

Notes of the course

Programmazione di Sistema - API

Politecnico di Torino
Prof. Giovanni Malnati

Anno 2022/23

Luigi Federico

Indice

Indice	1
11 - Smart Pointer	3
11.1 - Puntatori	3
11.2 - Smart pointer in Rust	4
std::Box<T>	4
std::rc::Rc<T>	6
std::rc::Weak<T>	7
std::cell::Cell<T>	8
std::cell::RefCell<T>	9
std::borrow::Cow<'a, B>	10
Smart Pointer e metodi	11
13 - Concorrenza	12
13.1 - Programmazione concorrente	12
Concorrenza in pratica	13
Modello di memoria	14
Problemi aperti	15
Errori	15
13.2 - Thread	15
Esecuzione e non determinismo	16
Sincronizzazione	17
Accesso condiviso: problemi e soluzioni	18
Uso dei thread	19
I tratti della concorrenza	20
Modelli di concorrenza	22
13.3 - Mutex e puntatori concorrenti	22
Rilasciare i mutex	23
Mutex in Rust	24
Tipi atomici	28
Dipendenze circolari	29
13.4 - Condition variables	29
Metodi principali	30
Meccanismo di funzionamento	32
Notifiche spurie e notifiche perse	32
Attesa temporizzata	33
13.5 - Condivisione di messaggi	34
Canali sincroni	36
13.6 - Crossbeam	36
Modello degli attori	39
14 - Processi	40
14.1 - Processi e isolamento	40
Concorrenza e processi	40

Terminare un processo	42
14.2 - Gestione dei processi	43
Creare un processo	43
Terminare un processo	46
Gestire altri processi	46
14.3 - InterProcess Communication (IPC)	48
Coda di messaggi	49
Pipe	49
Scambio di messaggi strutturati	50
Comunicazione tra processi	52
15 - Programmazione asincrona	54
15.1 - Esecuzione asincrona	54
Elaborazione asincrona	55
Esecuzione parziale	56
15.2 - Async e await	57
Il tratto Future	58
Generare la macchina a stati	59
Gestire l'esecuzione	62
15.3 - The Tokio framework	63
Gestione del tempo	65
Eseguire compiti computazionalmente intensi	65
Condividere dati tra task	66
Prestazioni a confronto	69

11 - Smart Pointer

11.1 - Puntatori

Ogni valore manipolato da un programma è memorizzato nello **spazio di indirizzamento** del processo.

- L'operatore **&** (e **&mut** in Rust) permette in C, C++ e Rust di ottenere l'indirizzo del primo byte in cui è memorizzato (**referencing**).
- In Rust, questi operatori attivano il borrow checker che impone i suoi vincoli sull'utilizzo dei riferimenti.

L'operazione duale, detta dereferenza (**dereferencing**) o risoluzione del riferimento, trasforma un indirizzo nel corrispondente valore puntato.

- Si usa l'operatore ***** (e **→** in C e C++) o **.** in Rust.
- Quando viene applicato ad un puntatore nativo o ad un riferimento Rust, il compilatore dà accesso al dato puntato.

Rust permette di **ridfinire il comportamento degli operatori del linguaggio per tipi arbitrari**. Permettendo anche di definire **tipi generici**, che possono essere espansi in una molteplicità di tipi concreti, in funzione di come vengono utilizzati.

- Questi meccanismi, applicati agli operatori di dereferenza, abilitano la definizione di **tipi che “sembrano” puntatori** sintatticamente ma che hanno ulteriori caratteristiche:
 - Garanzia di inizializzazione e rilascio
 - Conteggio dei riferimenti
 - Accesso esclusivo con attesa
 - etc..
- Questo ha portato al concetto di **smart pointer** e al suo utilizzo nelle librerie standard Rust, che sfruttano l'idea per rappresentare puntatori che possiedono i dati a cui puntano (in contrapposizione ai riferimenti, che godono del solo prestito).

Quindi:

- Smart pointer → puntatore che possiede i dati a cui punta.
- Riferimento → puntatore che prende in prestito (non possiede) il dato puntato.

Tramite l'uso di puntatori è possibile costruire **strutture dati dinamiche** come grafi, alberi e liste.

- Questo lascia grandi libertà al programmatore ma espone a problemi legati alla difficoltà di dedurre la corrispondenza del codice che lo manipola.
- Le regole restrittive del **borrow checker** di Rust **impediscono**, con l'uso di soli riferimenti, la **creazione di strutture cicliche**.
 - Ogni valore in Rust è parte di un solo albero la cui radice è contenuta in una qualche variabile.

Attraverso l'uso di smart pointer come `Rc<T>` e `Arc<T>`, è possibile avere più possessori di uno stesso valore. Smart pointer come `std::rc::Weak` e `std::sync::Weak` offrono invece la possibilità di avere strutture cicliche, nel rispetto di alcune restrizioni.

11.2 - Smart pointer in Rust

Rust offre una varietà di smart pointer allo scopo di coprire vari casi e definire ottimizzazioni (`Box<T>`, `Rc<T>`, `Arc<T>`, `Weak<T>`, `Cell<T>`, `RefCell<T>`, `Cow<T>`, `Mutex<T>`, `RwLock<T>`)

- In generale sono **realizzati mediante struct** che contengono le necessarie informazioni e che **implementano i tratti `Deref` e `DerefMut`**.
- Quando il compilatore incontra l'espressione `*ptr` (dove il tipo di `ptr` implementa tali tratti) la trasforma in `*ptr.deref()` e `*ptr.deref_mut()` a seconda dei casi.

```
pub trait Deref {  
    type Target: ?Sized;  
  
    // Required method  
    fn deref(&self) -> &Self::Target;  
}
```

```
pub trait DerefMut: Deref {  
    // Required method  
    fn deref_mut(&mut self) -> &mut Self::Target;  
}
```

`std::Box<T>`

Struttura che **incapsula un puntatore ad un blocco allocato** dinamicamente **sullo heap** all'atto della sua costruzione (tramite il metodo `Box::new(t)`).

- Il **dato puntato è posseduto da Box**: quando la struttura esce dal proprio scope sintattico, il blocco sullo heap viene rilasciato automaticamente, grazie all'implementazione del tratto `Drop`.
- È possibile anticipare il rilascio del blocco, invocando la funzione `drop(b)`.

Se la **struttura viene mossa in un'altra variabile** (o ritornata da una funzione), il **possesso** del puntatore **passa alla destinazione** che diventa responsabile del suo rilascio.

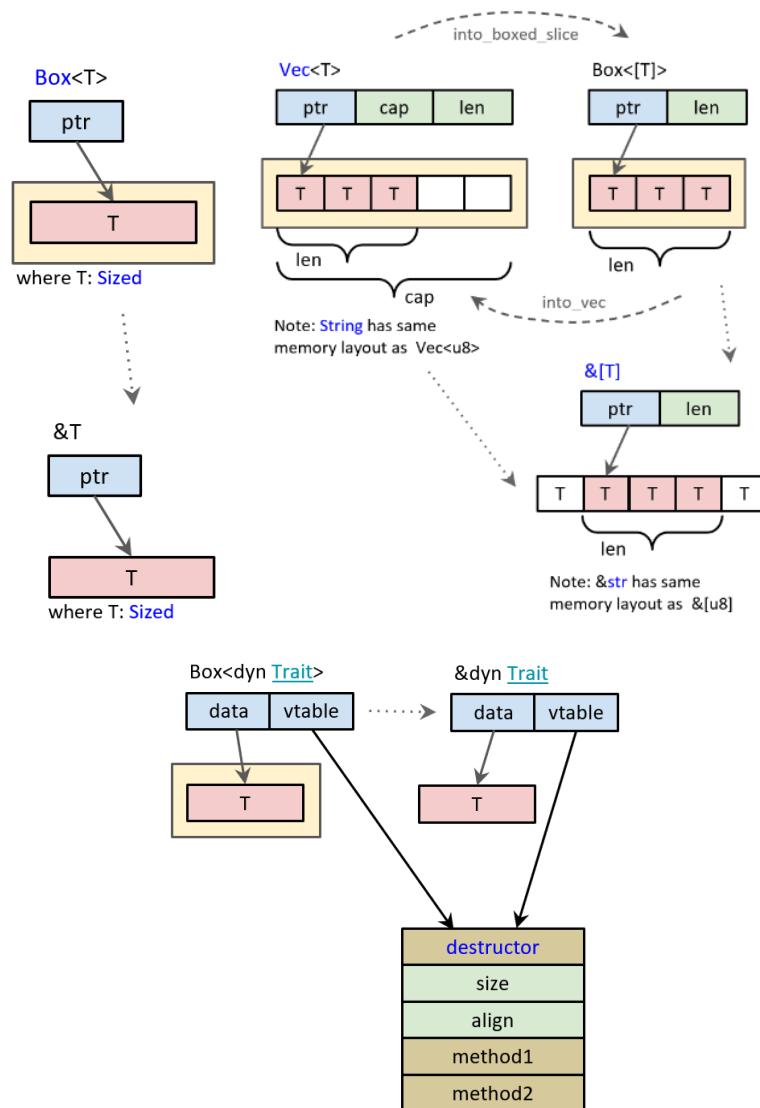
- Questo rende possibile ottenere cicli di vita che si estendono oltre la durata della funzione in cui il dato è stato creato.

Il tipo **T** può avere **dimensione non nota in fase di compilazione** (ovvero non implementare il tratto **Sized**).

- In questo caso, l'oggetto di tipo **Box<T>** si trasforma in un **fat pointer** formato da un puntatore seguito da un intero di dimensione **usize** contenente la lunghezza del dato puntato.
- Analogamente, se **al posto del tipo** concreto **T** si indica un **oggetto-tratto** (**dyn Trait**), si ha un **fat pointer** composto da due puntatori: quello al dato sullo heap e quello a vtable del tratto.

Quindi:

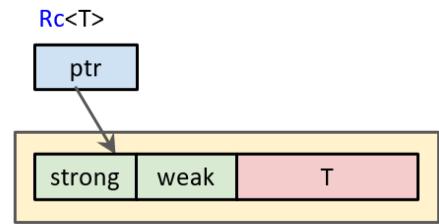
- **T implementa Sized?**
 - Si → dimensione nota → puntatore allo heap.
 - No → dimensione non nota → fat pointer: puntatore allo heap | dim. del dato
- **Oggetto tratto (dyn Trait)**
 - fat pointer: puntatore allo heap | puntatore a vtable



std::rc::Rc<T>

Nelle situazioni in cui occorre disporre di **più possessori di uno stesso dato immutabile**, è possibile usare questo smart pointer.

- Internamente mantiene una **copia del dato e due contatori**:
 - Il primo indica quante copie del puntatore esistono.
 - Il secondo indica quanti riferimenti deboli sono presenti.
- Ogni volta che questo puntatore viene **clonato**, il **primo contatore** viene **incrementato**.
- Quando il puntatore **esce dal proprio scope**, il **contatore** viene **decrementato**: se il primo contatore vale 0, il blocco viene rilasciato.



Rc<T> (Reference Count) si presta a realizzare alberi e grafi aciclici.

- Per motivi di efficienza, l'operatore di incremento e decremento sui campi privati **strong** e **weak** **non è thread-safe**. Per questo motivo, non può essere usato da più di un thread. (vedi std::sync::Arc<T>)

```
pub struct Rc<T: ?Sized> {
    ptr: NonNull<RcBox<T>>,
    phantom: PhantomData<RcBox<T>>,
}
```

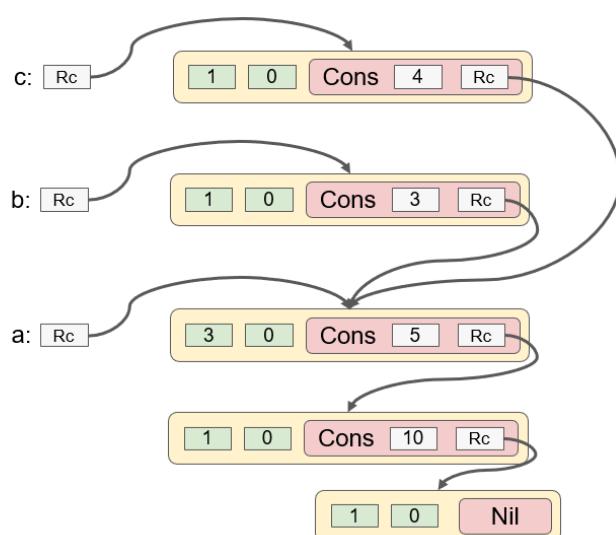
```
struct RcBox<T: ?Sized> {
    strong: Cell<usize>,
    weak: Cell<usize>,
    value: T,
}
```

Nota: PhantomData<T> è uno zero-sized type usato per marcare cosa che agiscono come se possedessero il tipo T (in questo caso RcBox<T>). Questo dice al compilatore che la struttura incapsula un valore di tipo T (RcBox<T>), anche se non lo possiede davvero.

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}
use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(
        Cons(5,
            Rc::new(
                Cons(10, Rc::new(Nil)))));

    let b = Rc::new(
        Cons(3, Rc::clone(&a)));
    let c = Rc::new(
        Cons(4, Rc::clone(&a)));
}
```



Per evitare problemi di omonimia con i metodi contenuti nel dato encapsulato, **tutti i metodi** di Rc sono **dichiarati** con la sintassi

```
pub fn strong_count(this: &Rc<T>) → usize
```

- Chiamando **this** (e non **self**) il parametro che indica l'istanza, non è possibile utilizzare la notazione puntata per invocare i metodi, ma occorre richiamarli nella forma estesa **Rc::<T>::strong_count(&a)**.

std::rc::Weak<T>

Se si costruisce, usando **Rc<T>**, una sequenza circolare di puntatori, la memoria allocata non potrebbe più essere rilasciata.

- La catena dei puntatori terrebbe in vita tutti i blocchi garantendo che il conteggio dei riferimenti valga almeno 1.

È possibile creare una struttura con dipendenza circolari utilizzando il tipo **Weak<T>**.

- Esso è una versione di Rc che contiene un riferimento senza possesso al blocco allocato.

Si crea un valore di tipo **Weak<T>** a partire da un valore di tipo **Rc<T>** con il metodo **Rc::downgrade(&rc)**.

- Se il valore originale è ancora in vita (**strong > 0**), è possibile costruire un nuovo valore di tipo **Rc<T>** invocando il metodo **upgrade()**
- Esso ritorna un valore di tipo **Option<Rc<T>>**

```
pub struct Weak<T: ?Sized> {
    // This is a `NonNull` to allow optimizing the size of this type in enums,
    // but it is not necessarily a valid pointer.
    // `Weak::new` sets this to `usize::MAX` so that it doesn't need
    // to allocate space on the heap. That's not a value a real pointer
    // will ever have because RcBox has alignment at least 2.
    // This is only possible when `T: Sized`; unsized `T` never dangle.
    ptr: NonNull<RcBox<T>>,
}
```

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

let five = Rc::new(5);           // five: 1 | 0 | 5
let weak_five = Rc::downgrade(&five) // weak_five: 1|0|5; five: 1|1|5
let strong_five: Option<Rc<_>> = weak_five.upgrade();
assert!(strong_five.is_some());

drop(strong_five);             // Destroy all strong pointers
drop(five);
assert!(weak_five.upgrade().is_none());
```

`std::cell::Cell<T>`

`Cell<T>`

Il borrow checker garantisce, in fase di compilazione, che dato un valore di tipo `T` ogni momento valgano i seguenti invarianti, mutuamente esclusivi:

`T`

- Non esiste alcun riferimento al valore al di là del suo possessore
- Esistono uno o più riferimenti immutabili (`&T`) - aliasing
- Esiste un solo riferimento mutable (`&mut T`) - mutabilità

Esistono situazioni in cui l'analisi statica eseguita in fase di compilazione non è troppo restrittiva.

