tâche !
Bonjour n°4 à partir de la tâche principale !
Bonjour n°4 à partir de la nouvelle tâche !
```

*Le paramètre d'entrée de la fonction **thread ::spawn()** est une **clôture** sur une fonction. **Rust** lance alors un nouvel **exétron** pour exécuter le code de la clôture. Cet **exétron** est un vrai sous-processus du système d'exploitation avec sa propre pile d'appels.*

Remarquez qu'avec cette fonction, la nouvelle tâche s'arrête lorsque la tâche principale s'arrête, qu'elle ait fini ou non de s'exécuter. La sortie de ce programme peut être différente à chaque fois, mais il devrait ressembler au résultat ci-dessus.

*L'appel à **thread::sleep()** force une tâche à mettre en pause son exécution pendant une petite durée, permettant à une autre tâche de s'exécuter. Les tâches se relaieront probablement, mais ce n'est pas garanti : cela dépend de comment votre système d'exploitation agence les tâches.*

*Lors de cette exécution, la tâche principale a d'abord écris, même si l'instruction d'écriture de la nouvelle tâche apparaît en premier dans le code. Et même si nous avons demandé à la nouvelle tâche d'écrire jusqu'à ce que **i** vaut **9**, elle l'a fait seulement jusqu'à **4**, juste au moment où la tâche principale s'arrête.*

Attendre que toutes les tâches aient finies en utilisant join

Le code précédent, non seulement stoppe la nouvelle tâche prématurément la plupart du temps à cause de la fin de la tâche principale, mais elle ne garantit pas non plus que la nouvelle tâche va s'exécuter une seule fois. La raison à cela est qu'il n'y a pas de garantie sur l'ordre dans lequel les tâches vont s'exécuter !

*Nous pouvons régler le problème des nouvelles tâches qui ne s'exécutent pas, ou pas complètement, en sauvegardant la valeur de retour de **thread::spawn()** dans une variable. Le type de retour de **thread::spawn()** est **JoinHandle**.*

*Un **JoinHandle** est une valeur possédée qui, lorsque nous appelons la méthode **join()** sur elle, va attendre que ses tâches finissent. L''exemple suivant nous montre comment utiliser le **JoinHandle** de la tâche que nous venons de créer en appelant la méthode **join()** pour s'assurer que la nouvelle tâche finisse bien avant que la fonction principale **main()** se termine :*

La tâche principale attend que la tâche secondaire se termine complètement

```
use std::thread::{spawn, sleep};
use std::time::Duration;

fn main() {
  let tache = spawn(|| {
    for i in 1..10 {
      println!("Bonjour n°{} à partir de la nouvelle tâche !", i);
      sleep(Duration::from_millis(1));
    }
  });

  for i in 1..5 {
    println!("Bonjour n°{} à partir de la tâche principale !", i);
    sleep(Duration::from_millis(1));
  }
```

```
  tache.join().unwrap();
}
```

Résultat

```
Bonjour n°1 à partir de la tâche principale !
Bonjour n°1 à partir de la nouvelle tâche !
Bonjour n°2 à partir de la tâche principale !
Bonjour n°2 à partir de la nouvelle tâche !
Bonjour n°3 à partir de la tâche principale !
Bonjour n°3 à partir de la nouvelle tâche !
Bonjour n°4 à partir de la tâche principale !
Bonjour n°4 à partir de la nouvelle tâche !
Bonjour n°5 à partir de la nouvelle tâche !
Bonjour n°6 à partir de la nouvelle tâche !
Bonjour n°7 à partir de la nouvelle tâche !
Bonjour n°8 à partir de la nouvelle tâche !
Bonjour n°9 à partir de la nouvelle tâche !
```

*L'appel de la fonction **join()** bloque la tâche principale qui s'exécute actuellement jusqu'à ce que la tâche secondaire se termine. Bloquer une tâche signifie que cette tâche est obligée d'accomplir le travail demandé ou de se terminer prématurément afin que le programme ne reste pas indéfiniment bloqué.*

*Dans notre exemple, les deux tâches continuent à alterner, mais la tâche principale attends à cause de l'appel à **tache.join()** et ne se termine pas avant que la nouvelle tâche soit finie. Mais voyons maintenant ce qui se passe lorsque nous déplaçons le **tache.join()** avant la boucle **for** du **main()** comme ceci :*

Attente prématurée

