Chapitre 19. Concurrence

> *À long terme, il n'est pas conseillé d'écrire de grands programmes simultanés dans des langages orientés machine qui permettent une utilisation sans restriction des emplacements de magasin et de leurs adresses. Il n'y a tout simplement aucun moyen de rendre de tels programmes fiables (même à l'aide de mécanismes matériels compliqués).*
>
> —Per Brinch Hansen (1977)

> *Les modèles de communication sont des modèles de parallélisme.*
>
> —Whit Morriss

Si votre attitude à l'égard de la concurrence a changé au cours de votre carrière, vous n'êtes pas seul. C'est une histoire commune.

Au début, écrire du code simultané est facile et amusant. Les outils (threads, verrous, files d'attente, etc.) sont faciles à prendre en main et à utiliser. Il y a beaucoup de pièges, c'est vrai, mais heureusement vous savez ce qu'ils sont tous, et vous faites attention à ne pas faire d'erreurs.

À un moment donné, vous devez déboguer le code multithread de quelqu'un d'autre, et vous êtes obligé de conclure que *certaines* personnes ne devraient vraiment pas utiliser ces outils.

Ensuite, à un moment donné, vous devez déboguer votre propre code multithread.

L'expérience inculque un scepticisme sain, sinon un cynisme pur et simple, envers tout le code multithread. Ceci est aidé par l'article occasionnel expliquant en détail pourquoi un idiome multithreading évidemment correct ne fonctionne pas du tout. (Cela a à voir avec « le modèle de mémoire. ») Mais vous finissez par trouver une approche de la concurrence que vous pensez pouvoir utiliser de manière réaliste sans faire constamment d'erreurs. Vous pouvez mettre à peu près tout dans cet idiome, et (si vous êtes *vraiment* bon) vous apprenez à dire « non » à la complexité supplémentaire.

Bien sûr, il y a plutôt beaucoup d'idiomes. Les approches que les programmeurs de systèmes utilisent couramment sont les suivantes :

- Un *thread d'arrière-plan* qui a un seul travail et se réveille périodiquement pour le faire.

- Pools de travail à usage général qui communiquent avec *les* clients via *des files d'attente de tâches*.
- *Pipelines* où les données circulent d'un thread à l'autre, chaque thread faisant un peu de travail.
- *Parallélisme des données*, où l'on suppose (à tort ou à raison) que l'ensemble de l'ordinateur ne fera principalement qu'un seul grand calcul, qui est donc divisé en *n* morceaux et exécuté sur *n* threads dans l'espoir de faire fonctionner tous les *n* cœurs de la machine en même temps.
- *Une mer d'objets synchronisés*, où plusieurs threads ont accès aux mêmes données, et les races sont évitées en utilisant des schémas de *verrouillage* ad hoc basés sur des primitives de bas niveau comme les mutex. (Java inclut un support intégré pour ce modèle, qui était très populaire dans les années 1990 et 2000.)
- *Les opérations d'entier atomique* permettent à plusieurs cœurs de communiquer en transmettant des informations à travers des champs de la taille d'un mot machine. (C'est encore plus difficile à obtenir que tous les autres, à moins que les données échangées ne soient littéralement que des valeurs entières. En pratique, il s'agit généralement de pointeurs.)

Avec le temps, vous pourrez peut-être utiliser plusieurs de ces approches et les combiner en toute sécurité. Vous êtes un maître de l'art. Et les choses seraient géniales, si seulement personne d'autre n'était jamais autorisé à modifier le système de quelque manière que ce soit. Les programmes qui utilisent bien les threads sont pleins de règles non écrites.

Rust offre une meilleure façon d'utiliser la concurrence, non pas en forçant tous les programmes à adopter un seul style (ce qui, pour les programmeurs de systèmes, ne serait pas du tout une solution), mais en prenant en charge plusieurs styles en toute sécurité. Les règles non écrites sont écrites (dans du code) et appliquées par le compilateur.

Vous avez entendu dire que Rust vous permet d'écrire des programmes sûrs, rapides et simultanés. C'est le chapitre où nous vous montrons comment c'est fait. Nous couvrirons trois façons d'utiliser les fils Rust :

- Parallélisme fourche-jointure
- Canaux
- État mutable partagé

En cours de route, vous allez utiliser tout ce que vous avez appris jusqu'à présent sur le langage Rust. Le soin que Rust prend avec les références, la mutabilité et les durées de vie est suffisamment précieux dans les programmes à thread unique, mais c'est dans la programmation simultanée

que la véritable signification de ces règles devient évidente. Ils permettent d'élargir votre boîte à outils, de pirater plusieurs styles de code multithread rapidement et correctement, sans scepticisme, sans cynisme, sans peur.

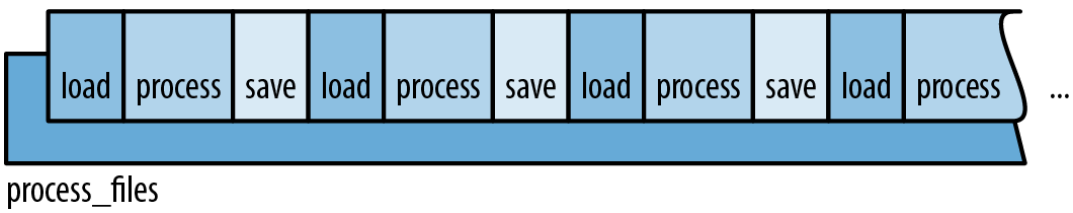
Parallélisme fourche-jointure

Les cas d'utilisation les plus simples pour les threads surviennent lorsque nous avons plusieurs tâches complètement indépendantes que nous aimerions effectuer à la fois.

Par exemple, supposons que nous fassions du traitement du langage naturel sur un grand corpus de documents. Nous pourrions écrire une boucle :

```
fn process_files(filenames: Vec<String>) -> io::Result<()> {
    for document in filenames {
        let text = load(&document)?;  // read source file
        let results = process(text);  // compute statistics
        save(&document, results)?;    // write output file
    }
    Ok(())
}
```

Le programme s'exécuterait comme illustré à la figure 19-1.



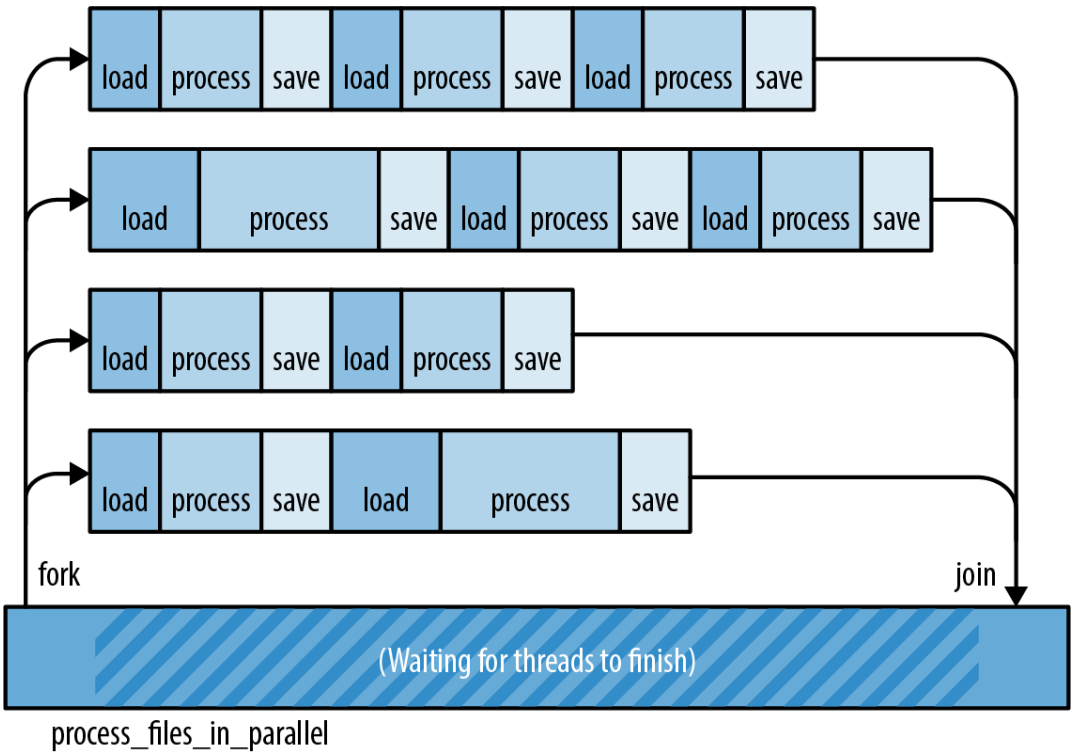
Graphique 19-1. Exécution monothread de `process_files()`

Étant donné que chaque document est traité séparément, il est relativement facile d'accélérer cette tâche en divisant le corpus en morceaux et en traitant chaque morceau sur un fil distinct, comme le montre la figure 19-2.

Ce modèle est appelé *parallélisme fourche-jointure*. *Forker*, c'est commencer un nouveau thread, et *joindre* un thread, c'est attendre qu'il se termine. Nous avons déjà vu cette technique : nous l'avons utilisée pour accélérer le programme Mandelbrot au chapitre 2.

Le parallélisme fourche-jointure est attrayant pour plusieurs raisons :

- C’est très simple. Fork-join est facile à mettre en œuvre, et Rust le rend facile à obtenir.
- Il évite les goulots d’étranglement. Il n’y a pas de verrouillage des ressources partagées dans fork-join. Le seul moment où un fil doit attendre un autre est à la fin. En attendant, chaque thread peut fonctionner librement. Cela permet de maintenir les frais généraux de changement de tâche bas.
- Le calcul de la performance est simple. Dans le meilleur des cas, en démarrant quatre fils, nous pouvons terminer notre travail en un quart du temps. La figure 19-2 montre une raison pour laquelle nous ne devrions pas nous attendre à cette accélération idéale : nous pourrions ne pas être en mesure de répartir le travail uniformément sur tous les fils. Une autre raison de prudence est que parfois les programmes de jointure de fourche doivent passer un certain temps après la jonction des threads *à combiner* les résultats calculés par les threads. Autrement dit, isoler complètement les tâches peut faire un peu de travail supplémentaire. Pourtant, en dehors de ces deux choses, tout programme lié au processeur avec des unités de travail isolées peut s’attendre à un coup de pouce significatif.
- Il est facile de raisonner sur l’exactitude du programme. Un programme de jointure de fourche est *déterministe* tant que les threads sont vraiment isolés, comme les threads de calcul dans le programme Mandelbrot. Le programme produit toujours le même résultat, quelles que soient les variations de vitesse de thread. C’est un modèle de concurrence sans conditions de concurrence.



Graphique 19-2. Traitement de fichiers multithread à l’aide d’une approche de jointure à fourche

Le principal inconvénient de la fourche-jointure est qu'elle nécessite des unités de travail isolées. Plus loin dans ce chapitre, nous examinerons certains problèmes qui ne se divisent pas aussi proprement.

Pour l'instant, restons avec l'exemple du traitement du langage naturel. Nous allons montrer quelques façons d'appliquer le modèle de jointure de fourche à la fonction. `process_files`

spawn et rejoindre

La fonction démarre un nouveau thread : `std::thread::spawn`

```
use std::thread;

thread::spawn(|| {
    println!("hello from a child thread");
});
```

Il faut un argument, une fermeture ou une fonction. Rust démarre un nouveau thread pour exécuter le code de cette fermeture ou fonction. Le nouveau thread est un véritable thread de système d'exploitation avec sa propre pile, tout comme les threads en C++, C# et Java. `FnOnce`

Voici un exemple plus substantiel, utilisé pour implémenter une version parallèle de la fonction d'avant: `spawn process_files`

```
use std::{thread, io};

fn process_files_in_parallel(filenames: Vec<String>) -> io::Result<()> {
    // Divide the work into several chunks.
    const NTHREADS: usize = 8;
    let worklists = split_vec_into_chunks(filenames, NTHREADS);

    // Fork: Spawn a thread to handle each chunk.
    let mut thread_handles = vec![];
    for worklist in worklists {
        thread_handles.push(
            thread::spawn(move || process_files(worklist))
        );
    }

    // Join: Wait for all threads to finish.
    for handle in thread_handles {
        handle.join().unwrap()?;
    }
```

```
    Ok(())
}
```

Prenons cette fonction ligne par ligne.

```
fn process_files_in_parallel(filenames: Vec<String>) -> io::Result<()> {
```

Notre nouvelle fonction a la même signature de type que l'original, ce qui en fait un remplacement pratique. `process_files`

```
    // Divide the work into several chunks.
    const NTHREADS: usize = 8;
    let worklists = split_vec_into_chunks(filenames, NTHREADS);
```

Nous utilisons une fonction utilitaire, non montrée ici, pour diviser le travail. Le résultat, , est un vecteur de vecteurs. Il contient huit portions de taille égale du vecteur d'origine. `split_vec_into_chunks worklists filenames`

```
    // Fork: Spawn a thread to handle each chunk.
    let mut thread_handles = vec![];
    for worklist in worklists {
        thread_handles.push(
            thread::spawn(move || process_files(worklist))
        );
    }
```

Nous créons un fil pour chaque fichier . renvoie une valeur appelée a , que nous utiliserons ultérieurement. Pour l'instant, nous mettons tous les s dans un vecteur. `worklist spawn() JoinHandle JoinHandle`

Notez comment nous obtenons la liste des noms de fichiers dans le thread de travail :

- `worklist` est défini et rempli par la boucle, dans le thread parent. `for`
- Dès que la fermeture est créée, est déplacé dans la fermeture. `move worklist`
- `spawn` puis déplace la fermeture (y compris le vecteur) sur le nouveau thread enfant. `worklist`

Ces déménagements sont bon marché. Comme les mouvements dont nous avons discuté au chapitre 4, les s ne sont pas clonés. En fait, rien n'est alloué ou libéré. La seule donnée déplacée est elle-même : trois mots machine. `Vec<String> String Vec`

La plupart des threads que vous créez ont besoin à la fois de code et de données pour commencer. Les fermetures antirouille contiennent facilement le code que vous voulez et les données que vous voulez.