- Il modulo `std::cell` offre alcuni contenitori che consentono una mutabilità condivisa e controllata.
- È possibile cioè avere **più riferimenti al valore pur essendo in grado di mutarlo**.
- I tipi offerti possono funzionare solo in contesti **non concorrenti**.

La struct `std::cell::Cell<T>` implementa la mutabilità del dato contenuto al suo interno attraverso metodi che non richiedono la mutabilità del contenitore (gli passi un riferimento non mutabile).

- Si dice che `Cell` implementa un meccanismo di **interior mutability**.

```
use std::cell::Cell;
struct SomeStruct {
    a: u8,
    b: Cell<u8>, // stessa dim in memoria, vincoli diversi per il compiler
}

let my_struct = SomeStruct {
    a: 0,
    b: Cell::new(1),
}
// my_struct.a = 100;
// ERRORE: 'my_struct' è immutabile!

my_struct.b.set(100); // Ok: anche se 'my_struct' è immutabile,
                     //      'b' è una Cell e può essere modificata.
assert_eq!(my_struct.b.get(), 100);
```

- Il metodo `get(&self) → T` restituisce il dato contenuto al suo interno
 - a condizione che `T` implementi il tratto `Copy`
- Il metodo `take(&self) → T` restituisce il valore contenuto, sostituendolo (dentro la cella) con il risultato dell'invocazione di `Default::default()`
 - a condizione che `T` implementi il tratto `Default`

- Il metodo `replace(&self, val: T) → T` sostituisce il valore contenuto nella cella con quello passato come parametro e lo restituisce come risultato.
 - La scrittura diventa uno swap dal vecchio al nuovo valore.
- Il metodo `into_inner(self) → T` consuma la cella e restituisce il valore contenuto.
 - Anche in questo caso, può essere usato con ogni tipo di dato.

```
pub struct Cell<T: ?Sized> {
    value: UnsafeCell<T>,
}
pub struct UnsafeCell<T: ?Sized> {
    value: T,
}
```

`std::cell::RefCell<T>`

`Cell<T>` non consente di creare riferimenti al dato contenuto al suo interno, ma solo di inserire, estrarre o sostituire il valore.

`RefCell<T>`



La struct `std::cell::RefCell<T>` rappresenta un blocco di memoria a cui è possibile accedere attraverso particolari smart pointer che simulano il comportamento di riferimenti condivisi e mutabili.

- Ma la cui compatibilità con le regole del borrow checker è stabilita in fase di esecuzione e non di compilazione.
- Eventuali tentativi di violazione delle regole generano una condizione di panic, comportando la terminazione del thread corrente.

Il metodo `borrow(&self) → Ref<'_, T>` restituisce uno smart pointer che implementa il tratto `Deref<T>`

- Oppure provoca un panic se è già presente un riferimento mutabile.

Il metodo `borrow_mut(&self) → RefMut<'_, T>` restituisce uno smart pointer che implementa il tratto `DerefMut<T>`.

- Oppure provoca un panic se è già presente un riferimento semplice.

```
pub struct RefCell<T: ?Sized> {
    borrow: Cell<BorrowFlag>,
    // Stores the location of the earliest currently active borrow.
    // This gets updated whenever we go from having zero borrows
    // to having a single borrow. When a borrow occurs, this gets included
    // in the generated `BorrowError`/`BorrowMutError`
    #[cfg(feature = "debug_refcell")]
    borrowed_at: Cell<Option<&'static crate::panic::Location<'static>>,
    value: UnsafeCell<T>,
}
```

```

use std::cell::RefCell;
let c = RefCell::new(5);
{
    let m = c.borrow_mut();           // posesso esclusivo se &mut
    assert!(c.try_borrow().is_err()); // modifica c
}
{
    let m = c.borrow();             // posesso multiplo in lettura
    assert!(c.try_borrow().is_ok());
    assert!(*m == 6);
}

```

std::borrow::Cow<'a, B>

Smart pointer che implementa il meccanismo **clone on write**

- Se ci cerca di modificare il dato contenuto, e questo è condiviso, il dato viene clonato: si prende possesso della copia e si effettua la modifica, lasciando l'originale invariato.
- Se il dato che si vuole modificare era già posseduto, non avviene nessuna clonazione e si opera la modifica direttamente.

```

pub enum Cow<'a, B>
where
    B: 'a + ToOwned + ?Sized,
{
    Borrowed(&'a B),
    Owned(<B as ToOwned>::Owned),
}

```

Si istanzia attraverso il metodo **Cow::from()**

- Il compilatore sceglie, in base al tipo di dato fornito, se collocare il valore nella variante **Owned** o **Borrowed**.

Questo smart pointer è quindi utile per:

- ridurre le clonazioni non necessarie (duplicare un valore è costoso)
- aumentare le performance (quando modificare ciò che è memorizzato è raro e quando il valore è spesso utilizzato per scopi read-only)
- salvare memoria

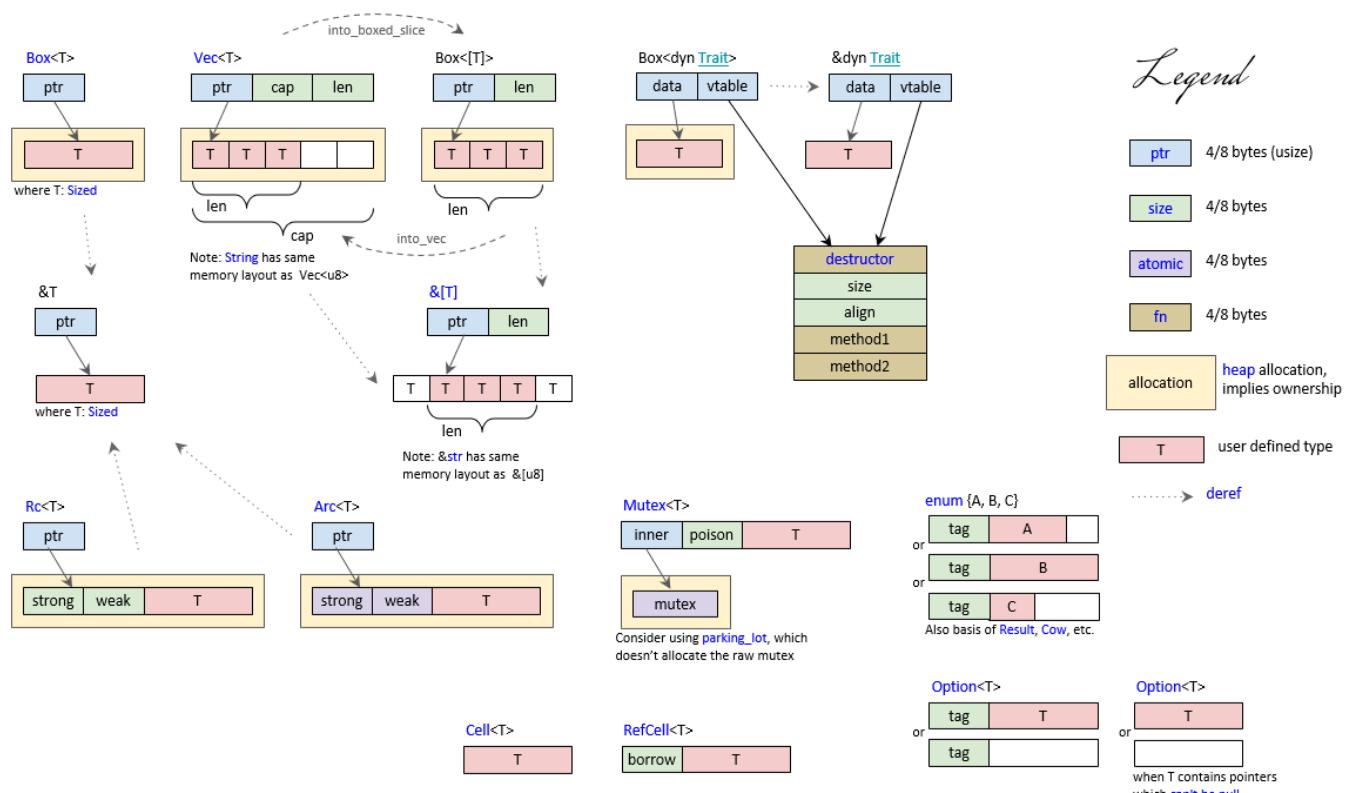
Smart Pointer e metodi

L'argomento `self` di un metodo può anche avere come tipo `Box<Self>`, `Rc<Sel>`, o `Arc<Sel>`.

- In tal caso, il metodo può essere solo invocato a partire dal corrispondente tipo di puntatore.
- L'invocazione del metodo passa la proprietà del puntatore al metodo stesso.

A differenza di quanto accade con i riferimenti, non è disponibile una forma abbreviata per la sintassi di `self`, il cui tipo deve essere dichiarato in modo esplicito, come nel caso dei parametri ordinari.

```
impl Node {
    fn append_to(self: Rc<Self>, parent: &mut Node) {
        parent.children.push(self);
    }
}
```



Rust container cheat sheet, by Raph Levien, Copyright 2017 Google Inc., released under Creative Commons BY, 2017-04-21, version 0.0.4

13 - Concorrenza

13.1 - Programmazione concorrente

Un **programma concorrente** è composto da più di un flusso di esecuzione contemporanei eseguiti in parallelo (se il processore ha più core) e/o alternati nel tempo, sotto il controllo di uno **schedulatore**.

- Alla creazione, il processo dispone di un unico flusso di esecuzione (thread principale).
- Il processo può richiedere allo schedulatore la creazione di altri thread.

Un **thread** rappresenta una **computazione indipendente**

- È basata su un **proprio stack** (pre-allocato alla creazione del thread) che sta sullo stesso spazio di indirizzamento in cui operano i thread del processo.
- La computazione si svolge fino al proprio termine, restituendo un risultato o un errore.
- Se la gestione del thread è demandata al SO, allora si parla di **thread nativi** (quelli che interessano a noi).
- Se il thread è gestito da librerie a livello utente (con supporto parziale del SO), si parla di **green thread** o **fibre**.

Il SO e le librerie di supporto allocano le risorse fisiche necessarie:

- Lo **scheduler** gestisce l'utilizzo dei core disponibili ripartendoli tra i diversi thread in modo non deterministico.
- Tutti i thread creati sono identificati in modo univoco e viene mantenuto l'indicatore del loro stato di esecuzione (per quelli in uso).

I **SO** offrono delle **funzionalità** per gestire il programma e i suoi thread:

- **Creazione di un thread**, indicando la funzione che il thread deve svolgere e la dimensione dello stack.
 - Questa operazione restituisce un **handle opaco** per poter fare riferimento al thread.
- **Identificazione del thread corrente**, tramite un valore univoco (**TID**).
- **Attesa della terminazione di un thread**, a partire dalla sua handle, e accesso al suo stato finale (successo / fallimento)

Nota: tra le funzioni non supportate è presente quella di **cancellazione** di un thread, implementabile solo in modo cooperativo dal thread stesso.

Concorrenza in pratica

Vantaggi della concorrenza:

- Possibilità di **sovrapporre temporalmente** attività di computazione e operazioni di **I/O**.
 - Tramite l'uso di API è possibile arrestare l'esecuzione di un thread finché il dato non è pronto.
- **Riduzione del sovraccarico** dovuto alla comunicazione tra processi.
 - La sincronizzazione e la comunicazione tra processi è molto più onerosa di quella tra thread.
- Possibilità di sfruttare appieno la capacità di **elaborazione** delle CPU **multicore**.
 - Vero parallelismo con più flussi di esecuzione contemporanei.

Svantaggi della concorrenza:

- **Aumento** significativo **della complessità** del programma.
 - Nuove fonti e tipologie di errore e **non determinismo dell'esecuzione**.
- La **memoria non** può più essere pensata come un "**deposito statico**"
 - I dati scritti possono cambiare in conseguenza dell'attività di altri thread.
- I thread devono **coordinare l'accesso** alla memoria tramite costrutti di sincronizzazione.
 - La presenza di cache legate ai singoli core introduce non determinismo nell'ordinamento e nella visibilità delle azioni sulla memoria.

In un processo i thread sono indipendenti tra loro ma solitamente vogliamo farli comunicare. Per farlo servono dei **costrutti di sincronizzazione** che, come contro,

- interferiscono con le ottimizzazioni dei processori per migliorare l'esecuzione
- sono complessi concettualmente

Come vengono gestiti i thread?

In un sistema **single-core**, il processore alterna il proprio ciclo di esecuzione basato sulla successione delle micro-operazioni fetch/decode/execute.

- Il SO può intervenire in questa sequenza grazie ad una interruzione che attiva lo scheduler.
- Questo salva lo stato dei registri in un'area di memoria dedicata alle meta-informationi del thread corrente e li ripristina con il contenuto relativo ad un thread differente (**thread switching**)

Se la CPU ha due o **più core**, il principio è lo stesso:

- Ciascun core procede indipendentemente dagli altri
- Lo scheduler provvede a gestire le attività dei core allocando quelli disponibili ad eseguire un thread, in base alle necessità.

Dato che i thread procedono indipendentemente è necessaria una comunicazione tramite un'area di **memoria condivisa**.

- Nonostante i thread utilizzino lo stesso spazio di indirizzamento e possano, in linea di principio, accedere al dato memorizzato, questa **operazione** risulta **complessa**.
- Per rendere la comunicazione utile praticamente ci si avvale di **pattern di interazione** che definiscono con precisione i ruoli che due o più parti devono svolgere (es. pattern Producer-Consumer).

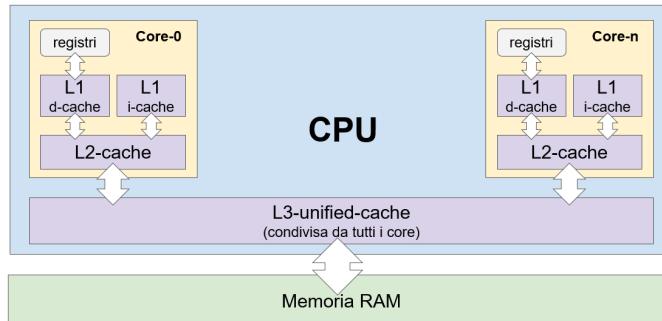
Modello di memoria

Quando un thread legge il contenuto di una locazione di memoria può trovare:

- Il **valore iniziale** contenuto nell'eseguibile che è stato mappato in memoria (es. variabile globale inizializzata)
- Il **valore che questo stesso thread ha precedentemente depositato** all'interno della locazione
- Il **valore** che è stato **depositato da un altro thread**.

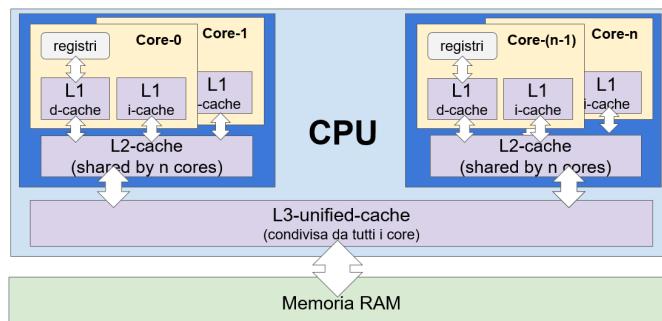
La presenza di **cache hardware** e il possibile riordinamento delle istruzioni da parte della CPU rendono il terzo caso problematico:

- In generale **non è predicibile quale valore venga letto** senza controllare letture e scritture da parte dei thread!
- Occorre usare un **costrutto di sincronizzazione esplicito** per definire l'ordine di esecuzione.