```
use std::thread::{spawn, sleep};
use std::time::Duration;

fn main() {
  let tache = spawn(|| {
    for i in 1..10 {
      println!("Bonjour n°{} à partir de la nouvelle tâche !", i);
      sleep(Duration::from_millis(1));
    }
  });

  tache.join().unwrap();

  for i in 1..5 {
    println!("Bonjour n°{} à partir de la tâche principale !", i);
    sleep(Duration::from_millis(1));
  }
}
```

Résultat

```
Bonjour n°1 à partir de la nouvelle tâche !
Bonjour n°2 à partir de la nouvelle tâche !
Bonjour n°3 à partir de la nouvelle tâche !
Bonjour n°4 à partir de la nouvelle tâche !
Bonjour n°5 à partir de la nouvelle tâche !
Bonjour n°6 à partir de la nouvelle tâche !
Bonjour n°7 à partir de la nouvelle tâche !
Bonjour n°8 à partir de la nouvelle tâche !
Bonjour n°9 à partir de la nouvelle tâche !
Bonjour n°1 à partir de la tâche principale !
Bonjour n°2 à partir de la tâche principale !
Bonjour n°3 à partir de la tâche principale !
Bonjour n°4 à partir de la tâche principale !
```

*Vu que la position de la méthode **join()** est placée en amont, la tâche principale attend que la nouvelle tâche se finisse et ensuite exécute sa propre boucle **for**, ainsi la sortie n'est plus du tout chevauchée, la **sous-tâche** est effectuée en premier et lorsque qu'elle a finie son travail, c'est la **tâche principale** qui prend le relais.*

Uiliser les clôtures « move » avec les tâches (données immuables)

La clôture **« move »** est souvent utilisé avec **thread::spawn()** car elle vous permet d'utiliser une donnée d'une tâche dans une autre tâche. Nous pouvions utiliser le mot-clé **move** avant la liste des paramètres d'une clôture pour la forcer à prendre possession des valeurs de son environnement qu'elle utilise. Cette technique est particulièrement utile lorsque nous créons des nouvelles tâches pour pouvoir transférer la possession des valeurs d'une tâche vers une autre.

*Dans tous les exemples précédents, la **clôture** associée à la tâche secondaire ne possède aucun argument : nous n'utilisons donc aucune donnée issue de la tâche principale dans le code de cette tâche secondaire. Pour utiliser des données de la tâche principale dans la nouvelle tâche, la clôture de la nouvelle tâche doit capturer les valeurs dont elle a besoin. L'exemple*

suivant montre une tentative de création d'un vecteur dans la tâche principale et l'utilisation dans la tâche secondaire. Cependant, cela ne fonctionne pas encore, comme vous allez le constater bientôt.

Tentative de récupération des données extérieures à la tâche

```
use std::thread::spawn;

fn main() {
  let donnees = vec![1, 2, 3];

  let tache = spawn(|| {
    println!("Voici nos données : {:?}", donnees);
  });

  tache.join().unwrap();
}
```

Rust

```
  --> src/main.rs:6:21
   |
6 |   let tache = spawn(| | {
   |                             ^^ may outlive borrowed value `donnees`
7 |     println!("Voici nos données : {:?}", donnees);
   |                                                          ------- `donnees` is borrowed here
   |
note: function requires argument type to outlive `'static`
  --> src/main.rs:6:15
   |
6 |     let tache = spawn(|| {
   |  _______________^
7 | |     println!("Voici nos données : {:?}", donnees);
8 | |   });
   | |____^
help: to force the closure to take ownership of `donnees` (and any other referenced variables), use the `move` keyword
   |
6 |   let tache = spawn(move | | {
   |                     ^^^^^^^
```

*Au cours de cet exemple, **Rust** déduit comment capturer **donnees**, et comme **println!()** n'a besoin que d'une référence à **donnees**, la **clôture** essaie d'emprunter cette variable. Cependant, il y a un petit problème : **Rust** ne peut pas savoir combien de temps la tâche va s'exécuter, donc il ne peut pas savoir si la référence à **donnees** sera toujours valide.*

*Pour corriger l'erreur de compilation, nous pouvons appliquer le conseil du message d'erreur. En ajoutant le mot-clé **move** avant la **clôture**, nous forçons la **clôture** à prendre possession des valeurs qu'elle utilise au lieu de laisser **Rust** en déduire qu'il doit emprunter les valeurs.*

Changement de possession explicite