Passons à autre chose :

```
        // Join: Wait for all threads to finish.
        for handle in thread_handles {
            handle.join().unwrap()?;
        }
```

Nous utilisons la méthode des s que nous avons collectés plus tôt pour attendre que les huit fils se terminent. Joindre des threads est souvent nécessaire pour l'exactitude, car un programme Rust se ferme dès qu'il revient, même si d'autres threads sont toujours en cours d'exécution. Les destructeurs ne sont pas appelés; les fils supplémentaires sont juste tués. Si ce n'est pas ce que vous voulez, assurez-vous de rejoindre tous les fils de discussion qui vous intéressent avant de revenir de `..join() JoinHandle main main`

Si nous parvenons à passer à travers cette boucle, cela signifie que les huit threads enfants se sont terminés avec succès. Notre fonction se termine donc par le retour : `Ok(())`

```
        Ok(())
    }
```

Gestion des erreurs entre les threads

Le code que nous avons utilisé pour joindre les threads enfants dans notre exemple est plus délicat qu'il n'y paraît, en raison de la gestion des erreurs. Revisitons cette ligne de code :

```
    handle.join().unwrap()?;
```

La méthode fait deux choses intéressantes pour nous. `.join()`

Tout d'abord, renvoie une erreur *si le thread enfant a paniqué.* Cela rend le threading dans Rust considérablement plus robuste qu'en C ++. En C++, un accès au tableau hors limites est un comportement indéfini et le reste du système ne protège pas les conséquences. Dans Rust, la panique est sûre et par fil. Les limites entre les threads servent de pare-feu pour la panique ; la panique ne se propage pas automatiquement d'un thread aux threads qui en dépendent. Au lieu de cela, une panique dans un

thread est signalée comme une erreur dans d'autres threads. Le programme dans son ensemble peut facilement
récupérer. `handle.join() std::thread::Result Result`

Dans notre programme, cependant, nous ne tentons pas de gérer la panique de manière sophistiquée. Au lieu de cela, nous utilisons immédiatement sur ce point, affirmant qu'il s'agit d'un résultat et non d'un résultat. Si un thread enfant *paniquait*, cette assertion échouerait, de sorte que le thread parent paniquerait également. Nous propageons explicitement la panique des threads enfants vers le thread
parent. `.unwrap() Result Ok Err`

Deuxièmement, transmet la valeur de retour du thread enfant au thread parent. La fermeture à laquelle nous sommes passés a un type de retour de , parce que c'est ce qui retourne. Cette valeur renvoyée n'est pas ignorée. Lorsque le thread enfant a terminé, sa valeur de retour est enregistrée et transfère cette valeur au thread
parent. `handle.join() spawn io::Result<()> process_files Join Handle::join()`

Le type complet renvoyé par dans ce programme est . Le fait partie de l'API / ; le fait partie de notre
application. `handle.join() std::thread::Result<std::io::Result <()>> thread::Result spawn join io::Result`

Dans notre cas, après avoir déballé le , nous utilisons l'opérateur sur le , propageant explicitement les erreurs d'E/S des threads enfants vers le thread parent. `thread::Result ? io::Result`

Tout cela peut sembler assez complexe. Mais considérez qu'il ne s'agit que d'une ligne de code, puis comparez-la avec d'autres langages. Le comportement par défaut en Java et C# est que les exceptions dans les threads enfants soient vidées vers le terminal, puis oubliées. En C++, la valeur par défaut est d'abandonner le processus. Dans Rust, les erreurs sont des valeurs (données) au lieu d'exceptions (flux de contrôle). Ils sont livrés sur des threads comme n'importe quelle autre valeur. Chaque fois que vous utilisez des API de threading de bas niveau, vous finissez par devoir écrire du code de gestion des erreurs avec soin, mais *étant donné que vous devez l'écrire, c'est* très agréable à avoir. `Result Result`

Partage de données immuables entre threads

Supposons que l'analyse que nous effectuons nécessite une grande base de données de mots et d'expressions anglais :

```
// before
fn process_files(filenames: Vec<String>)

// after
fn process_files(filenames: Vec<String>, glossary: &GigabyteMap)
```

Cela va être gros, alors nous le transmettons par référence. Comment pouvons-nous mettre à jour pour transmettre le glossaire aux threads de travail ? `glossary` `process_files_in_parallel`

Le changement évident ne fonctionne pas:

```
fn process_files_in_parallel(filenames: Vec<String>,
                             glossary: &GigabyteMap)
    -> io::Result<()>
{
    ...
    for worklist in worklists {
        thread_handles.push(
            spawn(move || process_files(worklist, glossary))  // error
        );
    }
    ...
}
```

Nous avons simplement ajouté un argument à notre fonction et l'avons transmis à . Rust se plaint: `glossary` `process_files`

```
error: explicit lifetime required in the type of `glossary`
   |
38 |             spawn(move || process_files(worklist, glossary))  // er
   |             ^^^^^ lifetime `'static` required
```

Rust se plaint de la durée de vie de la fermeture à laquelle nous passons, et le message « utile » que le compilateur présente ici n'est en fait d'aucune aide. `spawn`

`spawn` lance des threads indépendants. Rust n'a aucun moyen de savoir combien de temps le thread enfant s'exécutera, il suppose donc le pire : il suppose que le thread enfant peut continuer à fonctionner même après la fin du thread parent et que toutes les valeurs du thread parent ont disparu. De toute évidence, si le fil enfant doit durer aussi longtemps, la fermeture qu'il exécute doit durer aussi longtemps. Mais cette fermeture a une durée de vie limitée : elle dépend de la référence, et les références ne durent pas éternellement. `glossary`

Notez que Rust a raison de rejeter ce code! De la façon dont nous avons écrit cette fonction, il *est* possible qu'un thread rencontre une erreur d'E/S, ce qui provoque un renflouement avant que les autres threads ne soient terminés. Les fils de discussion enfants pourraient finir par essayer d'utiliser le glossaire une fois que le thread principal l'a libéré. Ce serait une course – avec un comportement indéfini comme prix, si le fil conducteur devait gagner. La rouille ne peut pas permettre cela. `process_files_in_parallel`

Il semble être trop ouvert pour prendre en charge le partage de références entre les threads. En effet, nous avons déjà vu un cas comme celui-ci, dans « Fermetures qui volent ». Là, notre solution consistait à transférer la propriété des données vers le nouveau thread, à l'aide d'une fermeture. Cela ne fonctionnera pas ici, car nous avons de nombreux threads qui doivent tous utiliser les mêmes données. Une alternative sûre est à l'ensemble du glossaire pour chaque fil, mais comme il est grand, nous voulons éviter cela. Heureusement, la bibliothèque standard offre un autre moyen : le comptage de référence atomique. `spawn move clone`

Nous l'avons décrit dans « Rc et Arc : Propriété partagée ». Il est temps de l'utiliser: `Arc`

```
use std::sync::Arc;

fn process_files_in_parallel(filenames: Vec<String>,
                             glossary: Arc<GigabyteMap>)
    -> io::Result<()>
{
    ...
    for worklist in worklists {
        // This call to .clone() only clones the Arc and bumps the
        // reference count. It does not clone the GigabyteMap.
        let glossary_for_child = glossary.clone();
        thread_handles.push(
            spawn(move || process_files(worklist, &glossary_for_child))
        );
    }
    ...
}
```

Nous avons changé le type de : pour exécuter l'analyse en parallèle, l'appelant doit passer dans un , un pointeur intelligent à un qui a été déplacé dans le tas, en utilisant
. `glossary Arc<GigabyteMap> GigabyteMap Arc::new(giga_map)`

Lorsque nous appelons , nous faisons une copie du pointeur intelligent, pas l'ensemble . Cela revient à incrémenter un nombre de références. `glossary.clone() Arc GigabyteMap`

Avec cette modification, le programme compile et s'exécute, car il ne dépend plus des durées de vie de référence. Tant *qu'un* thread possède un , il gardera la carte en vie, même si le thread parent se sauve plus tôt. Il n'y aura pas de course aux données, car les données dans un sont immuables. `Arc<GigabyteMap> Arc`

Rayonne

La fonction de la bibliothèque standard est une primitive importante, mais elle n'est pas conçue spécifiquement pour le parallélisme fourche-jointure. De meilleures API de jointure de fourche ont été construites dessus. Par exemple, dans le chapitre 2, nous avons utilisé la bibliothèque Crossbeam pour diviser certains travaux sur huit threads. Les filetages à lunette de Crossbeam prennent en charge le *parallélisme* fourche-jointure tout naturellement. `spawn`

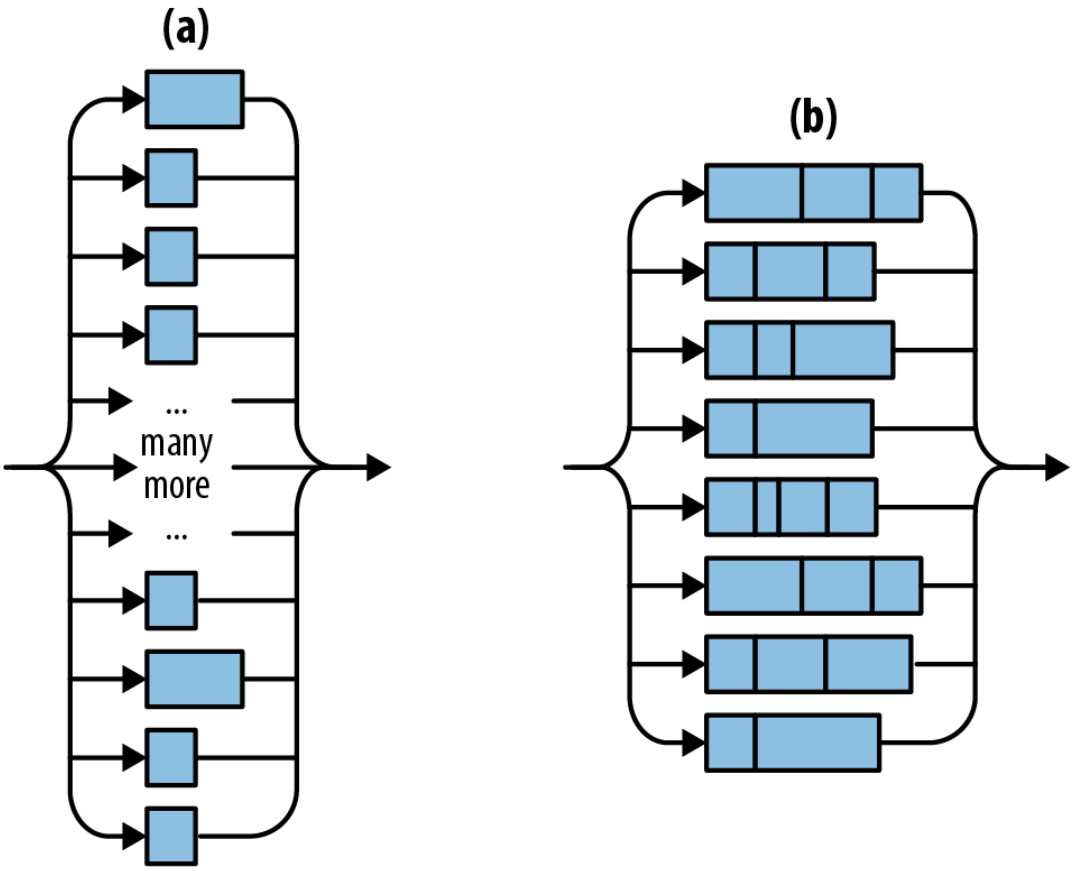
La bibliothèque Rayon, de Niko Matsakis et Josh Stone, en est un autre exemple. Il offre deux façons d'exécuter des tâches simultanément :

```
use rayon::prelude::*;

// "do 2 things in parallel"
let (v1, v2) = rayon::join(fn1, fn2);

// "do N things in parallel"
giant_vector.par_iter().for_each(|value| {
    do_thing_with_value(value);
});
```

`rayon::join(fn1, fn2)` appelle simplement les deux fonctions et renvoie les deux résultats. La méthode crée un , une valeur avec , , et d'autres méthodes, un peu comme un Rust . Dans les deux cas, Rayon utilise son propre pool de threads de travail pour répartir le travail lorsque cela est possible. Vous dites simplement à Rayon quelles tâches *peuvent* être effectuées en parallèle; Rayon gère les fils et distribue le travail du mieux qu'il peut. `.par_iter() ParallelIterator map filter Iterator`

Les diagrammes de la figure 19-3 illustrent deux façons de penser l'appel. (a) La rayonne agit comme si elle produisait un fil par élément dans le vecteur. (b) Dans les coulisses, Rayon dispose d'un thread de travail par

cœur de processeur, ce qui est plus efficace. Ce pool de threads de travail est partagé par tous les threads de votre programme. Lorsque des milliers de tâches arrivent à la fois, Rayon divise le travail. `giant_vector.par_iter().for_each(...)`



Graphique 19-3. Rayon en théorie et en pratique

Voici une version de l'utilisation de Rayon et une qui prend, plutôt que , juste un
: `process_files_in_parallel process_file Vec<String> &str`

```
use rayon::prelude::*;

fn process_files_in_parallel(filenames: Vec<String>, glossary: &Gigabyte
    -> io::Result<()>
{
    filenames.par_iter()
        .map(|filename| process_file(filename, glossary))
        .reduce_with(|r1, r2| {
            if r1.is_err() { r1 } else { r2 }
        })
        .unwrap_or(Ok(()))
}
```

Ce code est plus court et moins délicat que la version utilisant . Regardons-le ligne par ligne: `std::thread::spawn`

- Tout d'abord, nous utilisons pour créer un itérateur parallèle. `filenames.par_iter()`
- Nous avons l'habitude d'appeler chaque nom de fichier. Cela produit une séquence de valeurs. `.map() process_file ParallelIterator io::Result<() >`
- Nous utilisons pour combiner les résultats. Ici, nous conservons la première erreur, le cas échéant, et jetons le reste. Si nous voulions accumuler toutes les erreurs, ou les imprimer, nous pourrions le faire ici. `.reduce_with()`
  La méthode est également pratique lorsque vous passez une fermeture qui renvoie une valeur utile sur le succès. Ensuite, vous pouvez passer une clôture qui sait comment combiner deux résultats de succès. `.reduce_with() .map() .reduce_with()`
- `reduce_with` renvoie un qui n'est que si était vide. Nous utilisons la méthode de 's pour obtenir le résultat dans ce cas. `Option None filenames Option .unwrap_or() Ok(())`

Dans les coulisses, Rayon équilibre dynamiquement les charges de travail entre les threads, à l'aide d'une technique appelée *vol de travail.* Il fera généralement un meilleur travail en gardant tous les processeurs occupés que nous ne pouvons le faire en divisant manuellement le travail à l'avance, comme dans « spawn and join ».