⇒ Singoli core con cache L3 condivisa e L1,L2 private ⇒

↳ Packed cores con memoria L1 privata, L2 condivisa parzialmente, L3 condivisa tra tutti ↳



Problemi aperti

- **Atomicità:** Quali istruzioni devono avere effetti indivisibili?
 - Il problema è principalmente sulle variabili globali e su quelle istanza (cioè sulle variabili condivise tra i thread)
- **Visibilità:** Sotto quali condizioni le scritture compiute da un thread sono visibili da un secondo thread?
- **Ordinamento:** Sotto quali condizioni gli effetti di più operazioni effettuate da un thread possono apparire ad altri thread in ordine differente?

Ciascuna famiglia di processori offre una risposta a questi problemi:

- x86 → modello quasi sequenzialmente consistente con istruzioni di tipo *fence*
- ARM → modello basato su liste di propagazione dei cambiamenti con istruzioni di tipo *barrier* (consentono l'ordinamento causale rispettivamente per il calcolo degli indirizzi, delle istruzioni e dei dati)

Se queste istruzioni offerte non vengono incluse all'interno del codice generato, **non è garantito un ordinamento predicibile delle operazioni di lettura e scrittura di dati condivisi**, in presenza di attività concorrenti.

- Rust annega nelle funzioni di libreria dei tipi dedicati alla concorrenza queste istruzioni in modo da garantire le necessarie proprietà di funzionamento.

Errori

- L'**uso superficiale dei costrutti di sincronizzazione** porta a blocchi passivi o attivi del programma.
- L'**assenza di costrutti di sincronizzazione** porta a risultati imprevedibili.
- Si possono verificare **malfunzionamenti casuali** dovuti al comportamento non deterministico e asincrono dell'esecuzione concorrente (molto difficili da replicare ed eliminare).
- Gli errori possono manifestarsi cambiando la piattaforma di esecuzione o soltanto dopo numerose esecuzioni.

13.2 - Thread

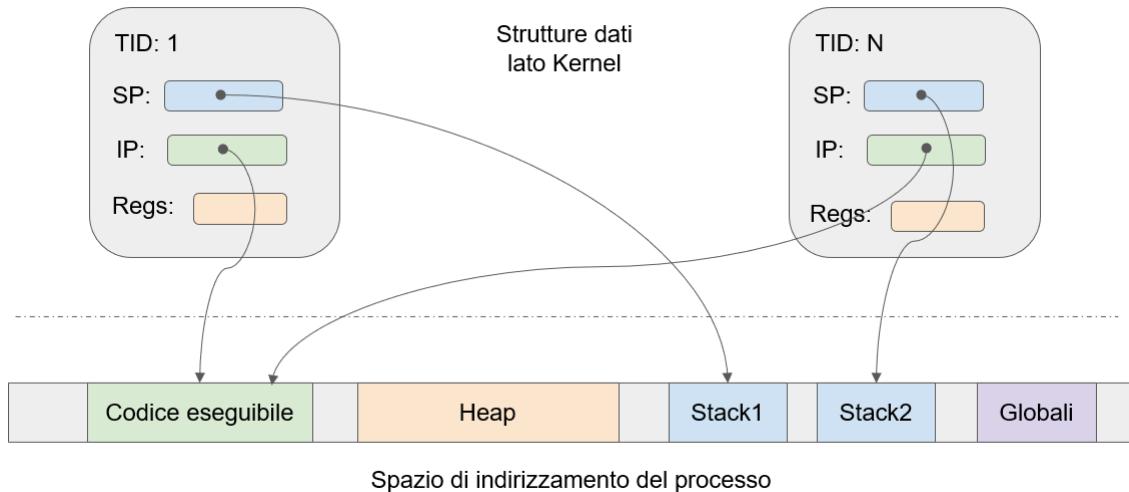
Per ogni thread presente all'interno di un processo, il SO mantiene:

- Un identificativo univoco (**TID** - Thread ID)
- Il suo **stato di esecuzione** (schedulabile, non schedulabile, in esecuzione sul core i, terminato con errore, ...)
- Le **informazioni** necessarie a **salvare/ripristinare lo stato dei registri** interni al processore.

Il SO provvede inoltre ad **allocare**, nello spazio di indirizzamento del processo, lo stack del thread.

Tutti i thread presenti in un processo **condividono**:

- Le variabili globali
- Le costanti
- L'area eseguibile in cui è contenuto il codice
- Lo heap



Esecuzione e non determinismo

L'esecuzione di ogni singolo thread procede secondo le normali regole sequenziali. Se più thread sono in esecuzione, **non è possibile fare assunzioni sulle velocità relative di avanzamento**, a meno di ricorrere a forme esplicite di sincronizzazione e comunicazione.

La sincronizzazione può riguardare

- il raggiungimento di un particolare stato da parte di un thread (abilitando altri a procedere)
- l'esigenza di un thread di eseguire azioni su aree condivise (per impedire ad altri di accedere alle stesse aree)

In alcuni casi, all'informazione logica che abilita/impedisce la prosecuzione di altri thread, si accompagna il trasferimento di informazioni più strutturate che rappresentano l'esito totale o parziale di una computazione avvenuta o la richiesta di elaborazione di ulteriori dati.

Il non determinismo dà origine a **comportamenti inattesi** in un contesto di elaborazione sequenziale.

- **Rust si fa garante che un'intera gamma di possibili errori non possono verificarsi**, grazie ai vincoli di possesso introdotti dal borrow checker

Sincronizzazione

Il fenomeno dell'**inferenza** si verifica quando più thread fanno accesso a uno stesso dato, modificandolo.

- La sua presenza dà origine a malfunzionamenti casuali, molto difficili da identificare.

Se due thread cercano di accedere in lettura/scrittura ad una stessa variabile (senza costrutti di sincronizzazione) si verifica una **corsa critica** (data race).

- In base a condizioni non controllabili dal programmatore il dato memorizzato potrebbe essere quello scritto dal primo thread, quello scritto dal secondo oppure un terzo valore completamente arbitrario.

L'accesso in lettura/scrittura a variabili il cui contenuto è (potenzialmente) scritto da altri thread è soggetto a **diversi vincoli**:

- Deve essere preceduto/seguito da istruzioni che proteggano da dati obsoleti presenti nella cache (fence/barrier).
- Deve avvenire solo quando c'è l'evidenza che il dato non sta venendo modificato da altri.
- Se si sta operando una lettura in attesa di un risultato, si vuole evitare di eseguire cicli continui di polling, che consumano inutilmente cicli di CPU e batteria.

Alcune delle condizioni citate sopra sono garantite da **apposite istruzioni macchina** che dipendono dal processore. Altre richiedono la garanzia di **invarianti a livello di sistema** che può essere fornita solo dal SO che, controllando la schedulazione dei thread, può farsi carico che avvenga o meno una determinata condizione.

- Ne consegue che i **meccanismi di sincronizzazione dipendono dalla coppia processore/sistema operativo**, che collettivamente definiscono l'interfaccia binaria dell'applicazione (**ABI** - Application Binary Interface).

Occorre fare in modo che **non capiti mai** che un thread “operi” su un dato mentre un altro sta già operando sullo stesso oggetto. In particolare, non devono essere visibili **stati transitori** dell'oggetto (servono operazioni atomiche).

- Tutti gli oggetti condivisi mutabili devono godere di questa proprietà (mantengono al proprio interno degli invarianti definiti a livello applicativo)
- Bisogna **impedire che gli invarianti vengano violati**: si effettuano i cambi di stato con metodi che garantiscono la validità degli invarianti prima e dopo l'esecuzione e che blocchino l'accesso concorrente mentre la mutazione è in corso.
- **Si accede allo stato attraverso altri metodi** che controllano che non ci sia una mutazione in corso e che impediscono che essa inizi mentre si sta facendo accesso allo stato condiviso.

In generale è compito del programmatore riconoscere **quando e dove** utilizzare la sincronizzazione. In **Rust**, le limitazioni imposte dal borrow checker sulla esclusività dell'accesso in scrittura, unite all'utilizzo di tratti che modellano il comportamento che un

tipo esibisce quando viene passato da un thread ad un altro, diventano **garanti della correttezza degli accessi**, trasformando errori in esecuzione difficili da identificare in errori di compilazione.

- Questo ha portato a definire questo aspetto di Rust come **fearless concurrency**.

Accesso condiviso: problemi e soluzioni

Problemi:

- **Atomicità:** Se due thread fanno accesso alla stessa struttura dati, rispettivamente in lettura e scrittura non c'è garanzia su quale delle due operazioni sia eseguita per prima.
- **Visibilità:** Se un thread legge un dato che un altro thread sta modificando il valore letto può mettere essere diverso sia dal valore iniziale che da quello finale.
- **Ordinamento:** Se, quando osservato dall'esterno, il comportamento di un singolo thread appare indistinguibile dopo una modifica alla sequenza delle istruzioni, sia il compilatore che la CPU possono invertire l'ordine di esecuzione delle singole istruzioni.

Soluzioni possibili:

- **Atomic**
 - Istruzioni apposite offerte dai processori per garantire operazioni di tipo Read-Modify-Write di tipo atomico, cioè non interrompibili e non osservabili nei loro stati intermedi.
 - Queste operazioni sono limitate a tipi semplici (booleani, interi, puntatori) e sono esposte dalle librerie standard di Rust attraverso opportune astrazioni, che incapsulano l'utilizzo di barriere di memoria.
- **Mutex**
 - Permette di estendere le garanzie di atomicità e dipendenza causale a strutture dati più complesse, estendendo il principio di mutua esclusione a thread differenti.
 - Un mutex può essere libero o posseduto da un singolo thread. Se un secondo thread cerca di ottenerne il possesso mentre è in uso da un altro thread, rimane in attesa (senza consumare cicli di CPU) fino a che esso non viene rilasciato.
- **Condition variable**
 - In alcuni casi occorre attendere che si verifichi una condizione più complessa del semplice rilascio del mutex.
 - Una condition variable permette di realizzare tale attesa a condizione che il thread che causa l'avverarsi della condizione si occupi di segnalarlo, generando una notifica tramite appositi metodi.
 - Può essere usata solo in coppia con un mutex.

Uso dei thread

Le API del SO permettono la gestione del ciclo di vita dei thread:

- Creazione e terminazione
- Meccanismi di sincronizzazione
- Aree private di memoria

In Rust viene ripresa la standardizzazione effettuata dal C++ nella creazione gestione dei thread e viene implementata nella libreria standard, con maggiore attenzione alla gestione del fallimento.

In Rust, si crea un thread nativo attraverso la funzione `std::thread::spawn(..)`

- **Accetta** una funzione **lambda** che rappresenta la computazione che il thread deve svolgere
- **Ritorna** una struct di tipo `std::thread::JoinHandle<T>`, dove T rappresenta il tipo restituito dalla computazione del thread (il tipo ritornato dalla lambda)

Per sapere quando la computazione del thread è terminata e quale valore abbia prodotto, occorre utilizzare il metodo `join()` offerto dalla handle.

- Tale metodo restituisce un'enumerazione di tipo `std::thread::Result` che contiene nella `Ok` il valore finale o nella `Err` il valore passato alla `panic!`.

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
{
    Builder::new().spawn(f).expect("failed to spawn thread")
}

pub struct JoinHandle<T>(JoinInner<'static, T>);
```

```
/// Inner representation for JoinHandle
struct JoinInner<'scope, T> {
    native: imp::Thread,
    thread: Thread,
    packet: Arc<Packet<'scope, T>>,
}

impl<'scope, T> JoinInner<'scope, T> {
    fn join(mut self) -> Result<T> {
        self.native.join();
        Arc::get_mut(&mut self.packet).unwrap().result.get_mut().take().unwrap()
    }
}
```

Nota: non si crea nessun rapporto di parentela tra il thread creatore (quello che invoca la `spawn(..)`) e il thread creato, né occorre che l'uno sopravviva all'altro.

- Quando l'handle di un thread esce dallo scope e viene rilasciato, non c'è più modo di avere notizie dirette sull'esito del thread creato, che acquisisce lo stato di **detached**.

Esempio

```
use std::thread

let thread_join_handle = thread::spawn(move || {
    // move trasferisce alla funzione
    // il possesso di quanto catturato (movimento)
    ...
})
...
match thread_join_handle.join() {
    Ok(res) => {...}, // valore computato dal thread
    Err(err) => {...}, // se il thread scatena un panic!(..)
}
```

Un thread può essere **configurato nel dettaglio** prima della sua esecuzione tramite la struct **std::thread::Builder** che permette di assegnare al thread un nome e di definire la dimensione dello stack da associare al thread.

- Il metodo `spawn(..)` consuma l'oggetto `Builder`, crea il thread corrispondente e restituisce un enum di tipo `io::Result<JoinHandle>`.

```
pub struct Builder {
    // A name for the thread-to-be, for identification in panic messages
    name: Option<String>,
    // The size of the stack for the spawned thread in bytes
    stack_size: Option<usize>,
}
```

Esempio

```
use std::thread;

let builder = thread::Builder::new()
    .name("t1".into())      // Utile per debuggare
    .stack_size(100_000);

let handler = builder.spawn(|| {...}).unwrap();
handler.join.unwrap();
```

I tratti della concorrenza

Allo scopo di garantire la correttezza degli accessi alla memoria e l'assenza di comportamenti non definiti, Rust introduce due **tratti marcatori** (senza metodi), il cui scopo è fornire indicazioni sul comportamento di un tipo in un contesto multi-thread.

- Il tratto `std::marker::Send` indica che il tipo può essere passato (ceduto) in modo safe per movimento tra thread.
 - Viene applicato automaticamente a tutti i tipi che possono essere trasferiti in sicurezza da un thread ad un altro, ovvero in grado di garantire che non è possibile avere accessi al loro contenuto contemporaneamente.
- Il tratto `std::marker::Sync` indica che il tipo può essere passato (condiviso) in modo safe per riferimento tra thread.
 - Viene applicato automaticamente a tutti i tipi T tali che $\&T$ risulta avere il tratto Send, ovvero che possono essere condivisi in sicurezza tra thread differenti, senza creare problemi di comportamenti non definiti

Nota: **Puntatori e riferimenti non hanno il tratto Send.**

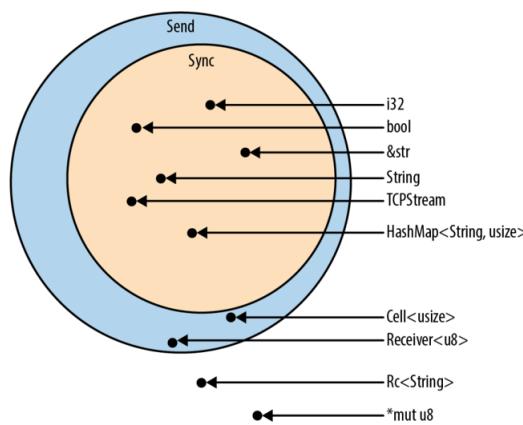
- L'esecuzione indipendente dei thread non permette al borrow checker di fornire le proprie garanzie di correttezza.

`pub unsafe auto trait Send { }`

- Se un tipo dispone del tratto Send, è lecito passarlo per valore (movimento) ad altri thread.
- L'uso del movimento (o della copia) garantisce la non contemporaneità degli accessi.
- I tipi composti (struct, tuple, enum, array) godono del tratto Send se tutti i loro campi lo possiedono (è comunque possibile forzare l'assegnazione/rimozione del tratto solo all'interno di un blocco unsafe).

`pub unsafe auto trait Sync { }`

- Se un tipo dispone del tratto Sync, è lecito pensarlo come riferimento non mutabile ad altri thread.
- I tipi che implementano una mutabilità interna (come `Cell` e `RefCell`) non dispongono di questo tratto.



È possibile **creare thread solo se i dati** catturati dalla funzione **lambda** che ne descrive la computazione e il suo tipo di ritorno **hanno il tratto Send**, altrimenti il compilatore genera un errore di compilazione.