```
use std::thread::spawn;

fn main() {
  let donnees = vec![1, 2, 3];

  let tache = spawn(move || {
    println!("Voici nos données : {:?}", donnees);
  });

  tache.join().unwrap();
}
```

Résultat

```
Voici nos données : [1, 2, 3]
```

Par contre, vu que le vecteur est maintenant possédé par la clôture, le programme principal ne peut plus utiliser cette variable. L'exemple ci-dessous ne peut être compilé à cause de cette remarque.

Changement de possession explicite

```
use std::thread::spawn;

fn main() {
  let donnees = vec![1, 2, 3];
  let tache = spawn(move || {
    println!("Voici nos données : {:?}", donnees);
  });
  println!("Voici nos données : {:?}", donnees);
```

```
  tache.join().unwrap();
}
```

Comptage de références atomiques - Arc

Si nous désirons résoudre cette problématique, nous pouvons proposer une autre approche grâce à la librairie standard avec le comptage de références atomiques représenté par **Arc** qui implémente le trait **Sync**.

Comptage de références atomiques

```
use std::thread::spawn;
use std::sync::Arc;

fn main() {
  let mut donnees = Arc::new(vec![1, 2, 3]);
  let donnees_sync = donnees.clone();

  let tache = spawn(move || {
    println!("Voici nos données : {:?}", donnees_sync);
  });

  println!("Voici nos données : {:?}", donnees);
  tache.join().unwrap();
}
```

Résultat

```
Voici nos données : [1, 2, 3]
Voici nos données : [1, 2, 3]
```

*Vous remarquez que nous avons modifié le type de **donnees** en l'englobant de la structure **Arc**. Grâce à cela, la méthode **clone()** clone seulement **Arc** et actualise le compteur de références, sans toucher le vecteur lui-même. Le clonage permet ainsi de créer une copie du pointeur intelligent **Arc** et non de tout le vecteur, ce qui se résume à augmenter de **1** le compteur de références.*

*Grâce à cette retouche, notre programme fonctionne maintenant parfaitement bien, car il ne dépend plus des durées de vie des références. Tant qu'au moins un **exétron** possède toujours un **Arc<Vec<i32>>**, le vecteur reste actif même si le parent se termine éventuellement plus tôt. Aucun conflit d'accès aux données n'est possible parce que les données d'un **Arc** sont immuables.*

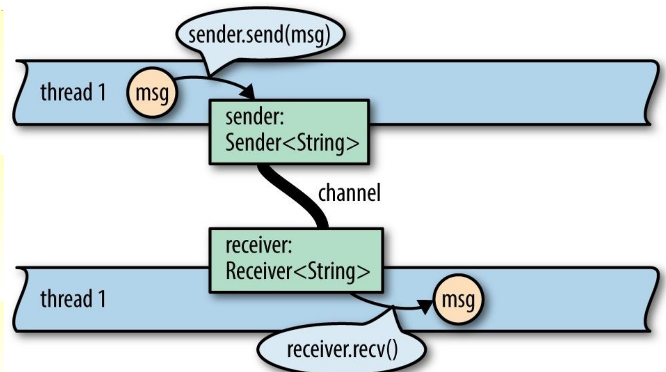
Les canaux (channels)

Un canal est un **conduit unidirectionnel** qui sert à envoyer des valeurs d'un **exétron** à un autre. Autrement dit, c'est une queue de communication **multi-exétrons**, donc apte à la parallélisation.

*Le concept s'apparente à celui des **pipes Unix** : un bout envoie des données et l'autre les reçoit, les deux bouts étant normalement possédés par deux **exétrons** différents. Les **pipes Unix** servent à transmettre des octets alors que les **canaux** transmettent des valeurs **Rust**.*

*La méthode **sender.send(msg)** dépose une valeur dans le canal et **receiver.recv()** en prélève une. La possession des valeurs passent de **l'exétron émetteur** vers **l'exétron récepteur**. Si le canal est vide, **receiver.recv()** reste bloqué en attente d'une valeur.*

Les canaux permettent donc aux exétrons d'échanger des valeurs, ce qui offre une coopération facile sans verrouillage, ni mémoire partagée.



À ce niveau, je vous propose de concevoir un programme qui possède une tâche pour générer des valeurs et les envoyer dans un canal, et une autre tâche qui reçoit ces valeurs et les affiche. Nous allons envoyer de simples valeurs entre les tâches en utilisant un canal pour illustrer cette fonctionnalité.

Une fois que vous serez familiarisée avec cette technique, vous pourrez utiliser les canaux pour créer un système de dialogue en ligne ou un système où de nombreuses tâches font chacune une partie d'un gros calcul et envoient leur résultat à une tâche chargée d'agréger ces résultats.

Communication par canaux