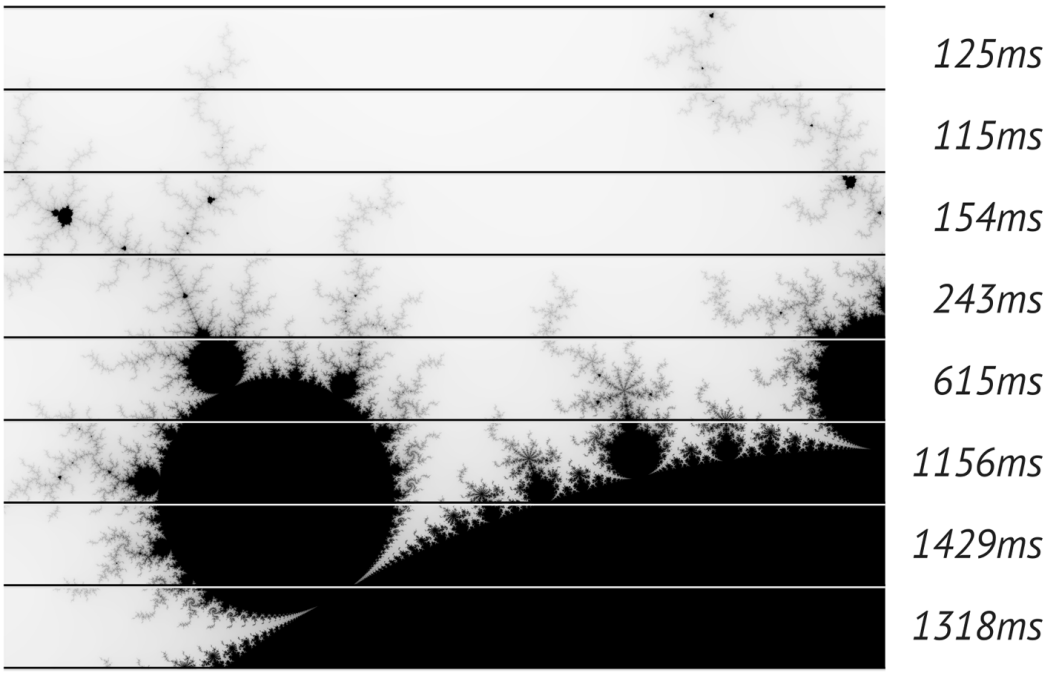
En prime, Rayon prend en charge le partage de références entre les threads. Tout traitement parallèle qui se produit dans les coulisses est garanti d'être terminé au moment du retour. Cela explique pourquoi nous avons pu passer à même si cette fermeture sera appelée sur plusieurs threads. `reduce_with glossary process_file`

(Incidemment, ce n'est pas un hasard si nous avons utilisé une méthode et une méthode. Le modèle de programmation MapReduce, popularisé par Google et Apache Hadoop, a beaucoup en commun avec le fork-join. Cela peut être considéré comme une approche de jointure de fourche pour interroger des données distribuées.) `map reduce`

Revisiter l'ensemble Mandelbrot

Au chapitre 2, nous avons utilisé la simultanéité de jointure de fourche pour rendre l'ensemble de Mandelbrot. Cela a rendu le rendu quatre fois plus rapide - impressionnant, mais pas aussi impressionnant qu'il pourrait l'être, étant donné que nous avons fait en sorte que le programme génère huit threads de travail et l'exécute sur une machine à huit cœurs!

Le problème est que nous n'avons pas réparti la charge de travail uniformément. Calculer un pixel de l'image revient à exécuter une boucle (voir « Qu'est réellement l'ensemble de Mandelbrot »). Il s'avère que les parties gris pâle de l'image, où la boucle se ferme rapidement, sont beaucoup plus rapides à rendre que les parties noires, où la boucle exécute les 255 itérations complètes. Ainsi, bien que nous divisions la zone en bandes horizontales de taille égale, nous créons des charges de travail inégales, comme le montre la figure 19-4.



Graphique 19-4. Répartition inégale du travail dans le programme Mandelbrot

Ceci est facile à réparer à l'aide de Rayon. Nous pouvons simplement lancer une tâche parallèle pour chaque ligne de pixels dans la sortie. Cela crée plusieurs centaines de tâches que Rayon peut répartir sur ses threads. Grâce au vol de travail, peu importe que les tâches varient en taille. Rayon équilibrera le travail au fur et à mesure.

Voici le code. La première ligne et la dernière ligne font partie de la fonction que nous avons montrée dans « A Concurrent Mandelbrot Program », mais nous avons changé le code de rendu, qui est tout ce qui se trouve entre les deux: `main`

```
let mut pixels = vec![0; bounds.0 * bounds.1];

// Scope of slicing up `pixels` into horizontal bands.
{
    let bands: Vec<(usize, &mut [u8])> = pixels
        .chunks_mut(bounds.0)
        .enumerate()
        .collect();
```

```
        bands.into_par_iter()
            .for_each(|(i, band)| {
                let top = i;
                let band_bounds = (bounds.0, 1);
                let band_upper_left = pixel_to_point(bounds, (0, top),
                                                     upper_left, lower_right
                let band_lower_right = pixel_to_point(bounds, (bounds.0, top
                                                      upper_left, lower_righ
                render(band, band_bounds, band_upper_left, band_lower_right)
            });
    }

    write_image(&args[1], &pixels, bounds).expect("error writing PNG file");
```

Tout d'abord, nous créons , la collection de tâches que nous allons transmettre à Rayon. Chaque tâche n'est qu'un tuple de type : le numéro de ligne, puisque le calcul l'exige, et la tranche de à remplir. Nous utilisons la méthode pour diviser le tampon d'image en lignes, pour attacher un numéro de ligne à chaque ligne et pour convertir toutes les paires nombre-tranche en un vecteur. (Nous avons besoin d'un vecteur car Rayon crée des itérateurs parallèles uniquement à partir de tableaux et de vecteurs.) `bands (usize, &mut [u8]) pixels chunks_mut enumerate collect`

Ensuite, nous nous transformons en un itérateur parallèle et utilisons la méthode pour dire à Rayon quel travail nous voulons faire. `bands .for_each()`

Puisque nous utilisons Rayon, nous devons ajouter cette ligne à *main.rs* :

```
use rayon::prelude::*;
```

et ceci à *Cargo.toml*:

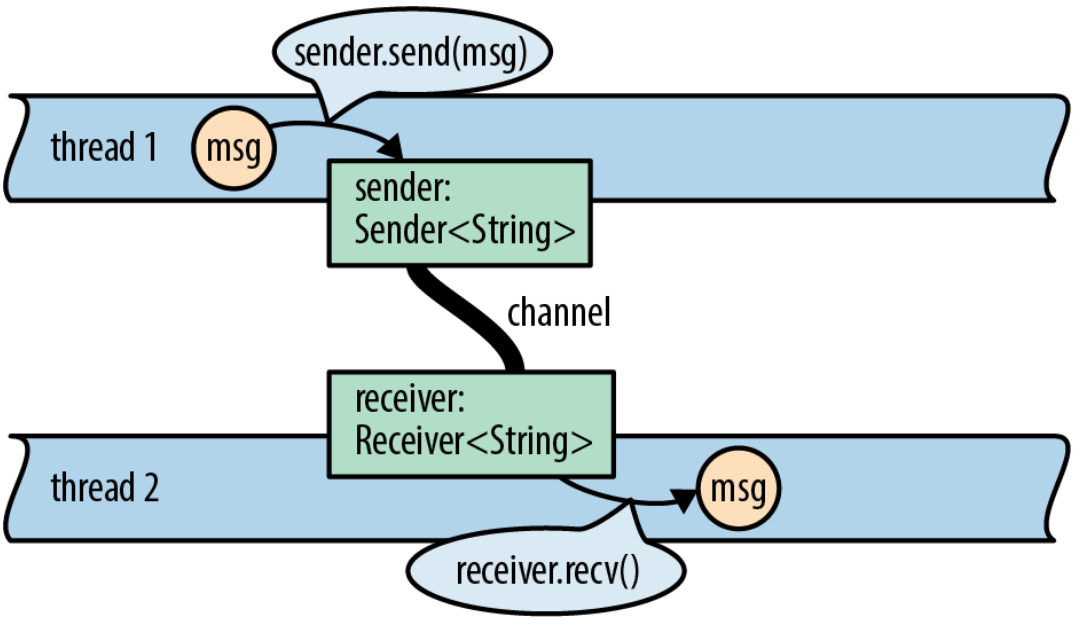
```
[dependencies]
rayon = "1"
```

Avec ces changements, le programme utilise maintenant environ 7,75 cœurs sur une machine à 8 cœurs. C'est 75% plus rapide qu'avant, lorsque nous divisions le travail manuellement. Et le code est un peu plus court, reflétant les avantages de laisser une caisse faire un travail (répartition du travail) plutôt que de le faire nous-mêmes.

Canaux

Un *canal* est un canal unidirectionnel permettant d'envoyer des valeurs d'un thread à un autre. En d'autres termes, il s'agit d'une file d'attente sécurisée.

La figure 19-5 illustre la façon dont les canaux sont utilisés. Ils sont quelque chose comme des tuyaux Unix: une extrémité est pour envoyer des données, et l'autre est pour recevoir. Les deux extrémités appartiennent généralement à deux threads différents. Mais alors que les tuyaux Unix sont destinés à l'envoi d'octets, les canaux sont destinés à l'envoi de valeurs Rust. met une seule valeur dans le canal ; en supprime un. La propriété est transférée du thread d'envoi au thread de réception. Si le canal est vide, bloque jusqu'à ce qu'une valeur soit envoyée. `sender.send(item) receiver.recv() receiver.recv()`



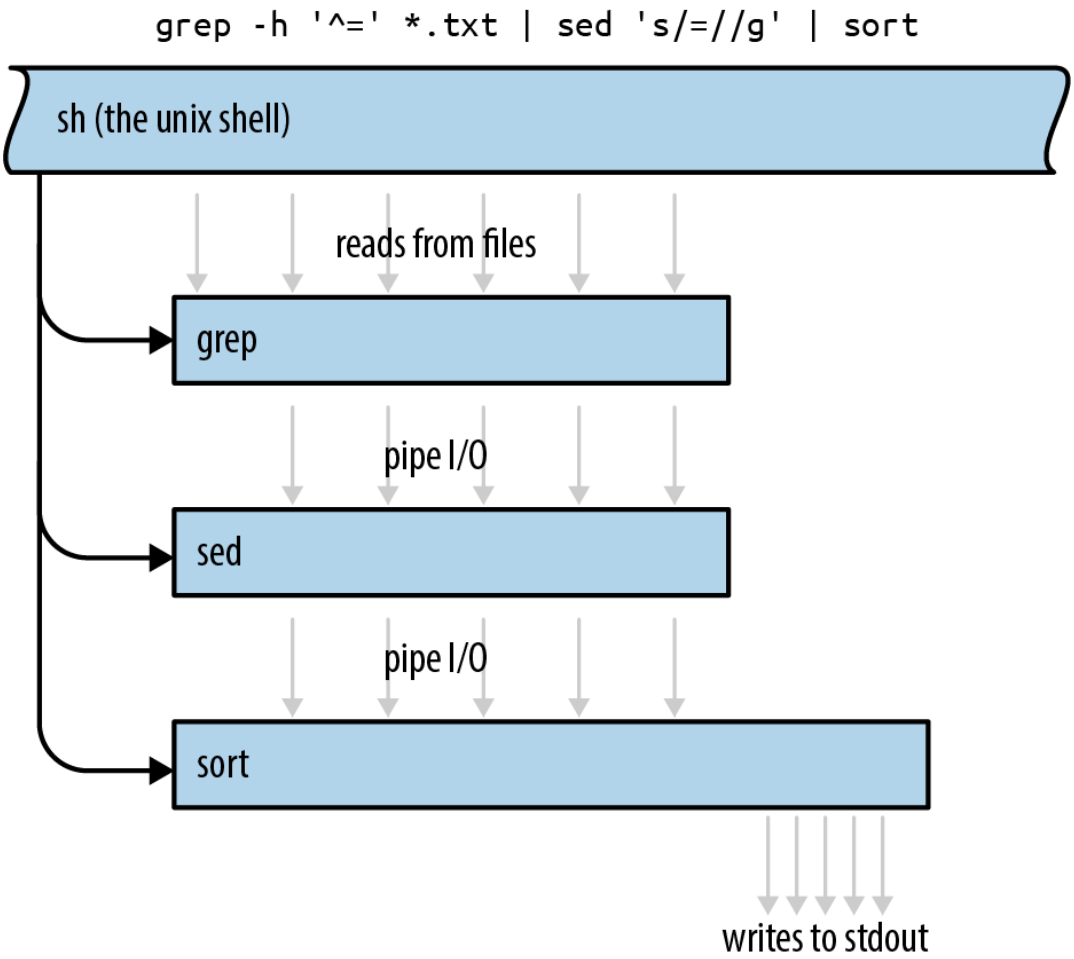
Graphique 19-5. Un canal pour s: la propriété de la chaîne msg est transférée du thread 1 au thread 2. `String`

Avec les canaux, les threads peuvent communiquer en se transmettant des valeurs. C'est un moyen très simple pour les threads de travailler ensemble sans utiliser de verrouillage ou de mémoire partagée.

Ce n'est pas une technique nouvelle. Erlang a des processus isolés et des messages qui passent depuis 30 ans maintenant. Les tuyaux Unix existent depuis près de 50 ans. Nous avons tendance à penser que les tuyaux offrent de la flexibilité et de la composabilité, pas de la concurrence, mais en fait, ils font tout ce qui précède. Un exemple de pipeline Unix est illustré à la figure 19-6. Il est certainement possible que les trois programmes fonctionnent en même temps.

Les canaux de rouille sont plus rapides que les tuyaux Unix. L'envoi d'une valeur la déplace plutôt que de la copier, et les déplacements sont rapi-

des, même lorsque vous déplacez des structures de données contenant plusieurs mégaoctets de données.