```
fn main() {
    let data1 = Rc::new(1);
    let data2 = data1.clone();
    println!("t0: {}", *data1);

    let jh = spawn(move || {
        println!("t1: {}", *data2);
    });
    jh.join().unwrap();
}
```

```
error[E0277]: `Rc<i32>` cannot be sent
between threads safely
--> src/main.rs:7:12
7  |      let jh = spawn(move || {
|      |          ^^^^^^ `Rc<i32>` cannot be
|      |          sent between threads safely
|      |          the trait `Send` is not implemented for
|      |          `Rc<i32>`
```

Modelli di concorrenza

La libreria standard di Rust supporta **due modelli base** per la realizzazione di programmi concorrenti:

1. La condivisione di dati basata su **sincronizzazione degli accessi ad una struttura dati condivisa**, a cui tutti i thread interessati possono accedere in lettura e scrittura (mutex e condition variables).
2. La condivisione di dati basata sullo **scambio di messaggi** che prevede uno o più mittenti ed un solo destinatario (canali).

Esistono anche librerie esterne che supportano altri modelli:

- actix → modello degli attori
- rayon → modello work stealing
- **crossbeam** → permette la condivisione di dati memorizzati nello stack del thread genitore con i thread figli.

13.3 - Mutex e puntatori concorrenti

Per poter condividere dati modificabili tra thread differenti, serve un meccanismo che permetta, ad un solo thread alla volta, di acquisire il permesso di modifica, bloccando altri thread che richiedono l'accesso alla stessa risorsa.

Il metodo più semplice per ottenere questo comportamento è attraverso l'uso di un **mutex** (MUTual EXclusion lock).

- Questo tipo di costrutto viene offerto dai sistemi operativi e sono riesportati in modo indipendente dalla piattaforma delle librerie di Rust.

Gli oggetti nativi offerti dai SO offrono almeno due metodi: **lock()** e **unlock()**.

- Invocando **lock()**, un thread **richiede il possesso del mutex**: se questo non può essere garantito al momento, l'invocazione si blocca fino a che il mutex non è stato rilasciato dall'attuale possessore.
- È lecito invocare **unlock()** solo se il thread che lo esegue è l'attuale possessore del mutex.

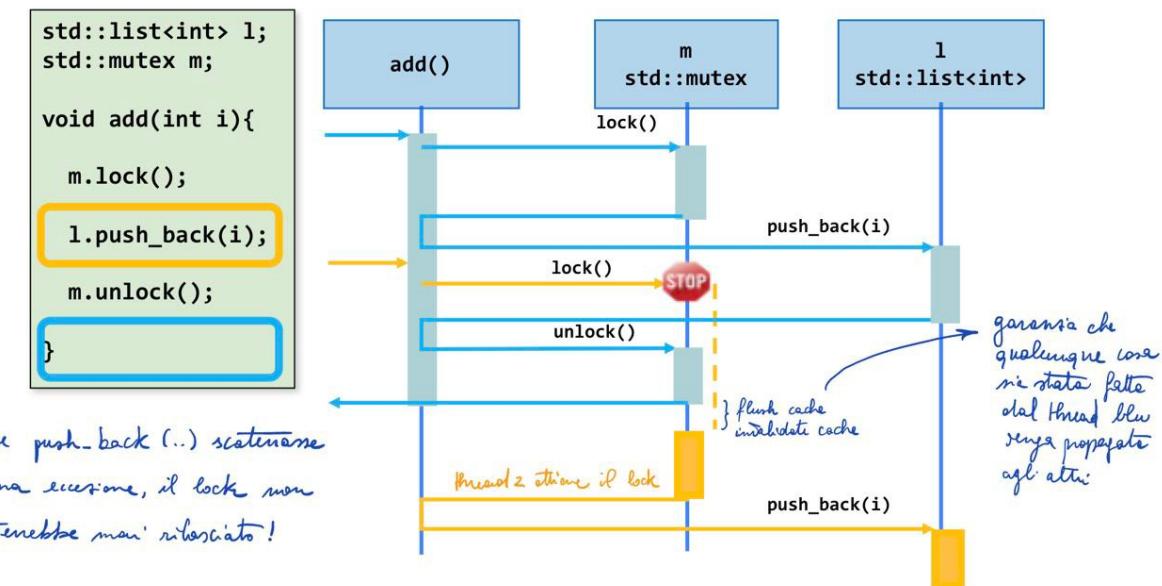
Entrambi i metodi **lock()** e **unlock()** includono una **barriera di memoria**

- Garantisce la visibilità delle operazioni eseguite fino a quel punto dagli altri thread.
- Permette di imporre la dipendenza causale (una istruzione viene prima di un'altra e né il compilatore né la CPU possono cambiarne l'ordine).

Sul piano pratico, se si vuole condividere una risorsa tra thread bisogna proteggerla con un mutex che deve sempre essere acquisito prima di fare accesso alla risorsa.

Nelle astrazioni di base offerte dai SO e in C++ non c'è una corrispondenza sintattica tra un mutex e la struttura dati che questo protegge.

- Un mutex potrebbe proteggere molte strutture diverse, ma questo riduce il grado di parallelismo complessivo del programma. Meglio avere un mutex per ogni risorsa.



Rilasciare i mutex

Se un thread che è in possesso di un mutex termina senza rilasciare il mutex (errore o altro), si crea un problema.

- I sistemi operativi tendono a liberare il mutex, ma che stato hanno le risorse che il mutex protegge?

Per ovviare a questo problema, viene sfruttato il paradigma RAI, che garantisce che il lock sia rilasciato automaticamente nel momento in cui l'oggetto che contiene il mutex viene distrutto.

- In C++, la classe `std::lock_guard` incapsula un `std::mutex`: il costruttore lo acquisisce e il distruttore lo rilascia.
- Problema (C++): dall'uso del pattern non emerge quali metodi della classe che contiene il mutex debbano essere sincronizzati, né impedisce che venga scritto del codice che fa accesso ai dati condivisi senza possedere il lock. Diventa tutto onere del programmatore gestire la correttezza del codice.

Mutex in Rust

L'accesso ad uno stato condiviso in Rust richiede l'utilizzo di **due blocchi** in cascata:

- Il primo permette il possesso multiplo di una struttura dati in sola lettura da parte di più thread, realizzato mediante il costrutto `std::sync::Arc<T>`.
- Il secondo consente l'acquisizione in lettura/scrittura della struttura dati, realizzato alternativamente mediante il costrutto `std::sync::Mutex<T>`, con `std::sync::RwLock<T>`, oppure tramite **tipi atomici**.

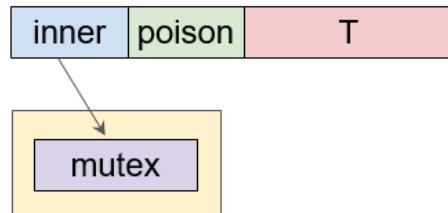
Questa combinazione di blocchi, che prende spunto dal pattern RAII, permette di **rendere esplicito** nella struttura del codice e nei pattern di accesso ai dati **cosa sia condiviso e impedisce** di fatto l'**accesso senza** il corretto possesso del **lock** relativo.

- Questo permette al compilatore di bloccare ogni tentativo di accesso non conforme.

`std::sync::Mutex<T>`

- Un oggetto di tipo `Mutex` incapsula un dato di tipo `T` oltre al riferimento ad un mutex nativo del sistema operativo.
 - L'unico modo per accedere al dato è invocare il metodo `lock()` che restituisce un oggetto di tipo `LockResult<MutexGuard<T>>` e resta bloccato fino a che non è stato possibile acquisire il mutex nativo.
 - Se l'ultimo thread che ha acquisito il mutex fosse terminato prima di averlo rilasciato, il mutex si troverebbe nello **stato avvelenato**, e la risposta conterrebbe un errore.

```
pub struct Mutex<T: ?Sized> {
    inner: sys::Mutex,
    poison: poison::Flag,
    data: UnsafeCell<T>,
}
```



- Se il metodo `lock()` ha successo, la risposta contiene un `MutexGuard<T>`:
 - Questo oggetto implementa il trait `Deref<T>` e si comporta come uno smart-pointer (può essere dereferenziato per ottenere un riferimento mutabile al dato `T`).
 - Quando il `MutexGuard<T>` esce dallo scope, il mutex nativo viene rilasciato, permettendo ad altri thread di chiederne il possesso.
 - A tutti gli effetti il `MutexGuard<T>` implementa il pattern RAII, ma (per come è costruito) è necessariamente disponibile solo se si possiede il mutex (vantaggio: il check avviene in fase di compilazione).

```

pub fn lock(&self) -> LockResult<MutexGuard<'_, T>> {
    unsafe {
        self.inner.lock();
        MutexGuard::new(self)
    }
}

pub struct MutexGuard<'a, T: ?Sized + 'a> {
    lock: &'a Mutex<T>,
    poison: poison::Guard,
}

```

`std::sync::Arc<T>`:

- Un oggetto di tipo `Mutex` può avere un solo possessore. Per superare questo vincolo lo si incapsula all'interno un oggetto di tipo `std::sync::Arc<T>`.
- `Arc<T>` permette di **condividere il possesso di un dato**, allocandolo nello heap e mantenendo un conteggio dei riferimenti esistenti di tipo thread-safe.
 - È possibile duplicare un oggetto di questo tipo attraverso il metodo `clone()`. Questo metodo duplica il puntatore al blocco sullo heap, avendo cura di incrementare atomicamente il contatore dei riferimenti associati al dato.
 - Il dato clonato viene ceduto ad un thread specificando la parola-chiave `move` di fronte alla funzione lambda che ne descrive la computazione.

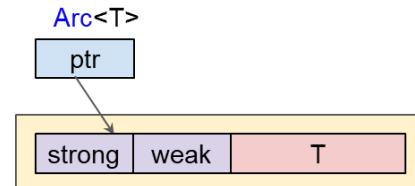
```

pub struct Arc<T: ?Sized> {
    ptr: NonNull<ArcInner<T>>,
    phantom: PhantomData<ArcInner<T>>,
}

struct ArcInner<T: ?Sized> {
    strong: AtomicUsize,
    weak: AtomicUsize,
    data: T,
}

```

PhantomData è uno zero-sized type usato per marcare cose che agiscono come se possedessero `T`.



```

let shared_data = Arc::new(Mutex::new(Vec::new()));
let mut threads = vec![];

for (i in 1..10) {
    let mut data = shared_data.clone(); // duplicazione del possesso
    threads.push(thread::spawn(move || { // data è ceduto al thread
        let mut v = data.lock().unwrap(); // v è di tipo MutexGuard<T>
        v.push(i); // Quando v esce dallo scope,
    }));
}
for t in threads { t.join().unwrap(); } // v contiene i numeri da 1 a 9
// n.b. l'ordine non è noto

```

Se un thread viene creato mediante la primitiva `std::thread::spawn(..)`, il compilatore non può fare assunzioni sulla sua durata.

- Di conseguenza, **impedisce l'utilizzo di riferimenti condivisi tra la funzione lambda del thread e la funzione all'interno della quale il thread viene creato.**

La libreria standard permette di creare un thread anche tramite la funzione

`std::thread::scope(|s| std::thread::Scope| {...})`

- Accetta come parametro una lambda il cui compito è racchiudere l'intero ciclo di vita dei thread creati al suo interno.
- Il parametro s passato a tale funzione offre il metodo `spawn()` mediante il quale è possibile creare nuovi thread.
- Terminata l'esecuzione della lambda, la funzione `scope(...)` non ritorna fino a che tutti i thread creati al suo interno non sono terminati.
 - Questo permette al borrow checker di considerare corretto l'uso di riferimenti a variabili locali, dato che la loro durata sarà almeno pari a quella della funzione `scope(..)` e, di conseguenza, a quella dei thread creati al suo interno.

```

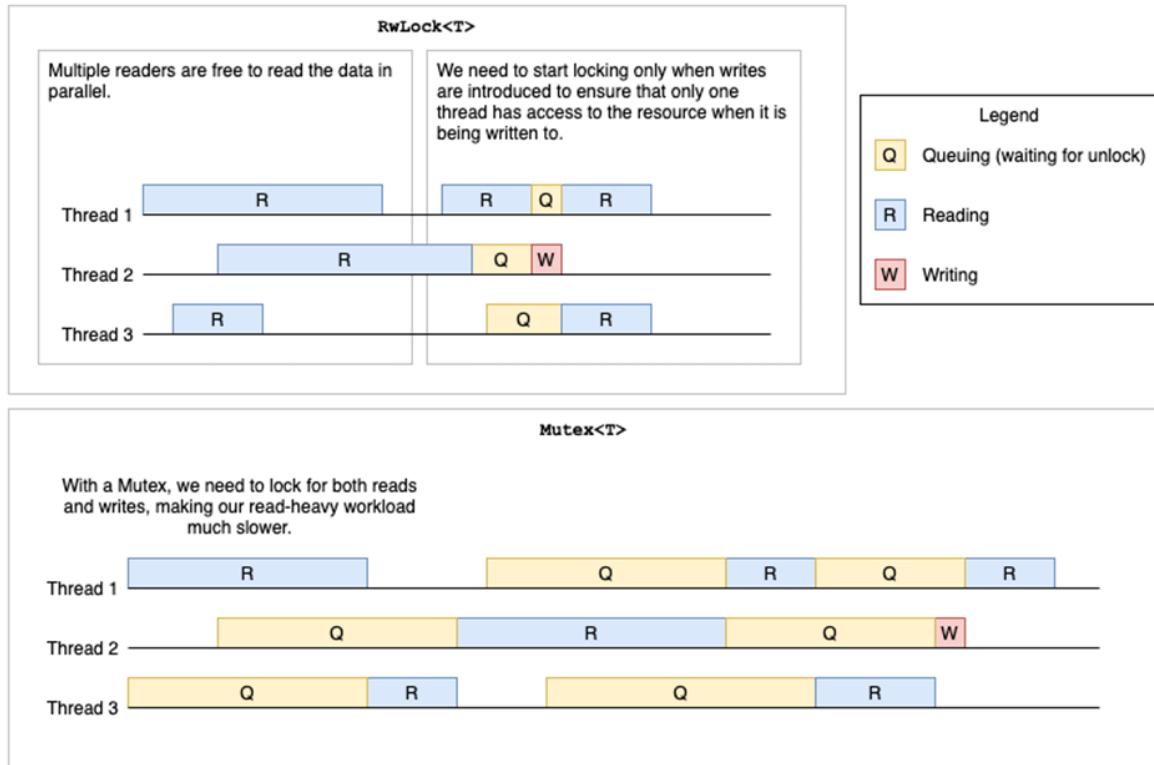
let mut v = vec![1, 2, 3];
let mut x = 0;
thread::scope(|s| {
    s.spawn(|| { // è lecito creare un riferimento a v
        println!("length: {}", v.len());
    });
    s.spawn(|| { // anche qui viene catturato
        for n in &v { println!("{}", n); }
        x += v[0] + v[2]; // x è catturato come &mut
    });
});
// Solo quando entrambi i thread saranno terminati si proseguirà
v.push(4);
assert_eq!(x, v.len());

```

`std::sync::RwLock<T>`

- Se gli accessi in lettura e in scrittura sono sbilanciati, può essere conveniente sostituire, alla struct `Mutex<T>`, la struct `RwLock<T>`.
- Questa struct offre il metodo `read()` per accedere, in modo condiviso, in lettura ed il metodo `write()` per accedere in modo esclusivo.
- Dal grafico sotto si può notare come uno stesso processo duri più tempo usando un `Mutex<T>` piuttosto che un `RwLock<T>`.