```
use std::thread::spawn;
use std::sync::mpsc::channel;

fn main() {
  let (emetteur, recepteur) = channel();
  let tache = spawn(move || {
    let reception = recepteur.recv().unwrap();
    println!("Message reçu ({})", reception);
  });
```

```
  let message = String::from("Bienvenue!");
  println!("Message envoyé ({})", message);
  emetteur.send(message).unwrap();
  tache.join().unwrap();
}
```

Résultat

```
Message envoyé (Bienvenue!)
Message reçu (Bienvenue!)
```

*Nous créons un nouveau canal en utilisant la fonction **mpsc::channel()** ; **mpsc** signifie **multiple producer, single consumer**, c'est-à-dire plusieurs producteurs, un seul consommateur. En bref, la façon dont la bibliothèque standard de **Rust** implémente ces canaux permet d'avoir plusieurs extrémités émettrices qui produisent des valeurs, mais seulement une seule extrémité réceptrice qui consomme ces valeurs.*

*La fonction **mpsc::channel()** retourne un **tuple**, le premier élément est celui qui permet d'envoyer des valeurs et le second est celui qui les reçoit.*

*Nous utilisons à nouveau **thread::spawn()** pour créer une nouvelle tâche et ensuite nous utilisons **move** pour permettre le déplacement de **reception** dans la **clôture** afin que la nouvelle tâche possède bien cette variable. La nouvelle tâche a besoin de posséder la partie réceptrice du canal afin d'être en capacité de recevoir des messages et d'afficher le texte correspondant. Tant que la valeur n'est pas envoyée, la tâche reste bloquée jusqu'à la réception du texte dans ce canal.*

*La partie émettrice possède la méthode **send()** qui prend en argument la valeur que nous souhaitons envoyer. La méthode **send()** retourne un type **Result<T, E>**, donc si la partie réceptrice a déjà été libérée et qu'il n'y a nulle part où envoyer la valeur, l'opération d'envoi va retourner une erreur.*

*Dans cet exemple, nous faisons appel à **unwrap()** pour activer une panique en cas d'erreur. Mais dans un vrai programme, nous devrions gérer ce cas correctement.*

*La partie réception d'un canal possède deux modes intéressants : **recv()** et **try_recv()**. Nous utilisons ici **recv()**, un raccourci pour recevoir, qui va bloquer l'exécution de la tâche principale et attendre jusqu'à ce qu'une valeur soit envoyée dans le canal. Une fois qu'une valeur est envoyée, **recv()** la retourne dans un **Result<T, E>**. Lorsque la partie transmission du canal se ferme, **recv()** retourne alors une erreur pour signaler qu'il n'y aura plus de valeurs qui pourrons arriver.*

*La méthode **try_recv()** elle ne bloque pas, mais retourne immédiatement un **Result<T, E>** : une valeur **Ok()** qui contient un message s'il y en a un de disponible, et une valeur **Err()** si au moment de la lecture il n'y a pas de message.*

*L'utilisation de **try_recv()** est bien pratique si cette tâche à d'autres choses à faire pendant qu'elle attend les messages. Nous pouvons ainsi écrire une boucle qui appelle régulièrement **try_recv()**, gère le message s'il y en a un, et sinon fait d'autres choses à l'attendant.*

*Nous avons utilisé **recv()** dans cet exemple pour des raisons de simplicité ; nous n'avons rien d'autres à faire dans la tâche d'affichage que d'attendre les messages venant de la tâche principale.*

Envoyer plusieurs valeurs et voir le récepteur les attendre

Je propose maintenant d'utiliser les canaux pour communiquer avec plusieurs messages envoyés au cours du temps. Cette fois-ci, la tâche principale possède un vecteur de chaînes de caractères que nous souhaitons envoyer à la tâche secondaire. Nous envoyons individuellement chaque message par itération, et nous faisons une pause entre chaque envoi avec la fonction **thread::sleep()** pour une durée de **1** seconde.

Plusieurs message dans le même canal