Graphique 19-6. Exécution d'un pipeline Unix

Envoi de valeurs

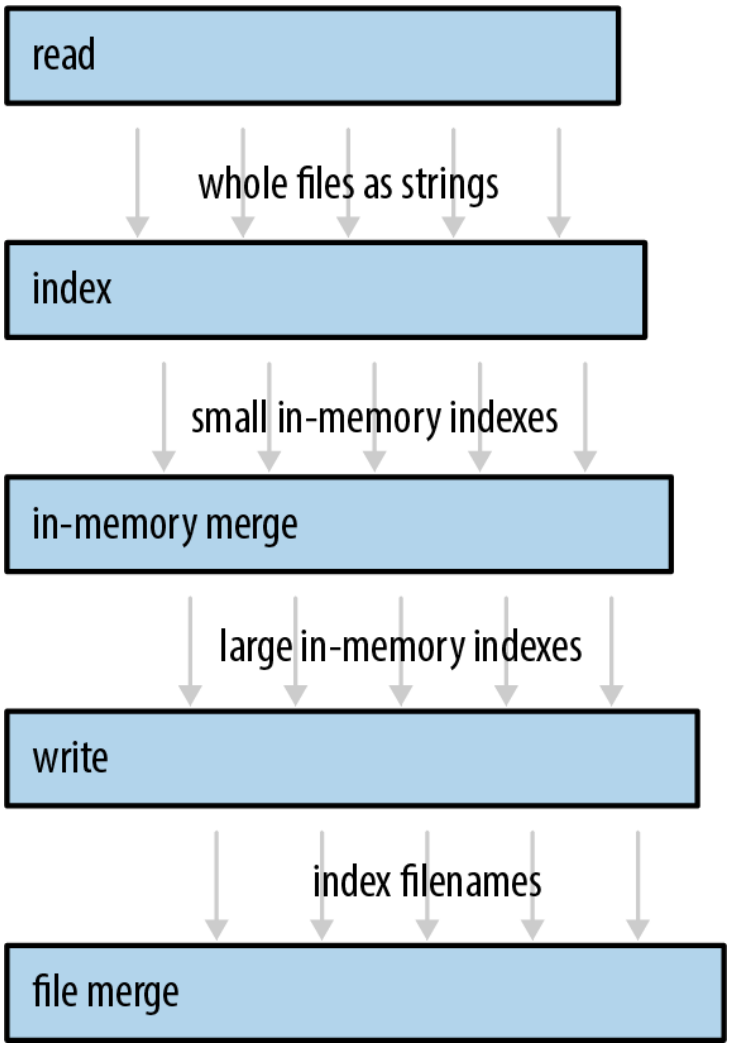
Au cours des prochaines sections, nous utiliserons des canaux pour créer un programme simultané qui crée un *index inversé*, l'un des ingrédients clés d'un moteur de recherche. Chaque moteur de recherche fonctionne sur une collection particulière de documents. L'index inversé est la base de données qui indique quels mots apparaissent où.

Nous allons montrer les parties du code qui ont à voir avec les threads et les canaux. Le programme complet est court, environ un millier de lignes de code en tout.

Notre programme est structuré comme un pipeline, comme le montre la figure 19-7. Les pipelines ne sont qu'une des nombreuses façons d'utiliser les canaux (nous discuterons de quelques autres utilisations plus tard), mais ils constituent un moyen simple d'introduire la concurrence d'accès concurrentiel dans un programme monothread existant.

Nous utiliserons un total de cinq threads, chacun effectuant une tâche distincte. Chaque thread produit une sortie en continu pendant toute la durée de vie du programme. Le premier thread, par exemple, lit simple-

ment les documents sources du disque en mémoire, un par un. (Nous voulons un thread pour le faire parce que nous allons écrire le code le plus simple possible ici, en utilisant , qui est une API de blocage. Nous ne voulons pas que le processeur reste inactif chaque fois que le disque fonctionne.) La sortie de cette étape est longue d'un par document, de sorte que ce thread est connecté au thread suivant par un canal de s. `fs::read_to_string` `String` `String`



Graphique 19-7. Le pipeline du générateur d'index, où les flèches représentent les valeurs envoyées via un canal d'un thread à un autre (les E/S disque ne sont pas affichées)

Notre programme commencera par générer le thread qui lit les fichiers. Supposons est un , un vecteur de noms de fichiers. Le code pour démarrer notre thread de lecture de fichiers ressemble à ceci : `documents` `Vec<PathBuf>`

```
use std::{fs, thread};
use std::sync::mpsc;

let (sender, receiver) = mpsc::channel();

let handle = thread::spawn(move || {
    for filename in documents {
        let text = fs::read_to_string(filename)?;
```

```
            if sender.send(text).is_err() {
                break;
            }
        }
        Ok(())
    });
```

Les canaux font partie du module. Nous expliquerons ce que ce nom signifie plus tard; Tout d'abord, regardons comment ce code fonctionne. Nous commençons par créer un canal : `std::sync::mpsc`

```
    let (sender, receiver) = mpsc::channel();
```

La fonction renvoie une paire de valeurs : un expéditeur et un récepteur. La structure de données de file d'attente sous-jacente est un détail d'implémentation que la bibliothèque standard n'expose pas. `channel`

Les canaux sont tapés. Nous allons utiliser ce canal pour envoyer le texte de chaque fichier, nous avons donc un de type et un de type . Nous aurions pu demander explicitement un canal de chaînes, en écrivant . Au lieu de cela, nous laissons l'inférence de type Rust le comprendre. `sender Sender<String> receiver Receiver<String> mpsc::channel::<String>()`

```
    let handle = thread::spawn(move || {
```

Comme précédemment, nous utilisons pour démarrer un thread. La propriété (mais non) est transférée au nouveau thread via cette fermeture. `std::thread::spawn sender receiver move`

Les quelques lignes de code suivantes lisent simplement les fichiers du disque:

```
        for filename in documents {
            let text = fs::read_to_string(filename)?;
```

Après avoir lu avec succès un fichier, nous envoyons son texte dans le canal:

```
            if sender.send(text).is_err() {
                break;
            }
        }
```

`sender.send(text)` déplace la valeur dans le canal. En fin de compte, il sera à nouveau déplacé vers celui qui recevra la valeur. Qu'elle contienne 10 lignes de texte ou 10 mégaoctets, cette opération copie trois mots machine (la taille d'une struct), et l'appel correspondant copiera également trois mots machine. `text text String receiver.recv()`

Les méthodes et renvoient toutes deux des s, mais ces méthodes échouent uniquement si l'autre extrémité du canal a été supprimée. Un appel échoue si le a été supprimé, car sinon la valeur resterait dans le canal pour toujours: sans un , il n'y a aucun moyen pour un thread de le recevoir. De même, un appel échoue s'il n'y a pas de valeurs en attente dans le canal et que le a été supprimé, sinon attendrait éternellement: sans un , il n'y a aucun moyen pour un thread d'envoyer la valeur suivante. Laisser tomber votre extrémité d'un canal est la façon normale de « raccrocher », de fermer la connexion lorsque vous avez terminé. `send recv Result send Receiver Receiver recv Sender recv Sender`

Dans notre code, n'échouera que si le thread du récepteur s'est arrêté prématurément. Ceci est typique pour le code qui utilise des canaux. Que cela se soit produit délibérément ou en raison d'une erreur, il est normal que notre fil de discussion pour le lecteur s'éteigne tranquillement. `sender.send(text)`

Lorsque cela se produit, ou que le fil de discussion termine la lecture de tous les documents, il renvoie : `Ok(())`

```
        Ok(())
    });
```

Notez que cette fermeture renvoie un fichier . Si le thread rencontre une erreur d'E/S, il se ferme immédiatement et l'erreur est stockée dans le fichier . `Result JoinHandle`

Bien sûr, comme tout autre langage de programmation, Rust admet de nombreuses autres possibilités en matière de gestion des erreurs. Lorsqu'une erreur se produit, nous pouvons simplement l'imprimer en utilisant et passer au fichier suivant. Nous pourrions transmettre les erreurs via le même canal que celui que nous utilisons pour les données, ce qui en fait un canal de s, ou créer un deuxième canal uniquement pour les erreurs. L'approche que nous avons choisie ici est à la fois légère et responsable : nous utilisons l'opérateur, il n'y a donc pas un tas de code standard, ni même un explicite comme vous pourriez le voir en Java, et

pourtant les erreurs ne passeront pas silencieusement. `println! Result ? try/catch`

Pour plus de commodité, notre programme enveloppe tout ce code dans une fonction qui renvoie à la fois le (que nous n'avons pas encore utilisé) et le nouveau thread : `receiver JoinHandle`

```
fn start_file_reader_thread(documents: Vec<PathBuf>)
    -> (mpsc::Receiver<String>, thread::JoinHandle<io::Result<()>>)
{
    let (sender, receiver) = mpsc::channel();

    let handle = thread::spawn(move || {
        ...
    });

    (receiver, handle)
}
```

Notez que cette fonction lance le nouveau thread et le renvoie immédiatement. Nous allons écrire une fonction comme celle-ci pour chaque étape de notre pipeline.

Réception des valeurs

Maintenant, nous avons un thread exécutant une boucle qui envoie des valeurs. On peut engendrer un second thread exécutant une boucle qui appelle : `receiver.recv()`

```
while let Ok(text) = receiver.recv() {
    do_something_with(text);
}
```

Mais s sont itérables, il y a donc une meilleure façon d'écrire ceci: `Receiver`

```
for text in receiver {
    do_something_with(text);
}
```

Ces deux boucles sont équivalentes. Quoi qu'il en soit, nous l'écrivons, si le canal se trouve vide lorsque le contrôle atteint le haut de la boucle, le thread récepteur se bloquera jusqu'à ce qu'un autre thread envoie une valeur. La boucle se ferme normalement lorsque le canal est vide et qu'il a été abandonné. Dans notre programme, cela se produit naturellement

lorsque le fil du lecteur se ferme. Ce thread exécute une fermeture qui possède la variable ; lorsque la fermeture sort, est abandonné. `Sender sender sender`

Nous pouvons maintenant écrire du code pour la deuxième étape du pipeline :

```
fn start_file_indexing_thread(texts: mpsc::Receiver<String>)
    -> (mpsc::Receiver<InMemoryIndex>, thread::JoinHandle<()>)
{
    let (sender, receiver) = mpsc::channel();

    let handle = thread::spawn(move || {
        for (doc_id, text) in texts.into_iter().enumerate() {
            let index = InMemoryIndex::from_single_document(doc_id, text
            if sender.send(index).is_err() {
                break;
            }
        }
    });

    (receiver, handle)
}
```

Cette fonction génère un thread qui reçoit des valeurs d'un canal () et envoie des valeurs à un autre canal (/). Le travail de ce thread consiste à prendre chacun des fichiers chargés dans la première étape et à transformer chaque document en un petit index inversé en mémoire. `String texts InMemoryIndex sender receiver`

La boucle principale de ce thread est simple. Tout le travail d'indexation d'un document se fait par la fonction . Nous n'afficherons pas son code source ici, mais il divise la chaîne d'entrée aux limites des mots, puis produit une carte des mots aux listes de positions. `InMemoryIndex:: from_single_document`

Cette étape n'effectue pas d'E/S, elle n'a donc pas à traiter avec s. Au lieu d'un , il renvoie . `io::Error io::Result<()> ()`

Exécution du pipeline

Les trois étapes restantes sont de conception similaire. Chacun consomme un créé par l'étape précédente. Notre objectif pour le reste du pipeline est de fusionner tous les petits index en un seul grand fichier d'index sur disque. Le moyen le plus rapide que nous avons trouvé pour le faire est

en trois étapes. Nous n'afficherons pas le code ici, juste les signatures de type de ces trois fonctions. La source complète est en ligne. `Receiver`

Tout d'abord, nous fusionnons les index en mémoire jusqu'à ce qu'ils deviennent encombrants (étape 3) :

```
fn start_in_memory_merge_thread(file_indexes: mpsc::Receiver<InMemoryInd
    -> (mpsc::Receiver<InMemoryIndex>, thread::JoinHandle<()>)
```

Nous écrivons ces grands index sur disque (étape 4) :

```
fn start_index_writer_thread(big_indexes: mpsc::Receiver<InMemoryIndex>,
                             output_dir: &Path)
    -> (mpsc::Receiver<PathBuf>, thread::JoinHandle<io::Result<()>>)
```

Enfin, si nous avons plusieurs fichiers volumineux, nous les fusionnons à l'aide d'un algorithme de fusion basé sur des fichiers (étape 5):

```
fn merge_index_files(files: mpsc::Receiver<PathBuf>, output_dir: &Path)
    -> io::Result<()>
```

Cette dernière étape ne renvoie pas un , car c'est la fin de la ligne. Il produit un seul fichier de sortie sur le disque. Il ne renvoie pas un , car nous ne prenons pas la peine de générer un fil pour cette étape. Le travail se fait sur le fil de l'appelant. `Receiver` `JoinHandle`

Nous arrivons maintenant au code qui lance les threads et vérifie les erreurs:

```
fn run_pipeline(documents: Vec<PathBuf>, output_dir: PathBuf)
    -> io::Result<()>
{
    // Launch all five stages of the pipeline.
    let (texts,   h1) = start_file_reader_thread(documents);
    let (pints,   h2) = start_file_indexing_thread(texts);
    let (gallons, h3) = start_in_memory_merge_thread(pints);
    let (files,   h4) = start_index_writer_thread(gallons, &output_dir);
    let result = merge_index_files(files, &output_dir);

    // Wait for threads to finish, holding on to any errors that they en
    let r1 = h1.join().unwrap();
    h2.join().unwrap();
    h3.join().unwrap();
    let r4 = h4.join().unwrap();

    // Return the first error encountered, if any.
```

```
        // (As it happens, h2 and h3 can't fail: those threads
        // are pure in-memory data processing.)
        r1?;
        r4?;
        result
    }
```

Comme précédemment, nous avons l'habitude de propager explicitement les paniques des threads enfants vers le thread principal. La seule autre chose inhabituelle ici est qu'au lieu d'utiliser tout de suite, nous mettons de côté les valeurs jusqu'à ce que nous ayons rejoint les quatre threads. `.join().unwrap() ? io::Result`

Ce pipeline est 40 % plus rapide que l'équivalent monothread. Ce n'est pas mal pour un après-midi de travail, mais dérisoire à côté du coup de pouce de 675% que nous avons obtenu pour le programme Mandelbrot. Nous n'avons clairement pas saturé la capacité d'E/S du système ni tous les cœurs de processeur. Que se passe-t-il?