```
pub struct RwLock<T: ?Sized> {
    inner: sys::RwLock,
    poison: poison::Flag,
    data: UnsafeCell<T>,
}
```



- I metodi `read()` e `write()` restituiscono rispettivamente oggetti di tipo `LockResult<RwLockReadGuard>` e `LockResult<RwLockWriteGuard>`
 - Il risultato contiene un errore se il lock è avvelenato. Questo capita quando un thread che lo possedeva in lettura è terminato senza averlo rilasciato o cercando di acquisirlo ulteriormente.
 - Entrambi gli oggetti di guardia implementano il pattern RAII, rilasciando il lock nel momento in cui sono distrutti.

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

let lock = Arc::new(RwLock::new(1));
let c_lock = Arc::clone(&lock);

let n = lock.read().unwrap();
assert_eq!(*n, 1);
thread::spawn(move || {
    let r = c_lock.read();
    assert!(r.is_ok());
}).join().unwrap();
```

Tipi atomici

Il modulo `std::sync::atomic` mette a disposizione alcune strutture dati che costituiscono primitive di comunicazione tra thread basate sul principio della **memoria condivisa**

- Esso offre versioni atomiche di valori booleani, numeri interi con e senza segno e puntatori nativi.
- Ciascun tipo è assegnato ad operazioni che permettono di sincronizzare gli aggiornamenti di tali valori tra thread differenti.

Accanto alle operazioni di lettura e scrittura con associata batteria di memoria, questi tipi offrono funzionalità di tipo **Read-Modify-Write**.

- `swap(..)`, `compare_exchange(..)`, `fetch_add(..)`, `fetch_update(..)`
- Ciascuna di queste operazioni riceve come parametro esplicito il tipo di barriera di memoria da applicare per la fase di lettura e per quella di scrittura.

Sebbene siano tutti thread-safe, in quanto implementano il tratto Sync, **non offrono meccanismi di condivisione esplicita**.

- Come tutti i valori in Rust, sono soggetti alla regola del possessore unico.
- Per permettere a più thread di accedere al loro valore, è comune **incapsularli** dentro un `Arc<T>` oppure dichiararli come variabili globali, attraverso la parola chiave `static`.

In modo analogo a `Cell<T>`, implementano il meccanismo di **mutabilità interna**.

- Poichè le operazioni di modifica sono garantite essere thread-safe, i metodi che ne modificano il contenuto richiedono solo un accesso condiviso (`&self`) e non un accesso esclusivo (`&mut self`)

```
use std::sync::Arc;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::{hint, thread};

fn main() {
    let spinlock = Arc::new(AtomicUsize::new(1));
    let spinlock_clone = Arc::clone(&spinlock);
    let thread = thread::spawn(move || {
        spinlock_clone.store(0, Ordering::Release);
    });

    // Attendi
    while spinlock.load(Ordering::Acquire) != 0 {
        hint::spin_loop();
    }
    thread.join().unwrap();
}
```

Dipendenze circolari

Analogamente a quanto succede con gli smart pointer di tipo `Rc<T>`, anche nel caso di `Arc<T>` la creazione di catene circolari impedisce il rilascio delle strutture.

- Per questo motivo è disponibile la struct `std::sync::Weak<T>` che permette di realizzare dipendenze circolari con riferimenti che non partecipano al conteggio, garantendo così la possibilità di rilascio.
- Per fare accesso al dato puntato, occorre invocare il metodo `upgrade()`, che restituisce un valore di tipo `Option<Arc<T>>`.

Si crea un oggetto di tipo `Weak<T>` a partire da un riferimento di tipo `Arc<T>` invocando su quest'ultimo il metodo `downgrade()`.

13.4 - Condition variables

Serve un meccanismo che permetta ad un thread di aspettare uno o più risultati intermedi prodotti da altri thread.

- L'attesa non deve consumare risorse e deve terminare non appena un dato è disponibile. Quindi il polling va evitato per due motivi:
 - Consuma capacità di calcolo e batteria in cicli inutili
 - Introduce una latenza tra il momento in cui il dato è disponibile e il momento in cui il secondo thread si sblocca.
- Per garantire l'assenza di interferenze tra thread, la presenza di dati condivisi richiede come minimo l'utilizzo di un mutex.

Per gestire queste situazioni, i SO offrono il concetto di **condition variable**, strutture dati di sincronizzazione che permettono di **bloccare l'esecuzione di un thread**, senza consumo di CPU, **nell'attesa che qualcosa succeda**.

- L'uso di una cv è basata sulla cooperazione all'interno del sistema: se un thread si sospende in attesa di una condizione, è necessario che tutti i thread che eseguono azioni che potrebbero provocare il verificarsi della condizione si facciano carico di inviare una notifica alla cv.

Il pattern di utilizzo prevede che esista una **espressione booleana** il cui valore possa essere usato **per determinare se occorre accedere o meno**.

- La valutazione di questa espressione avviene mentre si possiede un mutex, per garantire l'assenza di corse critiche.
- In Rust, questo vuol dire che le **variabili** che consentono la valutazione della condizione sono **incapsulate nel mutex**.

Ogni condition variable deve essere usata in coppia con un singolo mutex. Eventuali tentativi di usare mutex diversi per una stessa cv possono causare un errore a runtime.

Rust offre la struct `std::sync::Condvar`

- La struttura dei suoi metodi facilita il collegamento con il dato protetto dal mutex, rendendo più naturale il suo utilizzo.

```
pub struct Condvar {  
    inner: sys::Condvar,  
}
```

```
pub struct Condvar {  
    inner: Box<sys::Condvar>,  
    mutex: AtomicUsize,  
}
```

Metodi principali

- `pub fn new() → Condvar`
 - Crea una nuova istanza

```
pub const fn new() -> Condvar {  
    Condvar { inner: sys::Condvar::new() }  
}
```

```
pub fn new() -> Condvar {  
    let mut c = Condvar {  
        inner: box sys::Condvar::new(),  
        mutex: AtomicUsize::new(0),  
    };  
    unsafe {  
        c.inner.init();  
    }  
    c  
}
```

- `pub fn wait<'a, T>(&self, guard: MutexGuard<'a, T>) → LockResult<MutexGuard<'a, T>`
 - Sospende il thread corrente fino alla ricezione di una notifica: durante la sospensione, rilascia il lock; al ricevere della notifica, riacquisisce il lock e restituisce una nuova guardia.

```
pub fn wait<'a, T>(&self, guard: MutexGuard<'a, T>) -> LockResult<MutexGuard<'a, T>> {  
    let poisoned = unsafe {  
        let lock = mutex::guard_lock(&guard);  
        self.inner.wait(lock);  
        mutex::guard_poison(&guard).get()  
    };  
    if poisoned { Err(PoisonError::new(guard)) } else { Ok(guard) }  
}
```

```

pub fn wait<'a, T>(&self, guard: MutexGuard<'a, T>)
    -> LockResult<MutexGuard<'a, T>> {
    let poisoned = unsafe {
        let lock = mutex::guard_lock(&guard);
        self.verify(lock);
        self.inner.wait(lock);
        mutex::guard_poison(&guard).get()
    };
    if poisoned {
        Err(PoisonError::new(guard))
    } else {
        Ok(guard)
    }
}

```

- **pub fn notify_one(&self)**

- Sveglia un thread a caso tra quelli in attesa sulla condition variable.

```

pub fn notify_one(&self) {
    self.inner.notify_one()
}

```

```

pub fn notify_one(&self) {
    unsafe { self.inner.notify_one() }
}

```

- **pub fn notify_all(&self)**

- Sveglia tutti i thread in attesa sulla condition variable, che usciranno, uno alla volta, dal metodo wait possedendo il lock

```

pub fn notify_all(&self) {
    self.inner.notify_all()
}

```

```

pub fn notify_all(&self) {
    unsafe { self.inner.notify_all() }
}

```

NB: Quando mi sveglio dopo una wait devo verificare se:

- a. sono uscito perché dovevo uscire (ho il valore che aspettavo)
- b. mi hanno svegliato ma la notifica era “spuria” (non ho il valore ma mi hanno svegliato)

```

let pair = Arc::new((Mutex::new(false), Condvar::new()));
let pair2 = Arc::clone(&pair);

// Inside of our lock, spawn a new thread, and then wait for it to start.
thread::spawn(move|| {
    let (lock, cvar) = &*pair2;
    let mut started = lock.lock().unwrap();
    *started = true;
    // We notify the condvar that the value has changed.
    cvar.notify_one();
});

// Wait for the thread to start up.
let (lock, cvar) = &*pair;
let mut started = lock.lock().unwrap();
while !*started {
    started = cvar.wait(started).unwrap();
}

```

Meccanismo di funzionamento

Una condition variable mantiene una collezione di thread in attesa che si verifichi la condizione attesa.

- Inizialmente la lista è vuota.
- Quando un thread esegue il metodo `wait(..)`, viene sospeso e aggiunto alla lista.

Quando sono eseguiti i metodi `notify_one()` o `notify_all()`, uno o tutti i thread presenti nella collezione sono risvegliati.

- Si basa sul SO per sospendere/risvegliare i thread
- La presenza di un **unico lock** fa sì che, se più thread ricevono la notifica, il **risveglio sia progressivo**. Non appena un thread rilascia il lock, un altro può acquisirlo e proseguire.
- La relazione tra l'evento e la notifica è solo nella testa del programmatore. Per questo, si rende esplicito l'evento che si è verificato appoggiandosi ad una o più variabili condivise (i booleani) sotto il controllo del mutex.

Notifiche spurie e notifiche perse

È possibile che un thread in attesa su una condition variable sia risvegliato in assenza di un'esplicita notifica (problema delle cosiddette **notifiche spurie**)

- Questo è conseguenza di come è stato sviluppato il kernel. È importante verificare sempre al risveglio che ci siano le condizioni corrette per proseguire.

Per semplificare la verifica, esiste una versione del metodo di attesa che riceve come argomento una funzione volta a valutare il predicato richiesto:

- `pub fn wait_while<'a, T, F>(&self, guard: MutexGuard<'a, T>, condition: F) → LockResult<MutexGuard<'a, T>>`
`where F: FnMut(&mut T) → bool`
 - Al risveglio, acquisisce il lock e valuta la funzione `condition`: se questa restituisce `true`, si riaddormenta, altrimenti esce dall'attesa

```
pub fn wait_while<'a, T, F>(&self, mut guard: MutexGuard<'a, T>, mut condition: F, ) -> LockResult<MutexGuard<'a, T>> where F: FnMut(&mut T) -> bool, {
    while condition(&mut *guard) {
        guard = self.wait(guard)?;
    }
    Ok(guard)
}
```

Analogamente, se un thread ha eseguito una qualche azione che può abilitare la prosecuzione di un altro thread, ed invoca il metodo `notify_one()` / `notify_all()` per segnalare tale fatto, **è possibile che la notifica vada persa**.

- Succede se l'altro thread non ha ancora eseguito la corrispondente istruzione attesa.

Per questo motivo, occorre sempre racchiudere l'istruzione di attesa in un ciclo che verifica se occorra o meno addormentarsi e, al risveglio, se ci siano le condizioni o meno per continuare a dormire.

- In entrambi i casi, il metodo `wait_while(..)` protegge.

Attesa temporizzata

Altri metodi permettono di **limitare il tempo massimo di attesa**, permettendo al thread di risvegliarsi anche in assenza del verificarsi della condizione.

- `pub fn wait_timeout<'a, T>(&self, guard: MutexGuard<'a, T>, dur: Duration) → LockResult<(MutexGuard<'a, T>, WaitTimeoutResult)>`
 - Attende per un tempo massimo pari a `dur`

```
pub fn wait_timeout<'a, T>(&self, guard: MutexGuard<'a, T>, dur: Duration, ) -> LockResult<(MutexGuard<'a, T>, WaitTimeoutResult)> {
    let (poisoned, result) = unsafe {
        let lock = mutex::guard_lock(&guard);
        let success = self.inner.wait_timeout(lock, dur);
        (mutex::guard_poison(&guard).get(), WaitTimeoutResult(!success))
    };
    if poisoned { Err(PoisonError::new((guard, result))) } else { Ok((guard, result)) }
}
```

- `pub fn wait_timeout_while<'a, T, F>(
 &self,
 guard: MutexGuard<'a, T>,
 dur: Duration,
 condition: F
) -> LockResult<(MutexGuard<'a, T>, WaitTimeoutResult)>`
where `F: FnMut(&mut T) -> bool`
 - Attende per un tempo massimo pari a `dur`; eventuali notifiche ricevute portano a riaddormentarsi se la funzione `condition` restituisce `false`.

```
pub fn wait_timeout_while<'a, T, F>(
    &self,
    mut guard: MutexGuard<'a, T>,
    dur: Duration,
    mut condition: F,
) -> LockResult<(MutexGuard<'a, T>, WaitTimeoutResult)>
where
    F: FnMut(&mut T) -> bool,
{
    let start = Instant::now();
    loop {
        if !condition(&mut *guard) {
            return Ok((guard, WaitTimeoutResult(false)));
        }
        let timeout = match dur.checked_sub(start.elapsed()) {
            Some(timeout) => timeout,
            None => return Ok((guard, WaitTimeoutResult(true))),
        };
        guard = self.wait_timeout(guard, timeout)?;
    }
}
```

13.5 - Condivisione di messaggi

In alternativa alla condivisione dello stato, Rust offre un **meccanismo di comunicazione e sincronizzazione** tra thread basato sulla condivisione di messaggi.

- La funzione `std::sync::mpsc::channel<T>()` restituisce una coppia ordinata formata da una `struct Sender<T>` ed una `struct Receiver<T>`.
- Tutti i dati inviati tramite il metodo `send(..)` dalla prima possono essere consumati attraverso il metodo `recv()` della seconda, nello stesso ordine in cui sono stati inviati.
- Il metodo `send(..)` offre la garanzia che chi lo invoca non sarà bloccato (ovvero che il canale di comunicazione ha una capacità infinita di memorizzazione temporanea dei messaggi)

- Il metodo `recv()` si blocca senza consumare cicli macchina in attesa di un messaggio o della terminazione dell'oggetto `Sender` e di tutti i suoi eventuali cloni, mentre obbliga ad avere una singola copia dell'oggetto `Receiver`.
- L'implementazione fornità gode della proprietà **multiple producer - single consumer**, ovvero permette di creare più cloni dell'oggetto `sender` ma obbliga ad avere una singola copia dell'oggetto `receiver`.

In questo modello di comunicazione, il singolo **dato prodotto** da un thread **viene ceduto al canale** e da questo al thread ricevente che diventa il possessore finale del valore.