```
use std::thread::{spawn, sleep};
use std::sync::mpsc::channel;
use std::time::Duration;

fn main() {
  let (emetteur, recepteur) = channel();

  let tache = spawn(move || {
    for message in recepteur {
      if message=="fin" { break; }
      println!("{}", message);
    }
    println!("Tâche secondaire terminée!");
  });

  let messages = vec![
    String::from("Bienvenue"),
    String::from("à partir"),
    String::from("de la nouvelle tâche"),
    String::from("fin")
  ];

  for message in messages {
    emetteur.send(message).unwrap();
    sleep(Duration::from_secs(1));
  }
```

```
  tache.join().unwrap();
}
```

Résultat

```
Bienvenue
à partir
de la nouvelle tâche
Tâche secondaire terminée!
```

*Dans la tâche secondaire, nous n'appelons plus explicitement la fonction **recv()** : à la place, nous utilisons directement **recepteur** comme **un itérateur**. Pour chaque valeur reçue, nous l'affichons. C'est le message **« fin »** qui clôture l'itération.*

Créer plusieurs producteurs en clonant le transmetteur

Précédemment, nous avions évoqué que **mpsc** était un acronyme pour **multiple producer, single consumer**. Mettons **mpsc** en œuvre en élargissant le code précédents pour créer plusieurs tâches qui vont toutes envoyer des valeurs au même récepteur. Nous pouvons faire ceci en clonant la partie émettrice du canal.

*Afin de bien expérimenter ce principe, ce sont cette fois-ci les tâches secondaires qui vont émettre les messages. Avant de créer la première nouvelle tâche, nous appelons **clone()** sur la partie émettrice du canal. Cela nous donne un nouveau transmetteur que nous pourrons passer à la seconde nouvelle tâche. Nous avons donc deux tâches, chacune envoyant des messages différents à la partie réceptrice du canal.*

Plusieurs message dans le même canal

```
use std::thread::{spawn, sleep};
use std::sync::mpsc::{channel, Sender};
use std::time::Duration;

fn main() {
  let (emetteur, recepteur) = channel();
  let emetteur2 = Sender::clone(&emetteur);

  spawn(move || {
    let valeurs = vec![
      String::from("salutations"),
      String::from("à partir"),
      String::from("de la"),
      String::from("nouvelle tâche"),
    ];

    for valeur in valeurs {
      emetteur2.send(valeur).unwrap();
      sleep(Duration::from_secs(1));
    }
  });

  spawn(move || {
    let valeurs = vec![
      String::from("encore plus"),
      String::from("de messages"),
      String::from("pour"),
      String::from("vous"),
    ];

    for valeur in valeurs {
      emetteur.send(valeur).unwrap();
      sleep(Duration::from_secs(1));
    }
  });

  for recu in recepteur {
    println!("Réception : {}", recu);
  }
}
```

Résultat