Les pipelines sont comme des lignes d'assemblage dans une usine de fabrication : les performances sont limitées par le débit de l'étape la plus lente. Une toute nouvelle chaîne de montage non réglée peut être aussi lente que la production unitaire, mais les chaînes de montage récompensent le réglage ciblé. Dans notre cas, la mesure montre que la deuxième étape est le goulot d'étranglement. Notre thread d'indexation utilise et , il passe donc beaucoup de temps à fouiller dans les tables Unicode. Les autres étapes en aval de l'indexation passent la plupart de leur temps à dormir dans , en attendant l'entrée. `.to_lowercase() .is_alphanumeric() Receiver::recv`

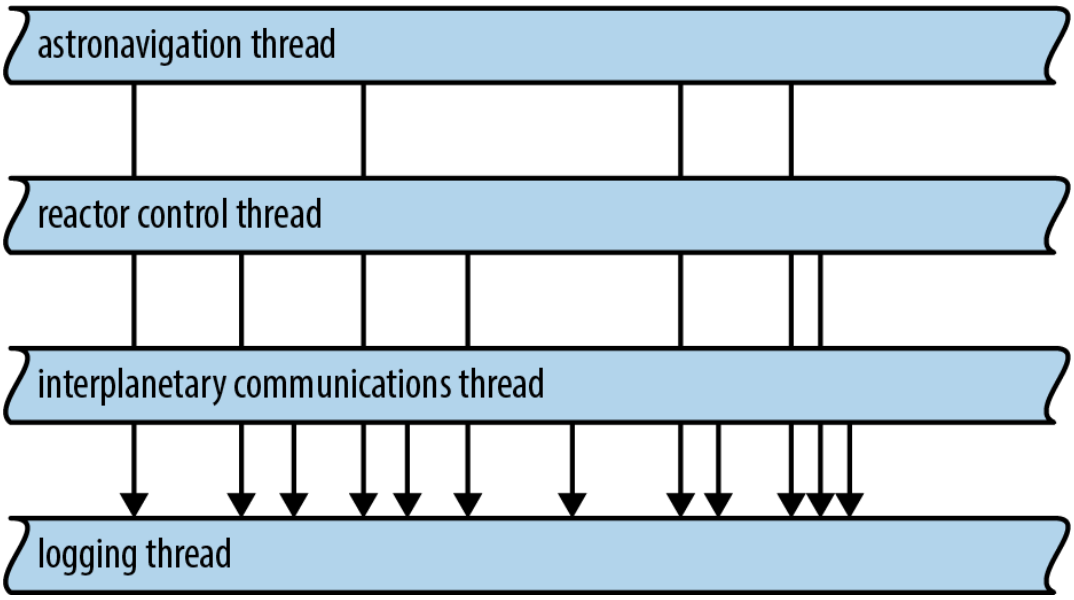
Cela signifie que nous devrions être en mesure d'aller plus vite. Au fur et à mesure que nous nous attaquerons aux goulets d'étranglement, le degré de parallélisme augmentera. Maintenant que vous savez comment utiliser les canaux et que notre programme est composé de morceaux de code isolés, il est facile de voir des moyens de résoudre ce premier goulot d'étranglement. Nous pourrions optimiser manuellement le code pour la deuxième étape, comme n'importe quel autre code; diviser l'œuvre en deux étapes ou plus; ou exécutez plusieurs threads d'indexation de fichiers à la fois.

Caractéristiques et performances du canal

La partie de signifie *multiproducteur, mono-consommateur,* une description laconique du type de communication fourni par les canaux de

Rust. `mpsc std::sync::mpsc`

Les canaux de notre exemple de programme transportent des valeurs d'un seul expéditeur à un seul récepteur. C'est un cas assez courant. Mais les canaux Rust prennent également en charge plusieurs expéditeurs, au cas où vous auriez besoin, par exemple, d'un seul thread qui gère les demandes de nombreux threads clients, comme illustré à la figure 19-8.



Graphique 19-8. Un canal unique recevant les demandes de nombreux expéditeurs

`Sender<T>` implémente le trait. Pour obtenir un canal avec plusieurs expéditeurs, créez simplement un canal normal et clonez l'expéditeur autant de fois que vous le souhaitez. Vous pouvez déplacer chaque valeur vers un thread différent. `Clone Sender`

A ne peut pas être cloné, donc si vous avez besoin de plusieurs threads recevant des valeurs du même canal, vous avez besoin d'un fichier . Nous montrerons comment le faire plus loin dans ce chapitre. `Receiver<T> Mutex`

Les canaux de rouille sont soigneusement optimisés. Lorsqu'un canal est créé pour la première fois, Rust utilise une implémentation de file d'attente spéciale « one-shot ». Si vous n'envoyez qu'un seul objet via le canal, la surcharge est minime. Si vous envoyez une deuxième valeur, Rust bascule vers une autre implémentation de file d'attente. Il s'installe à long terme, vraiment, préparant le canal à transférer de nombreuses valeurs tout en minimisant les frais généraux d'allocation. Et si vous clonez le , Rust doit se rabattre sur une autre implémentation, qui est sûre lorsque plusieurs threads essaient d'envoyer des valeurs à la fois. Mais même la plus lente de ces trois implémentations est une file d'attente sans verrouillage, de sorte que l'envoi ou la réception d'une valeur est tout au plus quelques opérations atomiques et une allocation de tas, plus le dé-

placement lui-même. Les appels système ne sont nécessaires que lorsque la file d'attente est vide et que le thread récepteur doit donc se mettre en veille. Dans ce cas, bien sûr, le trafic via votre canal n'est pas maximisé de toute façon. `Sender`

Malgré tout ce travail d'optimisation, il y a une erreur que les applications peuvent facilement commettre en ce qui concerne les performances des canaux : envoyer des valeurs plus rapidement qu'elles ne peuvent être reçues et traitées. Cela entraîne un arriéré de valeurs sans cesse croissant à accumuler dans le canal. Par exemple, dans notre programme, nous avons constaté que le thread du lecteur de fichiers (étape 1) pouvait charger les fichiers beaucoup plus rapidement que le thread d'indexation de fichiers (étape 2) pouvait les indexer. Le résultat est que des centaines de mégaoctets de données brutes seraient lus à partir du disque et stockés dans la file d'attente à la fois.

Ce genre de mauvaise conduite coûte de la mémoire et nuit à la localité. Pire encore, le thread d'envoi continue de fonctionner, utilisant le processeur et d'autres ressources système pour envoyer toujours plus de valeurs au moment où ces ressources sont les plus nécessaires à la réception.

Ici, Rust reprend une page des tuyaux Unix. Unix utilise une astuce élégante pour fournir une *certaine contre-pression* afin que les expéditeurs rapides soient obligés de ralentir: chaque tuyau d'un système Unix a une taille fixe, et si un processus essaie d'écrire sur un tuyau qui est momentanément plein, le système bloque simplement ce processus jusqu'à ce qu'il y ait de la place dans le tuyau. L'équivalent Rust est appelé *canal synchrone* :

```
use std::sync::mpsc;

let (sender, receiver) = mpsc::sync_channel(1000);
```

Un canal synchrone est exactement comme un canal normal, sauf que lorsque vous le créez, vous spécifiez le nombre de valeurs qu'il peut contenir. Pour un canal synchrone, il s'agit potentiellement d'une opération de blocage. Après tout, l'idée est que le blocage n'est pas toujours mauvais. Dans notre exemple de programme, le remplacement de l'entrée par un avec de la place pour 32 valeurs réduit l'utilisation de la mémoire des deux tiers sur notre ensemble de données de référence, sans diminuer le débit. `sender.send(value) channel start_file_reader_thread sync_channel`

Sécurité des threads : envoi et synchronisation

Jusqu'à présent, nous avons agi comme si toutes les valeurs pouvaient être librement déplacées et partagées à travers les fils. C'est en grande partie vrai, mais l'histoire complète de la sécurité du fil de Rust repose sur deux traits intégrés, et . `std::marker::Send std::marker::Sync`

- Les types qui implémentent peuvent être transmis en toute sécurité par valeur à un autre thread. Ils peuvent être déplacés d'un thread à l'autre. `Send`
- Les types qui implémentent peuvent passer en toute sécurité par non-référence à un autre thread. Ils peuvent être partagés entre les threads. `Sync mut`

Par *sûr* ici, nous entendons la même chose que nous voulons toujours dire: exempt de courses de données et d'autres comportements indéfinis.

Par exemple, dans l'exemple , nous avons utilisé une fermeture pour passer un du thread parent à chaque thread enfant. Nous ne l'avons pas signalé à l'époque, mais cela signifie que le vecteur et ses chaînes sont alloués dans le thread parent, mais libérés dans le thread enfant. Le fait que l'implémentation soit une promesse d'API que c'est OK: l'allocateur utilisé en interne par et est thread-safe. `process_files_in_parallel Vec<String> Vec<String> Send Vec String`

(Si vous deviez écrire vos propres types avec des allocateurs rapides mais non thread-safe, vous devrez les implémenter en utilisant des types qui ne le sont pas, tels que des pointeurs dangereux. Rust en déduirait alors que votre et les types ne le sont pas et les limiterait à une utilisation à filetage unique. Mais c'est un cas rare.) `Vec String Send NonThreadSafeVec NonThreadSafeString Send`

Comme l'illustre la figure 19-9, la plupart des types sont à la fois et . Vous n'avez même pas besoin d'utiliser pour obtenir ces traits sur les structs et les enums dans votre programme. Rust le fait pour vous. Une struct ou enum est si ses champs sont , et si ses champs sont . `Send Sync #[derive] Send Send Sync Sync`

Certains types sont , mais pas . Ceci est généralement intentionnel, comme dans le cas de , où il garantit que l'extrémité réceptrice d'un canal est utilisée par un seul thread à la fois. `Send Sync mpsc::Receiver mpsc`

Les quelques types qui ne sont ni ne sont principalement ceux qui utilisent la mutabilité d'une manière qui n'est pas thread-safe. Par exemple, considérez , le type de pointeurs intelligents de comptage de références. `Send Sync std::rc::Rc<T>`

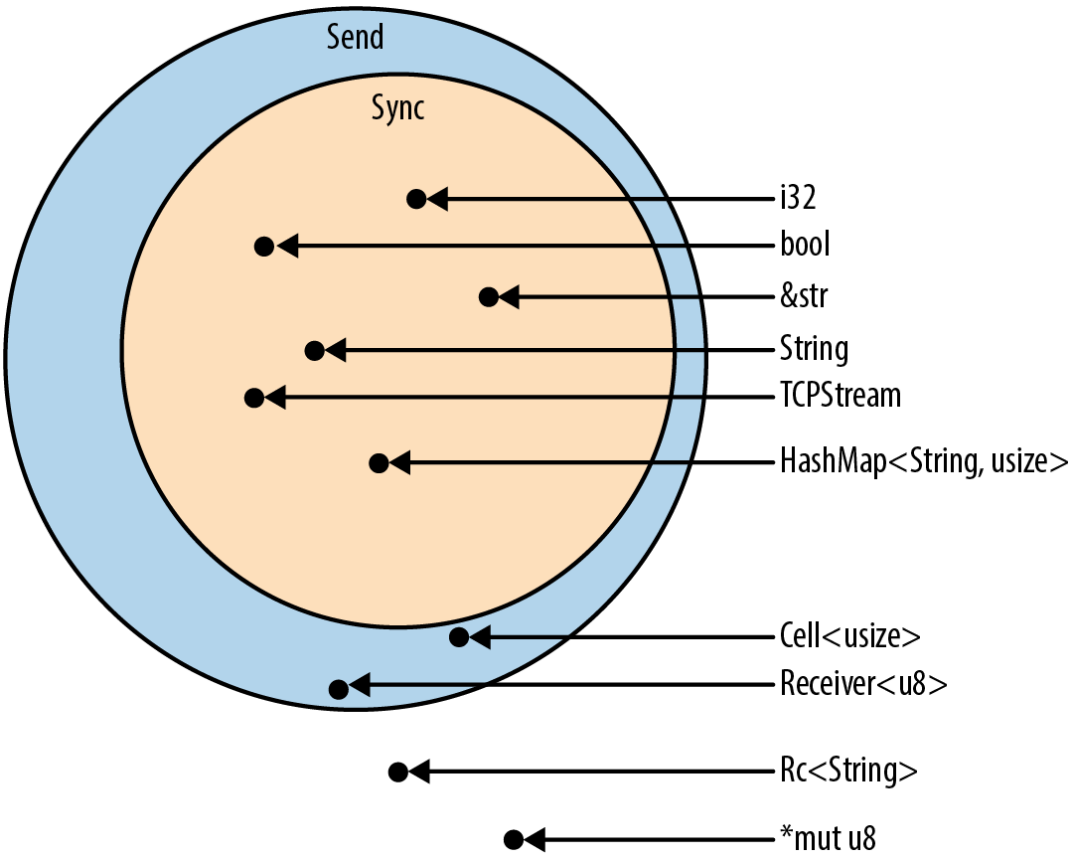
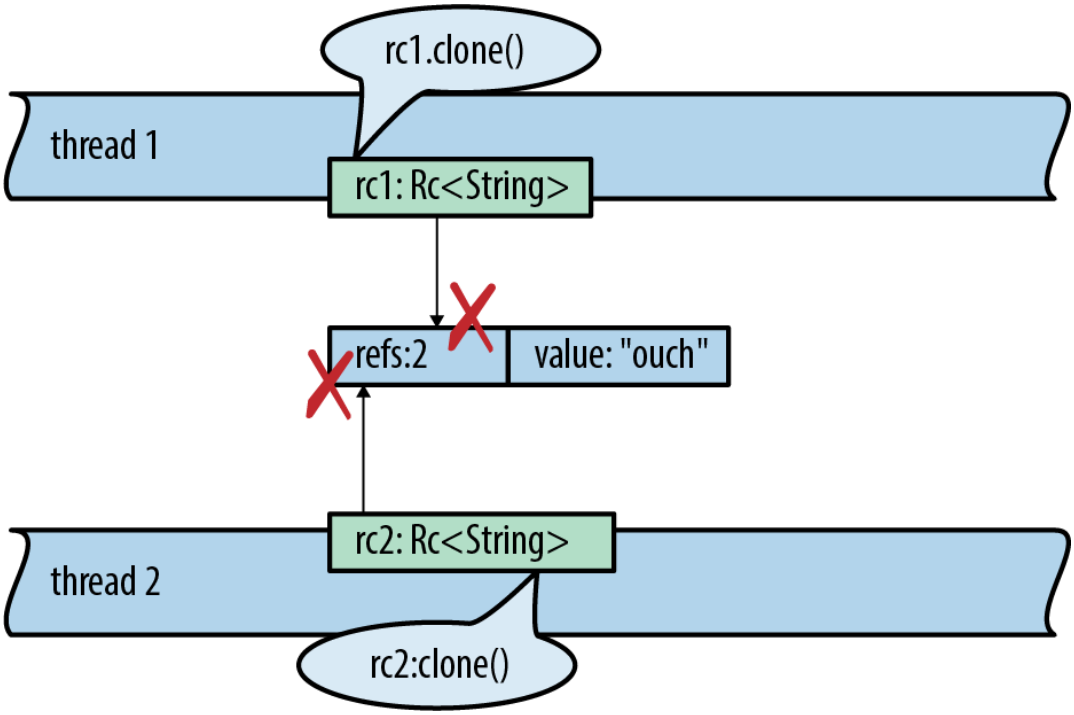