- Questa operazione agisce al tempo stesso da **sincronizzazione** (la ricezione è necessariamente successiva all'invio) e da **comunicazione** (il dato passato rappresenta l'unità di messaggio).
- Solo dopo che il `Sender` invia il messaggio, il `Receiver` può riceverlo.
- Non servono flag ausiliari.

Un **numero arbitrario di messaggi** può essere scambiato sul canale a condizione che il ricevitore sia attivo.

- Se il ricevitore viene deallocated, eventuali tentativi di invio falliscono con la generazione di un valore di tipo `SendError<T>`.
 - Il dato non inviato è contenuto nell'errore in modo tale da poterlo recuperare.
- Se tutti i trasmettitori vengono deallocated, tentativi di lettura sul ricevitore falliscono con la generazione di un valore di tipo `RecvError`.

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

let (tx, rx) = channel();

for _ in 0..3 {
    let tx = tx.clone();
    // cloned tx dropped within thread
    thread::spawn(move || tx.send("ok").unwrap());
}

// Drop the last sender to stop 'rx' waiting for a message.
// The program will not complete if we comment this out.
// **All** 'tx' needs to be dropped for 'rx' to have 'Err'.
drop(tx);

// Unbounded receiver waiting for all senders to complete.
while let Ok(msg) = rx.recv() {
    println!("{}: {}", msg);
}
```

Canali sincroni

La funzione `std::sync::mpsc::sync_channel<T>(bound: usize)` restituisce una coppia di valori di tipo (`SyncSender<T>`, `Receiver<T>`).

- A differenza di un canale semplice, questo è **limitato**: se il numero di messaggi giacenti nel canale raggiunge il limite definito (bound) le invocazioni del metodo `send(..)` diventano bloccanti fino a che non si libera un posto eseguendo una lettura con successo.

Se viene costruito un **canale sincrono di dimensione 0**, diventa un canale di tipo **rendezvous**: ogni operazione di lettura deve sovrapporsi temporalmente ad una scrittura.

- Le restanti operazioni offerte da `SyncSender<T>` hanno semantica simile alle corrispondenti offerte da `Sender<T>`.
- Il vantaggio di un rendezvous è che non richiede bufferizzazioni.

```
use std::sync::mpsc::sync_channel;
use std::thread;

let (sender, receiver) = sync_channel(1); // Max 1 message on the channel

sender.send(1).unwrap(); // This returns immediately
thread::spawn(move || {
    // This will block until the previous message has been received
    sender.send(2).unwrap();
});

assert_eq!(receiver.recv().unwrap(), 1);
assert_eq!(receiver.recv().unwrap(), 2);
```

13.6 - Crossbeam

Libreria che offre una serie di costrutti a supporto dell'elaborazione concorrente:

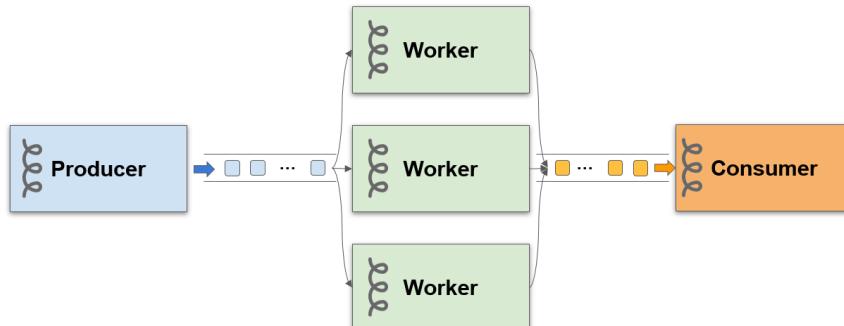
- **Costrutti atomici:**
 - La struct `crossbeam::atomic::AtomicCell<T>` estende il concetto di mutabilità interna offerto da `Cell<T>` a contesti multithread, appoggiandosi a primitive atomiche oppure ricorrendo all'uso di un lock interno per strutture dati più articolate.
- **Strutture dati concorrenti:**
 - La struct del crate `crossbeam::deque` (`Injector`, `Stealer` e `Worker`) offrono un meccanismo strutturato per la creazione di **schedulatori basati sul furto di attività** da eseguire.

- Le struct `crossbeam::queue::{ArrayQueue, SegQueue}` implementano code di messaggi (limitate o illimitate) basate sul paradigma **multiple-producer-multiple-consumer**.
- **Canali MPMC:**
 - Le funzioni `crossbeam::channel::{bounded(..), unbounded()}` creano canali unidirezionali con capacità limitata o illimitata basati sul paradigma MPMC i cui estremi possono essere condivisi per semplice clonazione.
 - Le funzioni `crossbeam::channel::{after(..), tick(..)}` creano il solo estremo di ricezione che consegnerà un messaggio dopo il tempo indicato o periodicamente.

Il paradigma MPMC offre un meccanismo potente per l'implementazione di pattern concorrenti in Rust.

- **Fan-out / Fan-in**

- Permette di distribuire attività a più thread indipendenti e raccogliere i risultati prodotti in un singolo punto
- usa una coppia di canali per distribuire e raccogliere i dati



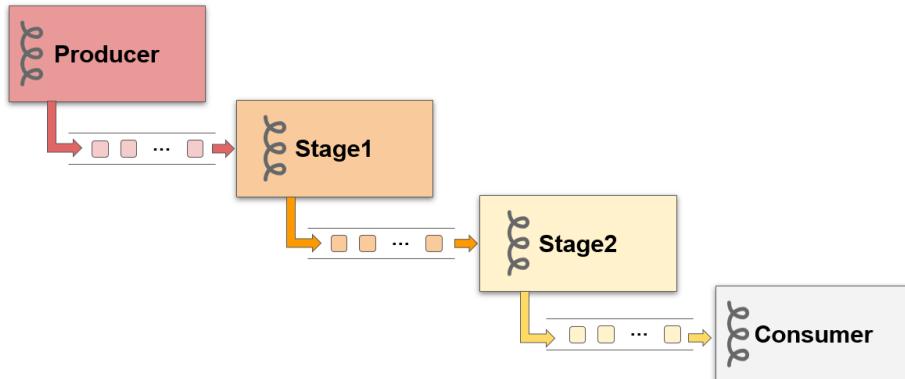
```

fn worker(id: usize, rx: Receiver<i32>, tx: Sender<String>) {
    while let Ok(value) = rx.recv() {
        tx.send(format!("W{} ({})", id, value)).unwrap();
    }
}

fn main() {
    let (tx_input, rx_input) = bounded::<i32>(10);
    let (tx_output, rx_output) = bounded::<String>(10);
    let mut worker_handles = Vec::new(); // vettore che contiene gli handle ai thread worker
    for i in 0..3 {
        let rx = rx_input.clone();
        let tx = tx_output.clone();
        worker_handles.push(thread::spawn(move || worker(i, rx, tx)));
    } ] genera i worker
    for i in 1..=10 { tx_input.send(i).unwrap(); } → producer
    drop(tx_input);
    while let Ok(result) = rx_output.recv() { → receiver
        println!("Received result: {}", result);
    }
    for handle in worker_handles { handle.join().unwrap(); }
}
  
```

- **Pipeline**

- Crea una serie di fasi di lavorazione, ciascuna delle quali è eseguita da un singolo thread e utilizza un canale per inoltrare i semi-lavorati tra due fasi successive.



```

fn stage_one(rx: Receiver<i32>, tx: Sender<String>) {
    while let Ok(value) = rx.recv() {
        tx.send(format!("S1({})", value)).unwrap();
    }
}
fn stage_two(rx: Receiver<String>, tx: Sender<String>) {
    while let Ok(value) = rx.recv() {
        tx.send(format!("S2( {} )", value)).unwrap();
    }
}
fn main() {
    let (tx_input, rx_input) = bounded::<i32>(10);
    let (tx_stage_one, rx_stage_one) = bounded::<String>(10); ] blocca
    let (tx_output, rx_output) = bounded::<String>(10); ] thread della pipeline

    let stage_one_handle = thread::spawn(move || stage_one(rx_input, tx_stage_one));
    let stage_two_handle = thread::spawn(move || stage_two(rx_stage_one, tx_output)); ] thread della pipeline

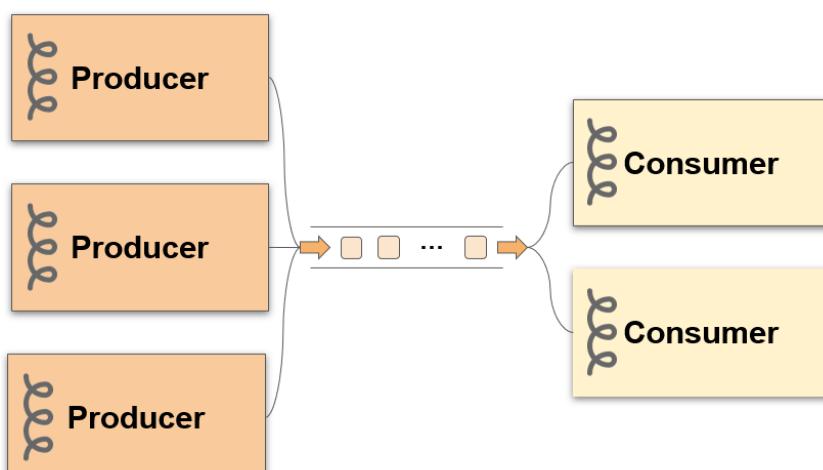
    for i in 1..=10 { tx_input.send(i).unwrap(); } —> Producer
    drop(tx_input);

    while let Ok(result) = rx_output.recv() { println!("Received result: {}", result); } —> Consumer

    stage_one_handle.join().unwrap();
    stage_two_handle.join().unwrap(); |—> mi assicuro che i thread terminino bene
}
  
```

- **Producer / Consumer**

- Consente ad uno o più thread produttori di generare valori che saranno elaborati dal primo thread consumatore disponibile.
- Usa un singolo canale per la comunicazione.



```

fn producer(id: usize, tx: Sender<(usize,i32)>) {
    for i in 1..=5 { tx.send((id,i)).unwrap(); }
}

fn consumer(id: usize, rx: Receiver<(usize,i32)>) {
    while let Ok((sender_id, val)) = rx.recv() {
        println!("Consumer {} received {} from {}", id, val, sender_id);
    }
}

fn main() {
    let (tx, rx) = bounded::<(usize,i32)>(10);

    let mut handles = Vec::new(); → Vec per contenere gli handle dei thread
    for i in 0..3 {
        let tx = tx.clone(); |→ Producer
        handles.push(thread::spawn(move || producer(i, tx)));
    }
    for i in 0..2 {
        let rx = rx.clone(); |→ Consumer
        handles.push(thread::spawn(move || consumer(i, rx)));
    }
    drop(tx); ← uccide il mio tx, gli altri muovono con i thread che me ne fanno il passo grazie alla move
    for handle in handles { handle.join().unwrap(); }
}

```

Modello degli attori

Modello concettuale che implementa la concorrenza a livello di tipo usando entità dette attori.

- Toglie il bisogno di lock e sincronizzazione fornendo un modo più pulito e lineare di introdurre il concetto di concorrenza in un sistema.

L'attore è la primitiva principale: contiene una **mailbox** alla quale possono essere inviati in modo asincrono messaggi.

- Un messaggio incapsula una richiesta che può essere inviata ad un attore.
- I messaggi vengono depositati nella mailbox dell'attore destinatario e sono normalmente elaborati in modalità FIFO.

La libreria **actix** offre un'implementazione di questo modello basata sul framework asincrono Tokio.

14 - Processi

14.1 - Processi e isolamento

Un **processo** costituisce l'unità base di esecuzione di un applicativo nel contesto di un SO.

- Esso viene identificato a livello di sistema tramite un numero intero **PID** (Process ID)
- Definisce uno **spazio di indirizzamento** all'interno del quale possono operare più thread
- Lo spazio di indirizzamento fornisce un meccanismo naturale di separazione (**isolamento**) allo scopo di evitare che le attività nel contesto di un processo possano "disturbare" quelle di altri processi.

Il livello di **isolamento** offerto dal concetto di processo è **parziale**:

- Due processi possono interferire attraverso il **file system**, sia a livello di **file "dati"** che a livello di **eseguibili e librerie**.
- Anche il **sottosistema di rete** e il sistema di **autenticazione/autorizzazione/accounting** sono fonti di potenziale interferenza.
- In generale, l'accesso alle periferiche di sistema ed alle risorse centralizzate può causare incompatibilità.

In alcune situazioni, il progettista vuole **esplicitamente ridurre** il livello di **isolamento** tra due o più processi, offrendo un meccanismo controllato di comunicazione in grado di superare i limiti imposti dalla separazione degli spazi di indirizzamento.

- I SO offrono meccanismi opportuni che ricadono sotto il nome di **IPC (Inter-Process Communication)**

Concorrenza e processi

L'uso dei thread permette di sfruttare le risorse computazionali presenti in un elaboratore.

- La presenza di uno spazio di indirizzamento condiviso facilita la coordinazione e la comunicazione tra thread.

Ci sono situazioni in cui la presenza di un singolo spazio di indirizzamento non è possibile o desiderabile:

- Riuso di programmi esistenti (di cui non abbiamo il controllo)
- Scalabilità su più computer (big data)
- Sicurezza (non è preferibile che molti tocchino lo stesso address space se questo non è strettamente necessario)

È possibile decomporre un sistema complesso in un insieme di processi collegati, creandoli a partire da un processo genitore e permettendo la cooperazione indipendente dalla loro genesi.

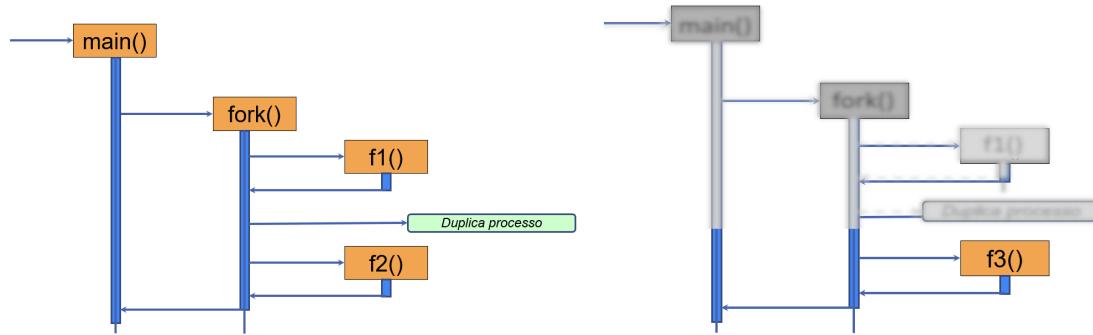
- Ad ogni processo è associato almeno un thread (**primary thread**)
- Un sistema multi processo è intrinsecamente concorrente e solleva gli stessi problemi di interferenza e necessità di coordinamento del multi thread.

Processi in Windows:

- Costituiscono entità separate, senza relazioni di dipendenza esplicita tra loro
- La funzione **CreateProcess(..)**
 - crea un nuovo spazio di indirizzamento,
 - lo inizializza con l'immagine di un eseguibile
 - attiva il thread primario al suo interno.
- Il processo figlio può condividere variabili d'ambiente ed handle a file, semafori, pipe ma non può condividere handle a thread, processi, librerie dinamiche e regioni di memoria.

Processi in Linux:

- Si crea un processo figlio con la system call **fork()**:
 - Crea un nuovo spazio di indirizzamento “identico” a quello del processo genitore
 - I due processi condividono i riferimenti alle stesse pagine di memoria fisica
 - Il figlio inizia la propria computazione trovandosi già:
 - uno stack popolato con la storia delle chiamate effettuate nel padre,
 - uno heap con della memoria allocata,
 - codice e spazio globale nello stesso stato in cui erano nel padre.
- Dopo l'esecuzione di **fork()**, tutte le pagine sono marcate con i flag **CopyOnWrite**.
 - Eventuali scritture comportano la duplicazione della pagina e la separazione tra i due spazi di indirizzamento
- Le funzioni **exec***() sostituiscono l'attuale immagine di memoria dello spazio di indirizzamento ri-inizializzandola a quella descritta dall'eseguibile indicato come parametro.
- Nel caso di programmi concorrenti, l'esecuzione di **fork()** crea un problema:
 - Il processo figlio conterrà un solo thread.
 - Gli oggetti di sincronizzazione presenti nel padre possono trovarsi in stati incongruenti!
 - **int pthread_atfork(**
 void (*prepare)(void),
 void (*parent)(void),
 void (*child)(void)
);
 - Registra un gruppo di funzioni che saranno chiamate in corrispondenza alle invocazioni a **fork()**.



Terminare un processo

Un processo continua la propria esecuzione finché non termina **di propria volontà** o viene **terminato dall'esterno**.

- Una opportuna **system call** permette di richiedere la terminazione del processo corrente e la conseguente deallocazione di tutte le risorse in uso e la loro restituzione al SO, che potrà renderle disponibili ad altri processi.

In Windows si invoca **`ExitProcess(int status)`** e in Linux la funzione **`_exit(int status)`**.

- Entrambe causano l'**immediata terminazione** di tutti gli altri thread associati al processo, senza ulteriori possibilità di esecuzione.
- Tutti i file aperti vengono chiusi.
 - Nel caso di Windows, nel contesto del thread che ha eseguito la `ExitProcess(..)` vengono rilasciate le DLL eventualmente caricate.

Le librerie standard del C e C++ offrono una soluzione portabile e più articolata:

- la funzione **`void exit(int status)`** definita in `<stdlib.h>`
- la funzione **`void std::exit(int status)`** definita in `<cstdlib>`
- tramite la funzione **`int std::atexit(void (*callback)())`** è possibile registrare una o più callback da invocare all'atto della terminazione. Il valore restituito indica se la registrazione è andata a buon fine o meno.

Un programma termina anche quando la funzione principale ritorna:

- Questo è conseguenza del codice di startup inserito dalla libreria di supporto del linguaggio.
- Questa inizializza l'ambiente di esecuzione, invoca la funzione `main(..)` e utilizza il valore da essa ritornato per invocare `exit(status)`.
- Se si verifica un'eccezione non gestita nel thread principale o in uno secondario, viene invocata la funzione `ExitProcess(..)` o `_exit(..)` con un codice di stato definito dalla libreria di esecuzione.

Il **valore restituito** dalla funzione **exit(..)** è arbitrario:

- Vale una convenzione generale per la quale 0 indica una terminazione “pulita” e qualunque altro valore indica una terminazione con errore.
- Il significato di tale codice, tuttavia, non è oggetto di specifica da parte del SO. Occorre documentare opportunamente il valore e attenersi alle buone pratiche definite in ciascun ambiente (OS + compilatore + libreria standard)

14.2 - Gestione dei processi

Per la gestione dei processi, la libreria standard di Rust mette a disposizione il **modulo std::process**

- I metodi offerti da Rust utilizzano al loro interno le **system call** del kernel del SO per gestire i processi.

Creare un processo

La **struct Command** permette la **creazione di un nuovo processo**:

- Utilizza il **pattern builder** per configurare, creare ed interagire con il processo figlio.
- I metodi **arg()** e **args()** possono essere utilizzati per passare al processo figlio rispettivamente uno o più argomenti.
- Il metodo **output()** genera il processo ed attende la sua terminazione ritornando un valore di tipo **Result<Output>**.