```
Réception : salutations
Réception : encore plus
Réception : à partir
Réception : de messages
Réception : de la
Réception : pour
Réception : vous
Réception : nouvelle tâche
```

Utiliser les mutex pour permettre l'accès à des données mutables

Mutex est une abréviation pour **mutual exclusion**, ce qui veut dire qu'un **mutex** ne permet qu'à une seule tâche d'accéder à une donnée en même temps. Pour accéder à la donnée dans un **mutex**, une tâche doit d'abord signaler qu'elle souhaite y accéder en demandant l'obtention du **verrou** du **mutex**.

> *Le verrou est une structure de données qui fait partie du **mutex** qui assure le suivi de qui possède actuellement l'accès à la donnée. Par conséquent, le **mutex** est qualifié de gardien de la donnée qu'il renferme via le système de **verrou**.*

> ***Pour conclure, un verrou mutex (mutuelle exclusion) est un mécanisme qui oblige plusieurs exétrons à se mettre dans une file d'attente pour accéder à des données partagées par l'ensemble.***

> *Les **mutex** ont la réputation d'être difficile à utiliser car vous devez veiller à deux règles :*
> - *Vous devez obtenir le verrou avant d'utiliser la donnée.*
> - *Lorsque vous avez fini avec la donnée que le **mutex** garde, vous devez déverrouiller la donnée afin que d'autres tâches puissent obtenir le verrou.*

L'API des Mutex<T>

Pour illustrer l'utilisation d'un **mutex**, commençons par l'utiliser dans le contexte d'une seule tâche, comme dans l'exemple ci-dessous. Comme avec beaucoup de types, nous créons un **Mutex<T>** en utilisant la méthode associée **new()**. Pour accéder à la donnée dans le **mutex**, nous utilisons la méthode **lock()** pour obtenir le **verrou**. Cela bloque la tâche courante, elle ne s'exécutera plus tant que ce n'est pas à notre tour d'avoir le **verrou**.

> ***L'appel à lock() échoue dans le cas où une autre tâche qui avait le verrou panique. Dans ce cas, personne ne pourra obtenir le verrou, donc nous avons choisi d'utiliser unwrap() pour faire en sorte que cette tâche panique si elle est dans cette situation.***

> *Après avoir obtenu le **verrou**, nous pouvons utiliser la valeur de retour comme **une référence mutable vers la donnée**, qui s'appelle **nombre** dans notre cas. Le système de type s'assure que nous obtenons le **verrou** avant d'utiliser la valeur présente dans mutex : le **Mutex<i32>** n'est pas un **i32**, donc nous devons obtenir le **verrou** afin de pouvoir utiliser la valeur **i32**. Nous ne pouvons pas l'oublier ; le système de type ne nous laissera pas accéder au **i32** à l'intérieur de toute façon.*
>
> *Comme vous pouvez vous en douter, **Mutex<T>** est un pointeur intelligent. Plus précisément, l'appel à **lock()** retourne un pointeur intelligent **MutexGuard**, intégré dans un **LockResult** que nous gérons en faisant appel à **unwrap()**.*
>
> *Le pointeur intelligent **MutexGuard** implémente **Deref** pour pointer sur la donnée interne ; ce pointeur intelligent implémente aussi **Drop** qui libère le **verrou** automatiquement lorsqu'un **MutexGuard** sort de la portée. Au final, nous ne risquons pas d'oublier de rendre le **verrou** et ainsi de bloquer l'utilisation du **mutex** par les autres tâches car la libération du **verrou** se produit automatiquement.*
>
> *Après avoir libéré le verrou, nous pouvons afficher la valeur dans le **mutex** et constater que nous avons pu changer la valeur interne du **i32** à **6**.*

Création d'un mutex sur un entier

```
use std::sync::Mutex;

fn main() {
  let mutex = Mutex::new(5);
  {
    let mut nombre = mutex.lock().unwrap();
    *nombre += 1;
  }
  println!("m = {:?}", mutex);
}
```

Résultat

```
m = Mutex { data: 6, poisoned: false, .. }
```

Partagé un Mutex<T> entre plusieurs tâches

Essayons maintenant de partager une valeur entre plusieurs tâches en utilisant **Mutex<T>**. Nous allons faire fonctionner 10 tâches et faire en sorte que chacune augmente la valeur du compteur de **1**, donc le compteur va passer de **0** à **10**. L'exemple suivant déboucher sur une erreur de compilation, et nous allons utiliser cette erreur pour en apprendre plus sur l'utilisation de **Mutex<T>** et voir comment **Rust** nous aide à l'utiliser correctement.

Création d'un mutex sur un entier avec plusieurs tâches secondaires