Figure 19-9. et types `Send Sync`

Que se passerait-il si , permettant aux threads de partager un seul via des références partagées ? Si les deux threads essaient de cloner le en même temps, comme illustré à la figure 19-10, nous avons une course de données car les deux threads incrémentent le nombre de références partagées. Le nombre de références peut devenir inexact, ce qui conduit à un comportement non défini d'utilisation après ou de double libre plus tard. `Rc<String> Sync Rc Rc`



Graphique 19-10. Pourquoi ni l'un ni l'autre `Rc<String>` `Sync` `Send`

Bien sûr, Rust empêche cela. Voici le code pour configurer cette course aux données :

```
use std::thread;
use std::rc::Rc;

fn main() {
    let rc1 = Rc::new("ouch".to_string());
    let rc2 = rc1.clone();
    thread::spawn(move || {  // error
        rc2.clone();
    });
    rc1.clone();
}
```

Rust refuse de le compiler, donnant un message d'erreur détaillé:

```
error: `Rc<String>` cannot be sent between threads safely
   |
10 |     thread::spawn(move || {  // error
   |             ^^^^^ `Rc<String>` cannot be sent between threads safel
   |
   = help: the trait `std::marker::Send` is not implemented for `Rc<Stri
   = note: required because it appears within the type `[closure@...]`
   = note: required by `std::thread::spawn`
```

Vous pouvez maintenant voir comment et aider Rust à appliquer la sécurité des fils. Ils apparaissent sous forme de limites dans la signature de type des fonctions qui transfèrent des données au-delà des limites de

thread. Lorsque vous êtes un thread, la fermeture que vous passez doit être , ce qui signifie que toutes les valeurs qu'il contient doivent être . De même, si vous souhaitez envoyer des valeurs via un canal à un autre thread, les valeurs doivent être . `Send Sync spawn Send Send Send`

Tuyauterie de presque n'importe quel itérateur vers un canal

Notre générateur d'index inversé est construit comme un pipeline. Le code est assez clair, mais il nous oblige à configurer manuellement des canaux et à lancer des threads. En revanche, les pipelines d'itérateurs que nous avons construits dans le chapitre 15 semblaient contenir beaucoup plus de travail en quelques lignes de code. Pouvons-nous construire quelque chose comme ça pour les pipelines de thread ?

En fait, ce serait bien si nous pouvions unifier les pipelines d'itérateurs et les pipelines de thread. Ensuite, notre générateur d'index pourrait être écrit en tant que pipeline d'itérateur. Cela pourrait commencer comme ceci:

```
documents.into_iter()
    .map(read_whole_file)
    .errors_to(error_sender)    // filter out error results
    .off_thread()               // spawn a thread for the above work
    .map(make_single_file_index)
    .off_thread()               // spawn another thread for stage 2
    ...
```

Les traits nous permettent d'ajouter des méthodes aux types de bibliothèque standard, de sorte que nous pouvons réellement le faire. Nous commençons par écrire un trait qui déclare la méthode que nous voulons:

```
use std::sync::mpsc;

pub trait OffThreadExt: Iterator {
    /// Transform this iterator into an off-thread iterator: the
    /// `next()` calls happen on a separate worker thread, so the
    /// iterator and the body of your loop run concurrently.
    fn off_thread(self) -> mpsc::IntoIter<Self::Item>;
}
```

Ensuite, nous implémentons ce trait pour les types d'itérateurs. Cela aide qui est déjà itérable: `mpsc::Receiver`

```
use std::thread;

impl<T> OffThreadExt for T
    where T: Iterator + Send + 'static,
          T::Item: Send + 'static
{
    fn off_thread(self) -> mpsc::IntoIter<Self::Item> {
        // Create a channel to transfer items from the worker thread.
        let (sender, receiver) = mpsc::sync_channel(1024);

        // Move this iterator to a new worker thread and run it there.
        thread::spawn(move || {
            for item in self {
                if sender.send(item).is_err() {
                    break;
                }
            }
        });

        // Return an iterator that pulls values from the channel.
        receiver.into_iter()
    }
}
```

La clause de ce code a été déterminée via un processus similaire à celui décrit dans « Reverse-Engineering Bounds ». Au début, nous avions juste ceci: `where`

```
impl<T> OffThreadExt for T
```

C'est-à-dire que nous voulions que la mise en œuvre fonctionne pour tous les itérateurs. Rust n'avait rien de tout cela. Étant donné que nous utilisons pour déplacer un itérateur de type vers un nouveau thread, nous devons spécifier . Étant donné que nous renvoyons les éléments via un canal, nous devons spécifier . Avec ces changements, Rust était satisfait. `spawn T T: Iterator + Send + 'static T::Item: Send + 'static`

C'est le caractère de Rust en un mot : nous sommes libres d'ajouter un outil d'alimentation de concurrence à presque tous les itérateurs de la langue, mais non sans d'abord comprendre et documenter les restrictions qui le rendent sûr à utiliser.

Au-delà des pipelines

Dans cette section, nous avons utilisé les pipelines comme exemples, car les pipelines sont un moyen simple et évident d'utiliser les canaux. Tout le monde les comprend. Ils sont concrets, pratiques et déterministes. Cependant, les canaux sont utiles pour plus que de simples pipelines. Ils constituent également un moyen rapide et facile d'offrir n'importe quel service asynchrone à d'autres threads dans le même processus.

Par exemple, supposons que vous souhaitiez effectuer la journalisation sur son propre thread, comme dans la figure 19-8. D'autres threads peuvent envoyer des messages de journalisation au thread de journalisation via un canal ; puisque vous pouvez cloner le canal , de nombreux threads clients peuvent avoir des expéditeurs qui expédient les messages de journalisation au même thread de journalisation. `Sender`

L'exécution d'un service comme la journalisation sur son propre thread présente des avantages. Le thread de journalisation peut faire pivoter les fichiers journaux chaque fois qu'il en a besoin. Il n'a pas besoin de faire de coordination sophistiquée avec les autres fils. Ces fils ne seront pas bloqués. Les messages s'accumuleront sans danger dans le canal pendant un moment jusqu'à ce que le fil de journalisation se remette au travail.

Les canaux peuvent également être utilisés dans les cas où un thread envoie une demande à un autre thread et a besoin d'obtenir une sorte de réponse. La requête du premier thread peut être une struct ou un tuple qui inclut un , une sorte d'enveloppe auto-adressée que le second thread utilise pour envoyer sa réponse. Cela ne signifie pas que l'interaction doit être synchrone. Le premier thread décide s'il faut bloquer et attendre la réponse ou utiliser la méthode pour l'interroger. `Sender .try_recv()`

Les outils que nous avons présentés jusqu'à présent – fork-join pour le calcul hautement parallèle, canaux pour connecter des composants lâches – sont suffisants pour un large éventail d'applications. Mais nous n'avons pas fini.

État mutable partagé

Dans les mois qui ont suivi la publication de la caisse dans le chapitre 8, votre logiciel de simulation de fougère a vraiment décollé. Maintenant, vous créez un jeu de stratégie multijoueur en temps réel dans lequel huit joueurs s'affrontent pour faire pousser principalement des fougères d'époque authentiques dans un paysage jurassique simulé. Le serveur de ce jeu est une application massivement parallèle, avec des requêtes affluant sur de nombreux threads. Comment ces fils peuvent-ils se coordonner

pour commencer une partie dès que huit joueurs sont disponibles ?
`fern_sim`

Le problème à résoudre ici est que de nombreux threads ont besoin d'accéder à une liste partagée de joueurs qui attendent de rejoindre un jeu. Ces données sont nécessairement à la fois mutables et partagées sur tous les threads. Si Rust n'a pas d'état mutable partagé, où cela nous laisse-t-il ?

Vous pouvez résoudre ce problème en créant un nouveau thread dont tout le travail consiste à gérer cette liste. D'autres fils communiqueraient avec elle via des canaux. Bien sûr, cela coûte un thread, qui a une certaine surcharge du système d'exploitation.

Une autre option consiste à utiliser les outils fournis par Rust pour partager en toute sécurité des données mutables. De telles choses existent. Ce sont des primitives de bas niveau qui seront familières à tout programmeur système qui a travaillé avec des threads. Dans cette section, nous couvrirons les mutex, les verrous en lecture/écriture, les variables de condition et les entiers atomiques. Enfin, nous montrerons comment implémenter des variables globales mutables dans Rust.

Qu'est-ce qu'un Mutex?

Un *mutex* (ou *verrou*) est utilisé pour forcer plusieurs threads à se relayer lors de l'accès à certaines données. Nous présenterons les mutex de Rust dans la section suivante. Tout d'abord, il est logique de se rappeler à quoi ressemblent les mutex dans d'autres langues. Une simple utilisation d'un mutex en C++ peut ressembler à ceci :

```
// C++ code, not Rust
void FernEmpireApp::JoinWaitingList(PlayerId player) {
    mutex.Acquire();

    waitingList.push_back(player);

    // Start a game if we have enough players waiting.
    if (waitingList.size() >= GAME_SIZE) {
        vector<PlayerId> players;
        waitingList.swap(players);
        StartGame(players);
    }

    mutex.Release();
}
```

Les appels et marquent le début et la fin d'une *section critique* dans ce code. Pour chacun d'un programme, un seul thread peut être exécuté dans une section critique à la fois. Si un thread se trouve dans une section critique, tous les autres threads qui appellent seront bloqués jusqu'à ce que le premier thread atteigne `.mutex.Acquire() mutex.Release() mutex mutex.Acquire() mute x.Release()`

Nous disons que le mutex *protège* les données: dans ce cas, protège . Il est de la responsabilité du programmeur, cependant, de s'assurer que chaque thread acquiert toujours le mutex avant d'accéder aux données, et le libère après. `mutex waitingList`

Les mutex sont utiles pour plusieurs raisons :

- Ils empêchent les *courses de données, des* situations où les threads de course lisent et écrivent simultanément la même mémoire. Les courses de données sont un comportement indéfini en C++ et Go. Les langages managés comme Java et C# promettent de ne pas planter, mais les résultats des courses de données sont toujours (pour résumer) absurdes.
- Même si les courses de données n'existaient pas, même si toutes les lectures et écritures se produisaient une par une dans l'ordre du programme, sans mutex, les actions des différents threads pouvaient s'entrelacer de manière arbitraire. Imaginez essayer d'écrire du code qui fonctionne même si d'autres threads modifient ses données pendant son exécution. Imaginez que vous essayez de le déboguer. Ce serait comme si votre programme était hanté.
- Mutexes prend en charge la programmation avec des *invariants,* des règles sur les données protégées qui sont vraies par construction lorsque vous les configurez et maintenues par chaque section critique.

Bien sûr, tout cela est vraiment la même raison: des conditions de course incontrôlées rendent la programmation insoluble. Les mutex apportent un peu d'ordre au chaos (mais pas autant d'ordre que les canaux ou le fork-join).

Cependant, dans la plupart des langues, les mutex sont très faciles à gâcher. En C++, comme dans la plupart des langages, les données et le verrou sont des objets distincts. Idéalement, les commentaires expliquent que chaque thread doit acquérir le mutex avant de toucher les données :

```
class FernEmpireApp {
    ...
```

```
private:
    // List of players waiting to join a game. Protected by `mutex`.
    vector<PlayerId> waitingList;

    // Lock to acquire before reading or writing `waitingList`.
    Mutex mutex;
    ...
};
```

Mais même avec de si beaux commentaires, le compilateur ne peut pas imposer un accès sécurisé ici. Lorsqu'un morceau de code néglige d'acquérir le mutex, nous obtenons un comportement indéfini. En pratique, cela signifie des bogues extrêmement difficiles à reproduire et à corriger.

Même en Java, où il existe une association notionnelle entre les objets et les mutex, la relation n'est pas très profonde. Le compilateur ne tente pas de l'appliquer et, dans la pratique, les données protégées par un verrou sont rarement exactement les champs de l'objet associé. Il inclut souvent des données dans plusieurs objets. Les schémas de verrouillage sont encore délicats. Les commentaires restent le principal outil pour les appliquer.

Mutex<T>

Nous allons maintenant montrer une implémentation de la liste d'attente dans Rust. Dans notre serveur de jeu Fern Empire, chaque joueur dispose d'un identifiant unique :

```
type PlayerId = u32;
```

La liste d'attente n'est qu'une collection de joueurs:

```
const GAME_SIZE: usize = 8;

/// A waiting list never grows to more than GAME_SIZE players.
type WaitingList = Vec<PlayerId>;
```

La liste d'attente est stockée sous la forme d'un champ de , un singleton qui est configuré dans un lors du démarrage du serveur. Chaque fil a un pointage vers lui. Il contient toute la configuration partagée et d'autres flotsam dont notre programme a besoin. La plupart de ces éléments sont en lecture seule. La liste d'attente étant à la fois partagée et modifiable, elle doit être protégée par un : `FernEmpireApp Arc Arc Mutex`

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

/// All threads have shared access to this big context struct.
struct FernEmpireApp {
    ...
    waiting_list: Mutex<WaitingList>,
    ...
}
```

Contrairement à C++, dans Rust, les données protégées sont stockées *à l'intérieur* du fichier . La configuration du ressemble à ceci: `Mutex Mutex`

```
use std::sync::Arc;

let app = Arc::new(FernEmpireApp {
    ...
    waiting_list: Mutex::new(vec![]),
    ...
});
```

La création d'un nouveau ressemble à la création d'un nouveau ou , mais tout en signifiant l'allocation de tas, il s'agit uniquement de verrouiller. Si vous voulez que votre allocation soit allouée dans le tas, vous devez le dire, comme nous l'avons fait ici en utilisant pour l'ensemble de l'application et juste pour les données protégées. Ces types sont couramment utilisés ensemble : ils sont pratiques pour partager des éléments entre les threads et sont pratiques pour les données modifiables partagées entre les
threads. `Mutex Box Arc Box Arc Mutex Mutex Arc::new Mutex::new Arc Mutex`

Maintenant, nous pouvons implémenter la méthode qui utilise le mutex: `join_waiting_list`

```
impl FernEmpireApp {
    /// Add a player to the waiting list for the next game.
    /// Start a new game immediately if enough players are waiting.
    fn join_waiting_list(&self, player: PlayerId) {
        // Lock the mutex and gain access to the data inside.
        // The scope of `guard` is a critical section.
        let mut guard = self.waiting_list.lock().unwrap();

        // Now do the game logic.
        guard.push(player);
        if guard.len() == GAME_SIZE {
```

```
                let players = guard.split_off(0);
                self.start_game(players);
            }
        }
    }
```

La seule façon d'accéder aux données est d'appeler la méthode : `.lock()`