```
pub struct Command {  
    program: OsString,  
    args: Vec<Arg>,  
    env: CommandEnv,  
    cwd: Option<OsString>,  
    flags: u32,  
    detach: bool, // not currently exposed in std::process  
    stdin: Option<Stdio>,  
    stdout: Option<Stdio>,  
    stderr: Option<Stdio>,  
    force_quotes_enabled: bool,  
}  
  
pub struct Command {  
    inner: imp::Command,  
}
```

```
pub fn output(&mut self) -> io::Result<(ExitStatus, Vec<u8>, Vec<u8>)> {  
    let (proc, pipes) = self.spawn(Stdio::MakePipe, false)?;  
    crate::sys_common::process::wait_with_output(proc, pipes)  
}
```

```
pub fn output(&mut self) -> io::Result<Output> {  
    let (status, stdout, stderr) = self.inner.output()?;  
    Ok(Output { status: ExitStatus(status), stdout, stderr })  
}
```

```
pub struct Output {  
    pub status: ExitStatus,  
    pub stdout: Vec<u8>,  
    pub stderr: Vec<u8>,  
}
```

```

use std::process::Command;

fn main() {
    let output = if cfg!(target_os = "windows") {
        Command::new("cmd")
            .args(["/C", "echo hello"])
            .output()
            .expect("failed to execute process")
    } else {
        Command::new("sh")
            .arg("-c")
            .arg("echo hello")
            .output()
            .expect("failed to execute process")
    };

    println!("{}: {:?}", output);
    // Output { status: ExitStatus(unix_wait_status(0)), stdout: "hello\n", stderr: "" }
}

```

per conoscere info sul sistema in cui sono eseguite le cose

comando da eseguire

attende la partenza del processo, raccogli ciò che ha scritto su stdout e mettilo dentro una struct Output

custom dell'errore se viene scatenato

può chiamare un solo arg con tutti gli argomenti

È possibile **configurare** le **variabili di ambiente** del processo prima di avviarlo.

- Per default, esso eredita quelle del processo corrente.
- I metodi `env<K, V>(&mut self, key: K, val: V)` e `envs<I, K, V>(&mut self, vars: I)` permettono di effettuare **aggiunte o modifiche**.
- I metodi `env_remove<K>(&mut self, key: K)` e `env_clear(&mut self)` permettono di **eliminare** una/tutte le variabili d'ambiente.
- È possibile conoscere l'**elenco delle variabili d'ambiente** usate da una struct di tipo `Command` con il metodo `get_envs(&self)` che restituisce un iteratore a tuple formate dalle coppie chiave/valore.

È possibile **ridirigere i flussi standard di ingresso/uscite**, tramite i metodi `stdin(..)`, `stdout(..)` e `stderr(..)`. È possibile passare loro una delle seguenti opzioni:

- `inherit()` → il processo figlio eredita il descrittore in uso nel processo genitore.
- `piped()` → verrà creata una pipe monodirezionale, un'estremità della quale sarà passata al processo figlio mentre l'altra sarà memorizzata nella struttura restituita dal comando di avvio.
- `null()` → il flusso sarà ignorato.

È possibile **modificare la cartella in cui si avvia il processo figlio** tramite il metodo `current_dir<P: AsRef<Path>>(&mut self, dir: P)`.

Il metodo `status(&mut self)` **avvia il processo** e ne attende la terminazione.

- Esso restituisce un valore di tipo `Result<ExitStatus>`, dove `ExitStatus` è una struttura che permette di avere informazioni sul codice di uscita del processo e, nel caso di sistemi Unix, sulla motivazione della sua terminazione.

Il metodo `spawn(&mut self)` avvia il processo senza attendere la terminazione.

- Esso restituisce un valore di tipo `Result<Child>`, dove `Child` è una struttura che consente di rappresentare ed interagire con il processo figlio.
- La struttura `Child` offre vari meccanismi per controllare e condizionare lo svolgimento del processo figlio.
 - Attraverso i campi `stdin`, `stdout`, `stderr` è possibile fare accesso alle handle dei relativi flussi, qualora siano stati catturati.
 - Il metodo `id(&self)` restituisce l'identificativo univoco assegnato dal SO
 - Il metodo `wait(&mut self)` ne attende la terminazione, restituendo il codice di uscita.
 - Il metodo `wait_with_output(&mut self)` chiude il flusso di ingresso del processo figlio, ne attende la terminazione e raccoglie quanto non ancora letto dei flussi di uscita ed errore in una struttura di tipo `Output`.
 - Il metodo `kill()` ne forza la terminazione.

Nota: Il terzo modo per far partire un processo è tramite la funzione `output()`.

```
use std::io::Write;
use std::process::{Command, Stdio};

let mut child = Command::new("rev")
    .stdin(Stdio::piped())           ↗ reverse (scrive la stringa in input al contrario)
    .stdout(Stdio::piped())           ↗ prepara una pipe per alimentare l'input
    .spawn()                         ↗ pipeline per prendere dall'output
    .expect("Failed to spawn child process");

let mut stdin = child.stdin.take().expect("Failed to open stdin");
std::thread::spawn(move || {
    stdin.write_all("Hello, world!".as_bytes())
        .expect("Failed to write to stdin");
});

let output = child.wait_with_output().expect("Failed to read stdout");
assert_eq!(String::from_utf8_lossy(&output.stdout), "!dlrow ,olleH");
```

Se, in Linux, un processo esegue `fork()`, tutto lo stato della sua memoria sarà duplicato.

- Nel caso in cui tale processo avesse avuto nei buffer di I/O contenuto pendente, tale contenuto verrà scaricato sul dispositivo corrispondente sia dal processo padre che da quello figlio, non appena eseguiranno una operazione di `flux()` o satureranno tale buffer con ulteriori operazioni.

Il problema è più evidente quando l'output di un programma è rediretto verso un file:

- Questo abilita una modalità di scrittura a blocchi (al posto di quella standard a linee) che aumenta la probabilità di osservare tale fenomeno
- Prima di eseguire `fork()`, può essere conveniente eseguire un `flush()` di tutti i file aperti

Terminare un processo

La funzione `std::process::exit(code: i32) → !` termina immediatamente il processo corrente, con tutti i thread presenti al suo interno.

- Nessun distruttore presente sullo stack del thread corrente né degli altri thread viene eseguito.
- È responsabilità del programmatore invocare questa funzione solo in punti in cui si abbia la sicurezza che tutto ciò che doveva essere liberato sia stato liberato.
- Nonostante il valore ricevuto sia a 32 bit, nella maggior parte dei sistemi Unix-like, solo gli 8 bit meno significativi sono effettivamente passati al SO.

La funzione `std::process::abort() → !` termina immediatamente il processo corrente, con tutti i thread presenti al suo interno.

- Forza un codice di errore che viene interpretato come **interruzione anomala**
- Valgono le stesse cautele espresse per la `exit(..)`.

La macro `panic!(..)` causa la **contrazione dello stack corrente**, con l'esecuzione di tutti i distruttori posti al suo interno, **senza determinare la terminazione del processo**.

- A meno che la chiamata avvenga nel contesto del thread principale.

Gestire altri processi

Ogni SO offre meccanismi per attendere la terminazione di un processo di cui si conosce la handle.

- Tale attesa non comporta consumo di CPU e termina quando il processo osservato termina per qualche motivo.
- In Windows si utilizzano `WaitForSingleObject(..)` / `WaitForMultipleObjects(..)` e `GetExitCodeProcess(..)`
- In Linux si utilizzano `wait(..)`, `waitpid(..)` e `waitid(..)`.

La funzione `pid_t wait(int *status)`, definita in `<sys/wait>`, opera come segue:

- Se un processo non ha processi figli, la funzione ritorna -1.
- Se nessuno dei figli dell'attuale processo è terminato, la chiamata si blocca in attesa di tale evento.
- In presenza di un figlio terminato:
 - Se `status` non è nullo, al suo interno viene indicata la motivazione che ne ha causato la terminazione.
 - Il sistema aggiorna i contatori di utilizzo del sistema (CPU, memoria, IO, rete) del processo corrente aggiungendo quelli del processo figlio.
 - Viene restituito il PID del figlio terminato
 - Eventuali ulteriori chiamate a `wait(..)` da parte del processo corrente non forniranno più indicazioni relative a questo processo figlio.

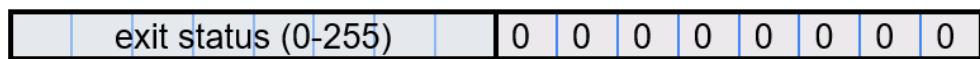
La funzione `pid_t waitpid(pid_t pid, int *status, int options)` permette di aspettare uno specifico processo figlio e di eseguire una verifica non bloccante del suo stato attuale.

- Attenzione all'uso del polling che può causare consumi eccessivi di CPU e batteria.
 - L'intero a cui punta **status** (se non è nullo) viene inizializzato con un valore a 16 bit. Il suo contenuto è definito in una serie di sotto-campi.

Il crate **sysinfo** permette di accedere alle informazioni dei processi così come esposte dai diversi SO.