```
use std::sync::Mutex;
use std::thread;

fn main() {
  let compteur = Mutex::new(0);
  let mut taches = vec![];

  for _ in 0..10 {
    let tache = thread::spawn(move || {
      let mut nombre = compteur.lock().unwrap();
```

```
    *nombre += 1;
   });
   taches.push(tache);
  }

  for tache in taches {
   tache.join().unwrap();
  }

  println!("Resultat : {}", *compteur.lock().unwrap());
}
```

Résultat

```
  --> src/main.rs:9:31
    |
5   |  let compteur = Mutex::new(0);
    |     ------ move occurs because compteur has type Mutex<i32> which does not implement the `Copy` trait
...
9   |   let tache = thread::spawn(move || {
    |                             ^^^^ value moved into closure here, in previous iteration of loop
10  |     let mut nombre = compteur.lock().unwrap();
    |                      -------- use occurs due to use in closure
```

*Dans cet exemple, nous créons une variable **compteur** pour stocker un **i32** dans un **Mutex<T>**. Ensuite, nous créons **10** tâches en les itérant sur un interval de nombres. Comme au préalable, nous utilisons **thread::spawn()** et nous donnons à toutes les tâches la même **clôture**, qui déplace le **compteur** dans la tâche, obtient le **verrou** sur le **Mutex<T>** en faisant appel à la méthode **lock()**, et ajoute ensuite **1** à la valeur présente dans le **mutex**. Lorsqu'une tâche finit d'exécuter sa **clôture**, **nombre** sort de la portée et libère le **verrou** afin qu'une autre tâche puisse l'obtenir.*

*Le message d'erreur signale que la valeur **compteur** a été déplacée dans l'itération précédente de la boucle. Donc **Rust** nous explique qu'il ne peut pas déplacer la possession du **verrou** de **compteur** dans plusieurs tâches. Corrigeons cette erreur de compilation avec une méthode qui permet d'avoir plusieurs propriétaires.*

Plusieurs propriétaires avec plusieurs tâches – compteur de référence atomique avec Arc<T>

Nous avons déjà rencontré cette problématique lors de notre étude. Comme nous l'avons déjà vu, la solution consiste à utiliser le **compteur de référence atomique** et de passer par un clonage du **mutex** (comptage d'une référence supplémentaire sans toucher à la valeur numérique elle-même).

Création d'un mutex sur un entier avec plusieurs tâches secondaires

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
  let compteur = Arc::new(Mutex::new(0));
  let mut taches = vec![];

  for _ in 0..10 {
   let compteur = Arc::clone(&compteur);
   let tache = thread::spawn(move || {
    let mut nombre = compteur.lock().unwrap();
    *nombre += 1;
   });
   taches.push(tache);
  }

  for tache in taches {
   tache.join().unwrap();
  }
  println!("Resultat : {}", *compteur.lock().unwrap());
}
```

Résultat

```
Resultat : 10
```

*Nous y sommes arrivés ! Nous avons compté de **0** à **10**, ce qui ne semble pas très impressionnant, mais cela nous a appris beaucoup sur **Mutex<T>** et la sécurité entre les tâches. Vous pouvez aussi utiliser cette structure de programme pour procéder à des opérations plus complexes que simplement incrémenter un compteur.*

*En utilisant cette stratégie, vous pouvez diviser un calcul en différentes parties, répartir ces parties sur des tâches, et ensuite utiliser un **Mutex<T>** pour faire en sorte que chaque tâche mette à jour le résultat final avec sa propre partie.*

Vous avez peut-être constaté que compteur est immuable mais que nous pouvons obtenir une référence mutable vers la valeur qu'il renferme ; cela signifie que Mutex<T> possède une mutabilité interne, bien pratique pour ce genre de situation.

Un Mutex possède sa botte secrète : le verrou. À vrai dire, un mutex revient essentiellement à permettre un accès exclusif mut aux données contenues, quand bien même plusieurs exétrons possèdent le même accès partagés (non mut) aux mutex lui-même.