```
    let mut guard = self.waiting_list.lock().unwrap();
```

`self.waiting_list.lock()` bloque jusqu'à ce que le mutex puisse être obtenu. La valeur renvoyée par cet appel de méthode est un wrapper mince autour d'un . Grâce aux coercitions deref, discutées, nous pouvons appeler des méthodes directement sur la garde: `MutexGuard<WaitingList> &mut WaitingList WaitingList`

```
    guard.push(player);
```

Le gardien nous permet même d'emprunter des références directes aux données sous-jacentes. Le système de durée de vie de Rust garantit que ces références ne peuvent pas survivre au gardien lui-même. Il n'y a aucun moyen d'accéder aux données dans un sans maintenir le verrou. `Mutex`

Lorsqu'il est déposé, le verrou est relâché. Normalement, cela se produit à la fin du bloc, mais vous pouvez également le déposer manuellement: `guard`

```
    if guard.len() == GAME_SIZE {
        let players = guard.split_off(0);
        drop(guard);  // don't keep the list locked while starting a game
        self.start_game(players);
    }
```

mut et Mutex

Il peut sembler étrange – certainement cela nous a semblé étrange au début – que notre méthode ne prenne pas par référence. Sa signature type est : `join_waiting_list self mut`

```
    fn join_waiting_list(&self, player: PlayerId)
```

La collection sous-jacente, , *nécessite* une référence lorsque vous appelez sa méthode. Sa signature type est : `Vec<PlayerId> mut push`

```
pub fn push(&mut self, item: T)
```

Et pourtant, ce code se compile et s'exécute correctement. Qu'est-ce qui se passe?

Dans Rust, signifie *un accès exclusif*. Plain signifie *accès partagé*. `&mut &`

Nous avons l'habitude de transmettre l'accès du parent à l'enfant, du conteneur au contenu. Vous ne vous attendez à pouvoir faire appel à des méthodes que si vous avez une référence pour commencer (ou si vous possédez, auquel cas félicitations pour être Elon Musk). C'est la valeur par défaut, car si vous n'avez pas un accès exclusif au parent, Rust n'a généralement aucun moyen de s'assurer que vous avez un accès exclusif à l'enfant. `&mut &mut`
`self starships[id].engine &mut starships starships`

Mais a un moyen: la serrure. En fait, un mutex n'est guère plus qu'un moyen de faire exactement cela, de fournir un accès *exclusif* () aux données à l'intérieur, même si de nombreux threads peuvent avoir *partagé* (non-) un accès à lui-même. `Mutex mut mut Mutex`

Le système de type Rust nous dit ce qui se passe. Il applique dynamiquement l'accès exclusif, ce qui est généralement fait statiquement, au moment de la compilation, par le compilateur Rust. `Mutex`

(Vous vous souviendrez peut-être que cela fait la même chose, sauf sans essayer de prendre en charge plusieurs threads. et sont les deux saveurs de mutabilité intérieure, que nous avons couvertes.) `std::cell::RefCell Mutex RefCell`

Pourquoi les mutex ne sont pas toujours une bonne idée

Avant de commencer sur les mutex, nous avons présenté quelques approches de la concurrence d'accès qui auraient pu sembler étrangement faciles à utiliser correctement si vous venez de C ++. Ce n'est pas un hasard : ces approches sont conçues pour fournir des garanties solides contre les aspects les plus déroutants de la programmation simultanée. Les programmes qui utilisent exclusivement le parallélisme fork-join sont déterministes et ne peuvent pas bloquer. Les programmes qui utilisent des canaux se comportent presque aussi bien. Ceux qui utilisent des canaux exclusivement pour le pipelining, comme notre générateur d'index, sont déterministes : le moment de la remise des messages peut vari-

er, mais cela n'affectera pas la sortie. Et ainsi de suite. Les garanties sur les programmes multithread sont agréables!

La conception de Rust vous fera presque certainement utiliser des mutex plus systématiquement et plus judicieusement que jamais auparavant. Mais il vaut la peine de faire une pause et de réfléchir à ce que les garanties de sécurité de Rust peuvent et ne peuvent pas aider. `Mutex`

Le code Safe Rust ne peut pas déclencher une *course aux données*, un type spécifique de bogue où plusieurs threads lisent et écrivent la même mémoire simultanément, produisant des résultats dénués de sens. C'est génial : les courses de données sont toujours des bugs, et ils ne sont pas rares dans les vrais programmes multithread.

Cependant, les threads qui utilisent des mutex sont sujets à d'autres problèmes que Rust ne résout pas pour vous :

- Les programmes Rust valides ne peuvent pas avoir de courses de données, mais ils peuvent toujours avoir d'autres *conditions de course*, des situations où le comportement d'un programme dépend du timing entre les threads et peut donc varier d'une exécution à l'autre. Certaines conditions de course sont bénignes. Certains se manifestent par des flocons généraux et des bogues incroyablement difficiles à corriger. L'utilisation de mutex de manière non structurée invite à des conditions de course. C'est à vous de vous assurer qu'ils sont bénins.
- L'état mutable partagé affecte également la conception du programme. Là où les canaux servent de limite d'abstraction dans votre code, ce qui facilite la séparation des composants isolés à des fins de test, les mutex encouragent une méthode de travail « juste-ajouter-une-méthode » qui peut conduire à un blob monolithique de code interdépendant.
- Enfin, les mutex ne sont tout simplement pas aussi simples qu'ils le semblent au premier abord, comme le montreront les deux sections suivantes.

Tous ces problèmes sont inhérents aux outils. Utilisez une approche plus structurée lorsque vous le pouvez; utilisez un quand vous devez. `Mutex`

Impasse

Un thread peut se bloquer en essayant d'acquérir un verrou qu'il détient déjà :

```
let mut guard1 = self.waiting_list.lock().unwrap();
let mut guard2 = self.waiting_list.lock().unwrap();  // deadlock
```

Supposons que le premier appel réussisse, en prenant le verrou. Le deuxième appel voit que le verrou est maintenu, il se bloque, attendant qu'il soit libéré. Il attendra pour toujours. Le fil d'attente est celui qui maintient la serrure. `self.waiting_list.lock()`

En d'autres termes, le verrou dans un n'est pas un verrou récursif. `Mutex`

Ici, le bug est évident. Dans un programme réel, les deux appels peuvent être dans deux méthodes différentes, dont l'une appelle l'autre. Le code de chaque méthode, pris séparément, aurait l'air bien. Il existe également d'autres moyens d'obtenir un blocage, impliquant plusieurs threads qui acquièrent chacun plusieurs mutex à la fois. Le système d'emprunt de Rust ne peut pas vous protéger contre les blocages. La meilleure protection est de garder les sections critiques petites: entrez, faites votre travail et sortez. `lock()`

Il est également possible d'obtenir une impasse avec les canaux. Par exemple, deux threads peuvent se bloquer, chacun attendant de recevoir un message de l'autre. Cependant, encore une fois, une bonne conception de programme peut vous donner une grande confiance que cela ne se produira pas dans la pratique. Dans un pipeline, comme notre générateur d'index inversé, le flux de données est acyclique. L'impasse est aussi improbable dans un tel programme que dans un pipeline shell Unix.

Mutex empoisonnés

`Mutex::lock()` renvoie un pour la même raison que : échouer gracieusement si un autre thread a paniqué. Lorsque nous écrivons, nous disons à Rust de propager la panique d'un fil à l'autre. L'idiome est similaire. `Result JoinHandle::join() handle.join().unwrap() mutex .lock().unwrap()`

Si un fil panique en tenant un , Rust marque le comme *empoisonné*. Toute tentative ultérieure à l'empoisonné obtiendra un résultat d'erreur. Notre appel dit à Rust de paniquer si cela se produit, propageant la panique de l'autre fil à celui-ci. `Mutex Mutex lock Mutex .unwrap()`

À quel point est-ce mauvais d'avoir un mutex empoisonné? Le poison semble mortel, mais ce scénario n'est pas nécessairement fatal. Comme nous l'avons dit au chapitre 7, la panique est sans danger. Un fil de panique laisse le reste du programme dans un état sûr.

La raison pour laquelle les mutex sont empoisonnés par la panique n'est donc pas par peur d'un comportement indéfini. Le problème est plutôt

que vous avez probablement programmé avec des invariants. Puisque votre programme a paniqué et renfloué une section critique sans avoir terminé ce qu'il faisait, peut-être après avoir mis à jour certains champs des données protégées mais pas d'autres, il est possible que les invariants soient maintenant cassés. La rouille empoisonne le mutex pour empêcher d'autres fils de gaffer involontairement dans cette situation brisée et de l'aggraver. Vous *pouvez* toujours verrouiller un mutex empoisonné et accéder aux données à l'intérieur, avec une exclusion mutuelle pleinement appliquée; reportez-vous à la documentation pour . Mais vous ne le ferez pas par accident. `PoisonError::into_inner()`

Canaux multiconsommateurs utilisant Mutexes

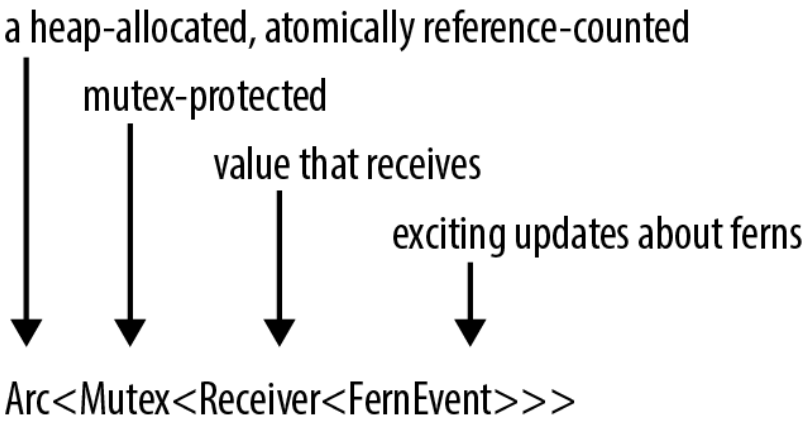
Nous avons mentionné plus tôt que les chaînes de Rust sont plusieurs producteurs, un seul consommateur. Ou pour le dire plus concrètement, un canal n'en a qu'un. Nous ne pouvons pas avoir un pool de threads où de nombreux threads utilisent un seul canal comme liste de travail partagée. `Receiver mpsc`

Cependant, il s'avère qu'il existe une solution de contournement très simple, en utilisant uniquement des éléments de bibliothèque standard. Nous pouvons ajouter un autour de la et le partager de toute façon. Voici un module qui le fait : `Mutex Receiver`

```
pub mod shared_channel {
    use std::sync::{Arc, Mutex};
    use std::sync::mpsc::{channel, Sender, Receiver};

    /// A thread-safe wrapper around a `Receiver`.
    #[derive(Clone)]
    pub struct SharedReceiver<T>(Arc<Mutex<Receiver<T>>>);

    impl<T> Iterator for SharedReceiver<T> {
        type Item = T;

        /// Get the next item from the wrapped receiver.
        fn next(&mut self) -> Option<T> {
            let guard = self.0.lock().unwrap();
            guard.recv().ok()
        }
    }

    /// Create a new channel whose receiver can be shared across threads
    /// This returns a sender and a receiver, just like the stdlib's
    /// `channel()`, and sometimes works as a drop-in replacement.
```

```
        pub fn shared_channel<T>() -> (Sender<T>, SharedReceiver<T>) {
            let (sender, receiver) = channel();
            (sender, SharedReceiver(Arc::new(Mutex::new(receiver))))
        }
    }
```

Nous utilisons un fichier . Les génériques se sont vraiment empilés. Cela se produit plus souvent dans Rust qu'en C++. Il peut sembler que cela deviendrait déroutant, mais souvent, comme dans ce cas, la simple lecture des noms peut aider à expliquer ce qui se passe, comme le montre la figure 19-11. `Arc<Mutex<Receiver<T>>>`



Graphique 19-11. Comment lire un type complexe

Verrous en lecture/écriture (RwLock<T>)

Passons maintenant des mutex aux autres outils fournis dans , la boîte à outils de synchronisation de thread de bibliothèque standard de Rust. Nous allons avancer rapidement, car une discussion complète de ces outils dépasse le cadre de ce livre. `std::sync`

Les programmes serveur ont souvent des informations de configuration qui sont chargées une fois et qui changent rarement. La plupart des threads interrogent uniquement la configuration, mais comme la configuration *peut* changer (il peut être possible de demander au serveur de recharger sa configuration à partir du disque, par exemple), elle doit de toute façon être protégée par un verrou. Dans des cas comme celui-ci, un mutex peut fonctionner, mais c'est un goulot d'étranglement inutile. Les threads ne devraient pas avoir à interroger la configuration à tour de rôle si elle ne change pas. C'est le cas d'un *verrou en lecture/écriture*, ou . `RwLock`

Alors qu'un mutex a une seule méthode, un verrou en lecture/écriture a deux méthodes de verrouillage et . La méthode est comme . Il attend un accès exclusif aux données protégées. La méthode fournit un non-accès, avec l'avantage qu'il est moins susceptible d'avoir à attendre, car de nom-

breux threads peuvent lire en toute sécurité à la fois. Avec un mutex, à tout moment, les données protégées n'ont qu'un seul lecteur ou écrivain (ou aucun). Avec un verrou de lecture / écriture, il peut avoir un écrivain ou plusieurs lecteurs, un peu comme les références Rust en général. `lock read write RwLock::write Mutex::lock mut RwLock ::read mut`

`FernEmpireApp` peut avoir une structure pour la configuration, protégée par un : `RwLock`

```
use std::sync::RwLock;

struct FernEmpireApp {
    ...
    config: RwLock<AppConfig>,
    ...
}
```

Les méthodes qui lisent la configuration utiliseraient : `RwLock::read()`

```
/// True if experimental fungus code should be used.
fn mushrooms_enabled(&self) -> bool {
    let config_guard = self.config.read().unwrap();
    config_guard.mushrooms_enabled
}
```

La méthode de rechargement de la configuration utiliserait : `RwLock::write()`

```
fn reload_config(&self) -> io::Result<()> {
    let new_config = AppConfig::load()?;
    let mut config_guard = self.config.write().unwrap();
    *config_guard = new_config;
    Ok(())
}
```

Rust, bien sûr, est particulièrement bien adapté pour appliquer les règles de sécurité sur les données. Le concept d'écrivain unique ou de lecteur multiple est au cœur du système d'emprunt de Rust. renvoie un garde qui fournit un accès non (partagé) au ; renvoie un autre type de garde qui fournit un accès (exclusif). `RwLock self.config.read() mut AppConfig self.config .write() mut`

Variables de condition (Condvar)

Souvent, un thread doit attendre qu'une certaine condition devienne vraie:

- Pendant l'arrêt du serveur, le thread principal peut avoir besoin d'attendre que tous les autres threads aient fini de se séparer.
- Lorsqu'un thread de travail n'a rien à faire, il doit attendre qu'il y ait des données à traiter.
- Un thread implémentant un protocole de consensus distribué peut devoir attendre qu'un quorum de pairs ait répondu.

Parfois, il existe une API de blocage pratique pour la condition exacte que nous voulons attendre, comme pour l'exemple d'arrêt du serveur. Dans d'autres cas, il n'y a pas d'API de blocage intégrée. Les programmes peuvent utiliser des *variables de condition* pour créer les leurs. Dans Rust, le type implémente des variables de condition. A a des méthodes et ; bloque jusqu'à ce qu'un autre thread appelle `.JoinHandle::join std::sync::Condvar Condvar .wait() .notify_all() .wait() .notify_all()`

Il y a un peu plus que cela, car une variable de condition concerne toujours une condition vraie ou fausse particulière à propos de certaines données protégées par un . Ceci et le sont donc liés. Une explication complète est plus que ce que nous avons de la place ici, mais pour le bénéfice des programmeurs qui ont déjà utilisé des variables de condition, nous allons montrer les deux bits clés du code. `Mutex Mutex Condvar`

Lorsque la condition souhaitée devient vraie, nous appelons (ou) pour réveiller les fils d'attente : `Condvar::notify_all notify_one`

```
self.has_data_condvar.notify_all();
```

Pour s'endormir et attendre qu'une condition devienne vraie, nous utilisons : `Condvar::wait()`

```
while !guard.has_data() {
    guard = self.has_data_condvar.wait(guard).unwrap();
}
```

Cette boucle est un idiome standard pour les variables de condition. Cependant, la signature de est inhabituelle. Il prend un objet par valeur, le consomme et renvoie un nouveau sur le succès. Cela capture l'intuition que la méthode libère le mutex, puis le réacquiert avant de revenir. Pass-

er la valeur par est une façon de dire : « Je vous accorde, méthode, mon autorité exclusive pour libérer le mutex.
» `while Condvar::wait MutexGuard MutexGuard wait MutexGuard . wait()`

Atomes

Le module contient des types atomiques pour une programmation simultanée sans verrouillage. Ces types sont fondamentalement les mêmes que les atomes C ++ standard, avec quelques extras: `std::sync::atomic`

- `AtomicIsize` et sont des types entiers partagés correspondant au thread unique et aux types. `AtomicUsize isize usize`
- `AtomicI8` , , , , et leurs variantes non signées comme sont des types entiers partagés qui correspondent aux types monothread , , etc. `AtomicI16 AtomicI32 AtomicI64 AtomicU8 i8 i16`
- An est une valeur partagée. `AtomicBool bool`
- An est une valeur partagée du type de pointeur non sécurisé . `AtomicPtr<T> *mut T`

L'utilisation correcte des données atomiques dépasse le cadre de ce livre. Il suffit de dire que plusieurs threads peuvent lire et écrire une valeur atomique à la fois sans provoquer de courses de données.

Au lieu des opérateurs arithmétiques et logiques habituels, les types atomiques exposent des méthodes qui effectuent des *opérations atomiques,* des charges individuelles, des magasins, des échanges et des opérations arithmétiques qui se produisent en toute sécurité, en tant qu'unité, même si d'autres threads effectuent également des opérations atomiques qui touchent le même emplacement de mémoire. L'incrémentation d'un nom ressemble à ceci : `AtomicIsize atom`

```
use std::sync::atomic::{AtomicIsize, Ordering};

let atom = AtomicIsize::new(0);
atom.fetch_add(1, Ordering::SeqCst);
```

Ces méthodes peuvent être compilées en instructions spécialisées en langage machine. Sur l'architecture x86-64, cet appel se compile en une instruction, où un ordinaire peut compiler en une instruction simple ou un certain nombre de variations sur ce thème. Le compilateur Rust doit également renoncer à certaines optimisations autour de l'opération atomique, car, contrairement à une charge ou un stockage normal, il peut

légitimement affecter ou être affecté par d'autres threads immédiatement. `.fetch_add() lock incq n += 1 incq`

L'argument est un *ordre de mémoire.* Les ordres de mémoire sont quelque chose comme les niveaux d'isolement des transactions dans une base de données. Ils disent au système à quel point vous vous souciez de notions philosophiques telles que les causes des effets précédents et le temps n'ayant pas de boucles, par opposition à la performance. Les ordres de mémoire sont cruciaux pour l'exactitude du programme, et ils sont difficiles à comprendre et à raisonner. Heureusement, la pénalité de performance pour le choix de la cohérence séquentielle, l'ordre de mémoire le plus strict, est souvent assez faible, contrairement à la pénalité de performance pour la mise en mode d'une base de données SQL. Donc, en cas de doute, utilisez . Rust hérite de plusieurs autres ordres de mémoire des atomes C++ standard, avec diverses garanties plus faibles sur la nature de l'existence et la causalité. Nous n'en discuterons pas ici. `Ordering::SeqCst SERIALIZABLE Ordering::SeqCst`

Une utilisation simple des atomes est pour l'annulation. Supposons que nous ayons un thread qui effectue des calculs de longue durée, tels que le rendu d'une vidéo, et que nous aimerions pouvoir l'annuler de manière asynchrone. Le problème est de communiquer au thread que nous voulons qu'il s'arrête. Nous pouvons le faire via un partage : `AtomicBool`

```
use std::sync::Arc;
use std::sync::atomic::AtomicBool;

let cancel_flag = Arc::new(AtomicBool::new(false));
let worker_cancel_flag = cancel_flag.clone();
```

Ce code crée deux pointeurs intelligents qui pointent vers le même tas alloué , dont la valeur initiale est . Le premier, nommé , restera dans le fil principal. Le second, , sera déplacé vers le thread de travail. `Arc<AtomicBool> AtomicBool false cancel_flag worker_cancel_flag`

Voici le code du travailleur :

```
use std::thread;
use std::sync::atomic::Ordering;

let worker_handle = thread::spawn(move || {
    for pixel in animation.pixels_mut() {
        render(pixel); // ray-tracing - this takes a few microseconds
```

```
            if worker_cancel_flag.load(Ordering::SeqCst) {
                return None;
            }
        }
        Some(animation)
    });
```

Après avoir rendu chaque pixel, le thread vérifie la valeur de l'indicateur en appelant sa méthode : `.load()`

```
worker_cancel_flag.load(Ordering::SeqCst)
```

Si, dans le thread principal, nous décidons d'annuler le thread de travail, nous stockons dans le, puis attendons que le thread se ferme
: `true AtomicBool`

```
// Cancel rendering.
cancel_flag.store(true, Ordering::SeqCst);

// Discard the result, which is probably `None`.
worker_handle.join().unwrap();
```

Bien sûr, il existe d'autres moyens de mettre cela en œuvre. L'ici pourrait être remplacé par un ou un canal. La principale différence est que les atomes ont une surcharge minimale. Les opérations atomiques n'utilisent jamais d'appels système. Une charge ou un stockage se compile souvent en une seule instruction CPU. `AtomicBool Mutex<bool>`

Les atomes sont une forme de mutabilité intérieure, comme ou , de sorte que leurs méthodes prennent par référence partagée (non-). Cela les rend utiles en tant que variables globales simples. `Mutex RwLock self mut`

Variables globales

Supposons que nous écrivions du code réseau. Nous aimerions avoir une variable globale, un compteur que nous incrémentons chaque fois que nous servons un paquet :

```
/// Number of packets the server has successfully handled.
static PACKETS_SERVED: usize = 0;
```

Cela compile bien. Il n'y a qu'un seul problème. n'est pas mutable, donc nous ne pouvons jamais le changer. `PACKETS_SERVED`

Rust fait tout ce qu'elle peut raisonnablement pour décourager l'état mutable mondial. Les constantes déclarées avec sont, bien sûr, immuables. Les variables statiques sont également immuables par défaut, il n'y a donc aucun moyen d'obtenir une référence à une. A peut être déclaré , mais alors y accéder n'est pas sûr. L'insistance de Rust sur la sécurité des fils est une raison majeure de toutes ces règles. `const mut static mut`

L'état global mutable a également des conséquences malheureuses en génie logiciel: il a tendance à rendre les différentes parties d'un programme plus étroitement couplées, plus difficiles à tester et plus difficiles à modifier plus tard. Pourtant, dans certains cas, il n'y a tout simplement pas d'alternative raisonnable, nous ferions donc mieux de trouver un moyen sûr de déclarer des variables statiques mutables.

Le moyen le plus simple de prendre en charge l'incrémentation, tout en le gardant thread-safe, est d'en faire un entier atomique : `PACKETS_SERVED`

```
use std::sync::atomic::AtomicUsize;

static PACKETS_SERVED: AtomicUsize = AtomicUsize::new(0);
```

Une fois cette statique déclarée, l'incrémentation du nombre de paquets est simple :

```
use std::sync::atomic::Ordering;

PACKETS_SERVED.fetch_add(1, Ordering::SeqCst);
```

Les globaux atomiques sont limités aux entiers simples et aux booléens. Néanmoins, créer une variable globale de tout autre type revient à résoudre deux problèmes.

Tout d'abord, la variable doit être rendue thread-safe d'une manière ou d'une autre, car sinon elle ne peut pas être globale : pour la sécurité, les variables statiques doivent être à la fois et non-. Heureusement, nous avons déjà vu la solution à ce problème. Rust a des types pour partager en toute sécurité des valeurs qui changent: , et les types atomiques. Ces types peuvent être modifiés même lorsqu'ils sont déclarés comme non-. C'est ce qu'ils font. (Voir « mut et Mutex ».) `Sync mut Mutex RwLock mut`

Deuxièmement, les initialiseurs statiques ne peuvent appeler que des fonctions spécifiquement marquées comme , que le compilateur peut évaluer pendant la compilation. En d'autres termes, leur production est déterministe; cela ne dépend que de leurs arguments, pas d'un autre état

ou d'E/S. De cette façon, le compilateur peut intégrer les résultats de ce calcul en tant que constante de temps de compilation. Ceci est similaire à C ++ . `const constexpr`

Les constructeurs pour les types (, , et ainsi de suite) sont tous des fonctions, ce qui nous a permis de créer un précédent. Quelques autres types, comme , et , ont des constructeurs simples qui le sont aussi. `Atomic AtomicUsize AtomicBool const static AtomicUsize String Ipv4Addr Ipv6Addr const`

Vous pouvez également définir vos propres fonctions en préfixant simplement la signature de la fonction par . Rust limite ce que les fonctions peuvent faire à un petit ensemble d'opérations, qui sont suffisantes pour être utiles tout en ne permettant aucun résultat non déterministe. les fonctions ne peuvent pas prendre les types comme arguments génériques, seulement les durées de vie, et il n'est pas possible d'allouer de la mémoire ou de fonctionner sur des pointeurs bruts, même en blocs. Nous pouvons cependant utiliser des opérations arithmétiques (y compris l'encapsulation et l'arithmétique saturée), des opérations logiques qui ne court-circuitent pas et d'autres fonctions. Par exemple, nous pouvons créer des fonctions pratiques pour faciliter la définition des s et s et réduire la duplication du code
: `const const const const unsafe const static const`

```
const fn mono_to_rgba(level: u8) -> Color {
    Color {
        red: level,
        green: level,
        blue: level,
        alpha: 0xFF
    }
}

const WHITE: Color = mono_to_rgba(255);
const BLACK: Color = mono_to_rgba(000);
```

En combinant ces techniques, nous pourrions être tentés d'écrire:

```
static HOSTNAME: Mutex<String> =
    Mutex::new(String::new());  // error: calls in statics are limited t
                                // constant functions, tuple structs, an
                                // tuple variants
```

Malheureusement, alors que et sont , n'est pas. Afin de contourner ces limitations, nous devons utiliser la caisse. `AtomicUsize::new() String::new() const fn Mutex::new() lazy_static`

Nous avons introduit la caisse dans « Building Regex Values Lazily ». La définition d'une variable avec la macro vous permet d'utiliser n'importe quelle expression de votre choix pour l'initialiser ; il s'exécute la première fois que la variable est déréférencée et la valeur est enregistrée pour toutes les utilisations ultérieures. `lazy_static lazy_static!`

Nous pouvons déclarer un global -contrôlé avec comme ceci: `Mutex HashMap lazy_static`

```
use lazy_static::lazy_static;

use std::sync::Mutex;

lazy_static! {
    static ref HOSTNAME: Mutex<String> = Mutex::new(String::new());
}
```

La même technique fonctionne pour d'autres structures de données complexes comme s et s. C'est également très pratique pour les statiques qui ne sont pas du tout mutables, mais qui nécessitent simplement une initialisation non triviale. `HashMap Deque`

L'utilisation impose un coût de performance minime à chaque accès aux données statiques. L'implémentation utilise , une primitive de synchronisation de bas niveau conçue pour une initialisation unique. Dans les coulisses, chaque fois qu'une statique paresseuse est consultée, le programme exécute une instruction de charge atomique pour vérifier que l'initialisation a déjà eu lieu. (est un but assez spécial, nous ne le couvrirons donc pas en détail ici. Il est généralement plus pratique à utiliser à la place. Cependant, il est pratique pour initialiser des bibliothèques non-Rust; pour obtenir un exemple, voir « Une interface sécurisée vers libgit2 ».) `lazy_static! std::sync::Once Once lazy_static!`

À quoi ressemble le piratage de code simultané dans Rust

Nous avons montré trois techniques d'utilisation des threads dans Rust : le parallélisme fourche-jointure, les canaux et l'état mutable partagé avec

des verrous. Notre objectif a été de fournir une bonne introduction aux pièces fournies par Rust, en mettant l'accent sur la façon dont elles peuvent s'intégrer dans de vrais programmes.

Rust insiste sur la sécurité, donc à partir du moment où vous décidez d'écrire un programme multithread, l'accent est mis sur la construction d'une communication sûre et structurée. Garder les fils principalement isolés est un bon moyen de convaincre Rust que ce que vous faites est sûr. Il arrive que l'isolement soit également un bon moyen de s'assurer que ce que vous faites est correct et maintenable. Encore une fois, Rust vous guide vers de bons programmes.

Plus important encore, Rust vous permet de combiner des techniques et des expériences. Vous pouvez itérer rapidement : discuter avec le compilateur vous permet d'être opérationnel correctement beaucoup plus rapidement que de déboguer des courses de données.