```
let mut system = sysinfo::System::new();
system.refresh_all();
let name = system.process(Pid::from(1)).unwrap().name();
println!("Process with id 1 is {}", name);
```

Terminazione normale



Terminazione a seguito di un segnale



Bloccato da un segnale



Continuato da un segnale



Se un processo padre termina prima che l'ultimo dei suoi figli sia terminato, il processo figlio diventa **orfano**.

- Linux riassegna a tale figlio il PID 1 (init) come ID del processo padre.
 - Questo può essere sfruttato da un processo per sapere se il proprio padre è ancora vivo o meno (ipotizzando che non sia stato lanciato direttamente da init).

Se un processo figlio termina prima che il rispettivo padre abbia eseguito `wait*(..)` diventa **zombie**.

- La maggior parte delle sue risorse viene restituita dal SO, tranne l'ID, lo stato di terminazione e i contatori relativi all'utilizzo delle risorse
 - Quando il padre effettua un `wait(..)` lo zombie viene rimosso.

14.3 - InterProcess Communication (IPC)

Il SO impedisce il trasferimento diretto di dati tra processi.

- Ogni processo dispone di uno spazio di indirizzamento separato e non è possibile sapere cosa sta capitando in un altro processo.
- Ogni SO offre alcuni meccanismi per superare tale barriera in modo controllato, permettendo lo scambio di dati (**comunicazione**) e la **sincronizzazione** delle attività.

Indipendentemente dal tipo di meccanismo adottato, occorre adattare le informazioni scambiate, così da renderle comprensibili al destinatario.

- Internamente un processo può usare una **varietà di rappresentazioni**:
 - Tipi elementari (numerici, logici, caratteri, ...)
 - Tipi strutturati (record, array, classi, ...)
 - Puntatori per strutture dati complesse (alberi, grafi, ...)

La **rappresentazione interna** al processo non è adatta ad essere esportata.

- I puntatori non hanno senso al di fuori del proprio spazio di indirizzamento.
- Alcune informazioni non sono esportabili, ad es. gli handle dei file sono un numero (file descriptor) che ha senso solo all'interno del processo.

La **rappresentazione esterna** è un formato intermedio che permette la rappresentazione di strutture dati arbitrarie sostituendo i puntatori con riferimenti indipendenti alla memoria.

- Formati basati su testo:
 - XML, JSON, CSV, ...
- Formati binari:
 - XDR, HDF, protobuf, ...
- Possono essere trattate come **blocchi compatti di byte** che possono essere duplicati e trasferiti senza comprometterne il significato.

I dati vengono scambiati nel formato esterno.

- La sorgente esporta le proprie informazioni (**marshalling**).
- Il destinatario ricostruisce una rappresentazione su cui può operare direttamente (**unmarshalling**).
- Le operazioni di marshalling e unmarshalling possono essere codificate esplicitamente o essere eseguite da codice generato automaticamente dall'ambiente di sviluppo.

Coda di messaggi

Struttura dati mantenuta dal SO che **permette a molti processi sorgente di inviare messaggi ad uno specifico destinatario.**

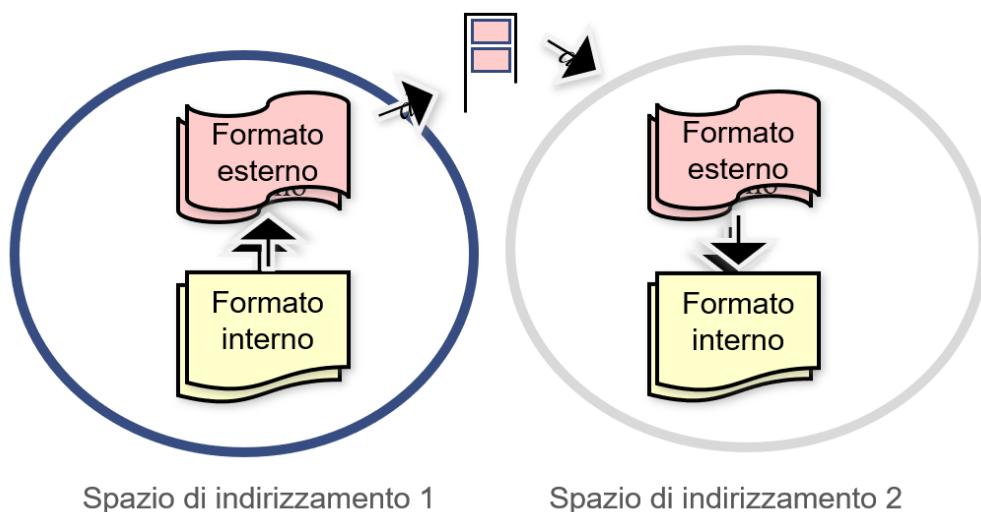
- La comunicazione è di tipo **asincrono**, il SO si fa carico di memorizzare il messaggio inviato fino a che il destinatario non lo leggerà.
- La comunicazione è **mono-direzionale**.

Ogni coda è caratterizzata da un **identificativo univoco** a livello di SO (una stringa).

- Il destinatario della coda è responsabile di inizializzarla e leggerla al suo nome
- Le sorgenti devono avere modo di conoscere a priori il nome della coda o servirsi di altri meccanismi di IPC per venirne a conoscenza.

Gli oggetti “**mailslot**” in Windows e “**fifo**” in Linux ricadono in questa categoria

- Vengono costruiti in C/C++ tramite le funzioni `CreateMailSlot(..)` e `mkfifo(..)`



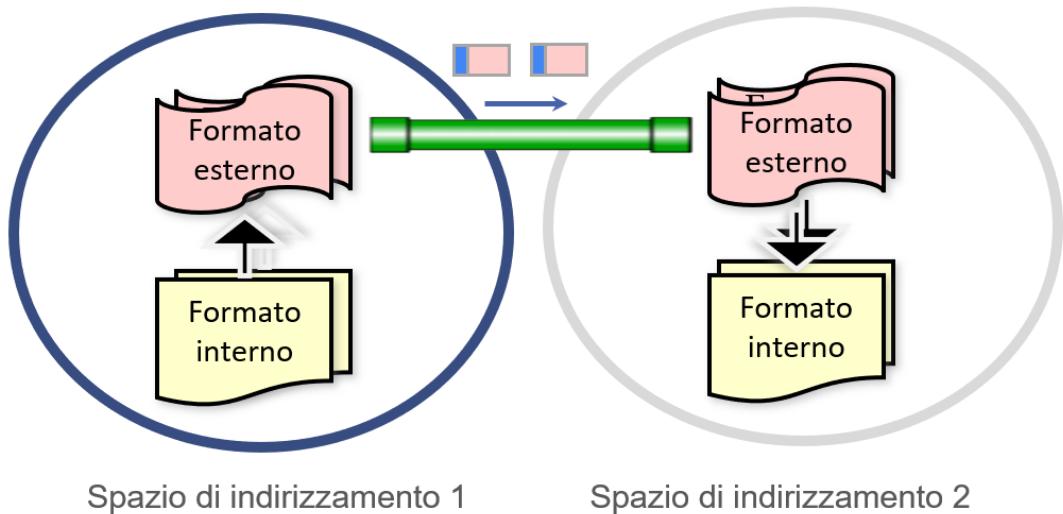
Pipe

Le pipe sono “tubi” che permettono il **trasferimento di sequenze di byte di dimensioni arbitrarie**

- Occorre inserire marcatori che consentono di delimitare i singoli messaggi.
- Comunicazione sincrona 1-1
- Vengono implementate come **buffer all'interno della memoria del kernel**.

In C/C++, una pipe può essere creata da programma con le funzioni **BOOL CreatePipe(..)** in Windows e **int pipe(int fd[2])** in Linux.

- Si legge e scrive da una pipe con le operazioni di lettura/scrittura del SO:
 - **FileRead(..)/FileWrite(..)** in Windows
 - **read(..)/write(..)** in Linux
- Si chiude una pipe con la funzione corrispondente:
 - **CloseHandle(..)/close(..)**



```
let echo_child = Command::new("echo")
    .arg("Oh no, a tpyo!")
    .stdout(Stdio::piped()) → redirige l'output
    .spawn() → parte la pipe
    .expect("Failed to start echo process");

let echo_out = echo_child.stdout.expect("Failed to open echo stdout");

let mut sed_child = Command::new("sed") → stream editor ( UNIX command )
    .arg("s/tpyo/typo/")
    .stdin(Stdio::from(echo_out)) ← pipe in
    .stdout(Stdio::piped()) ← pipe out
    .spawn() ← parte
    .expect("Failed to start sed process");

let output = sed_child.wait_with_output().expect("Failed to wait on sed");
assert_eq!(b"Oh no, a typo!\n", output.stdout.as_slice());
```

Scambio di messaggi strutturati

Il crate **serde** (SERialize DEserialize) offre le funzionalità necessarie a serializzare e deserializzare buona parte delle strutture dati Rust usando vari tipi di formato dati possibili.

- Per ognuno di questi formati esiste un apposito crate che deve essere incluso insieme a quello base.

La libreria estende la macro `#[derive(Serialize, Deserialize)]` per aggiungere in modo automatico il supporto delle operazioni di conversione a tipi definiti dall'utente.

- Occorre includere il crate `serde` con la relativa versione e abilitare la funzionalità `derive`.
- `serde = {version = "1.0.137", features = ["derive"]}`

```
struct W {
    a: i32,
    b: i32
}
let w = W {a: 0, b: 0}; // Rappresentato come l'oggetto '{"a":0,"b":0}'

struct X(i32, i32);
let x = X(0, 0);      // Rappresentato come l'array '[0,0]'

struct Y(i32);
let y = Y(0);          // Rappresentato come il solo valore '0'

struct Z;
let z = Z;              // Rappresentato come 'null'

enum E {
    W { a: i32, b: i32},
    X(i32, i32),
    Y(i32),
    Z,
}

let w = E::W {a: 0, b: 0} // Rappresentato come '{"W":{"a":0,"b":0}}'
let x = E::X(0, 0);      // Rappresentato come '{"X": [0,0]}'
let y = E::Y(0);          // Rappresentato come '{"Y":0}'
let z = E::Z;              // Rappresentato come '"Z"'
```

È possibile alterare il comportamento di default, ottenendo rappresentazioni alternative delle enumerazioni, quando occorra interoperare con altri linguaggi.

```
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize, Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point = Point { x: 1, y: 2 };

    let serialized = serde_json::to_string(&point).unwrap();

    println!("{}", serialized); // {"x":1,"y":2}

    let deserialized: Point = serde_json::from_str(&serialized).unwrap();

    println!("{}:{}", deserialized); // Point { x: 1, y: 2 }
}
```

Comunicazione tra processi

Il crate **interprocess** offre la possibilità di gestire la comunicazione tra processi tramite un'interfaccia univoca multipiattaforma continuando a garantire le funzionalità specifiche dei SO.

- Unnamed/Windows named pipes, Posix/C signals, socket

Il crate **zbus** è l'implementazione rust del protocollo D-Bus ed offre una vasta gamma di astrazioni per la comunicazione tra processi.

- Disponibile esclusivamente su piattaforme Linux
- Ampio supporto alla programmazione asincrona tramite l'utilizzo del **Tokio** runtime.

```
use std::io::{BufRead, BufferedReader, Write};
use std::process::{Command, Stdio};
use std::sync::mpsc::{channel, Receiver, Sender};
use std::thread;
use std::thread::sleep;
use std::time::Duration;

fn start_process(sender: Sender<String>, receiver: Receiver<String>) {

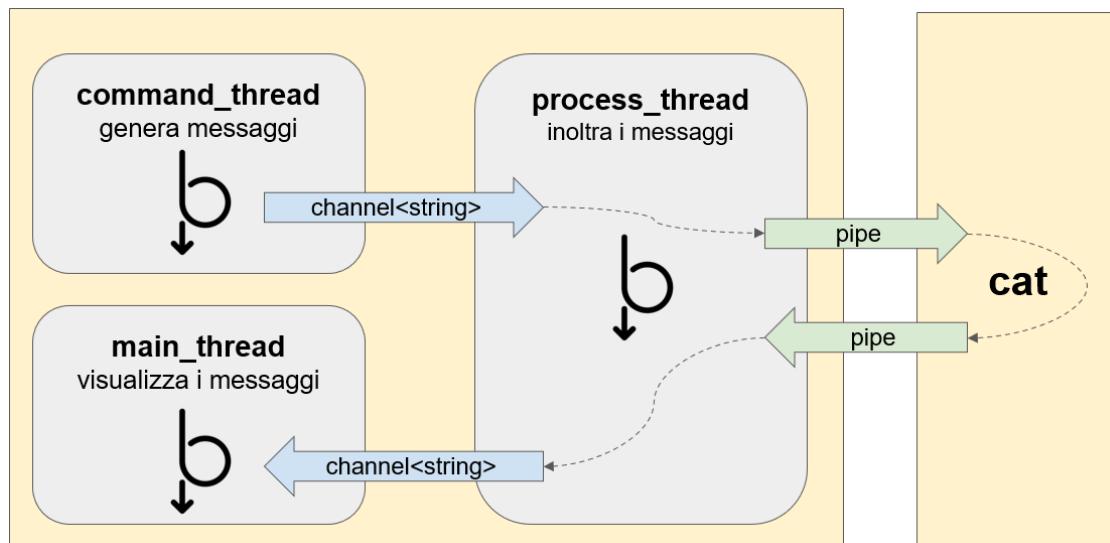
    let child = Command::new("cat")
        .stdin(Stdio::piped())
        .stdout(Stdio::piped())
        .spawn()
        .expect("Failed to start process");
    thread::spawn(move || {
        let mut pipe_in = BufferedReader::new(child.stdout.unwrap());
        let mut pipe_out = child.stdin.unwrap();
        for line in receiver {
            pipe_out.write_all(line.as_bytes()).unwrap();
            let mut buf = String::new();
            match pipe_in.read_line(&mut buf) {
                Ok(_) => {
                    // inoltra quanto ricevuto dalla pipe
                    // sul canale di uscita
                    sender.send(buf).unwrap();
                    continue;
                }
                Err(e) => {
                    println!("an error!: {:?}", e);
                    break;
                }
            }
        }
    });
}
```

```

fn start_command_thread(sender: Sender<String>) {
    thread::spawn(move || {
        for i in 1..10 {
            sleep(Duration::from_secs(3));
            sender.send(String::from(format!("Message {} from command
thread\n", i))).unwrap();
        }
    });
}

fn main() {
    let (tx1, rx1) = channel();
    let (tx2, rx2) = channel();
    start_process(tx1, rx2);
    start_command_thread(tx2);
    rx1.iter().for_each(|line| println!("Echo process response: {}", line))
}

```



15 - Programmazione asincrona

15.1 - Esecuzione asincrona

La programmazione multithread ha il vantaggio di poter parallelizzare l'esecuzione di algoritmi complessi e di sfruttare appieno hardware multicore, ma il prezzo viene pagato, da un lato, dalla **complessità** legata ai **meccanismi di sincronizzazione** e, dall'altro, dalla necessità di **allocare preventivamente lo stack** di esecuzione di ciascun thread.

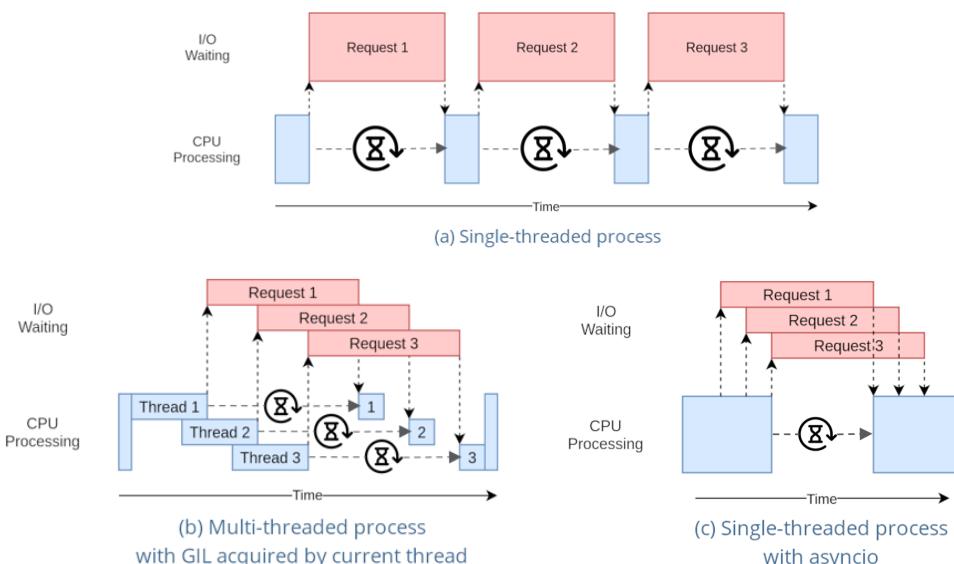
- In caso di creazione di molti thread (1000+), tale costo diventa significativo e proibitivo.

L'approccio multi-thread non è ottimale in quelle situazioni in cui la computazione richiede di **ricevere informazioni da un sottosistema separato** (file system, rete, un timer, un altro programma, etc..).

- In queste situazioni l'esecuzione non può proseguire e occorre attendere che il sottosistema in questione fornisca le informazioni attese.
- Normalmente, il SO rileva la situazione e sposta il thread corrente nello stato "NotRunnable", sospendendone l'esecuzione fino a che non si verifica la condizione attesa.

Se il programma deve svolgere altri compiti, oltre a quello che ha generato l'attesa, si creano tre possibilità:

- a. Gli altri compiti saranno **eseguiti successivamente**, dal thread corrente
- b. Si creano più **thread secondari per eseguirli**, accollandosi la complessità legata alla loro sincronizzazione.
- c. Si organizza il codice in modo tale da **separare la richiesta di eseguire l'operazione dalle operazioni che dovranno essere fatte quando arriverà la risposta**, così da non bloccare l'esecuzione del thread corrente, ammesso che ci sia altro da fare.

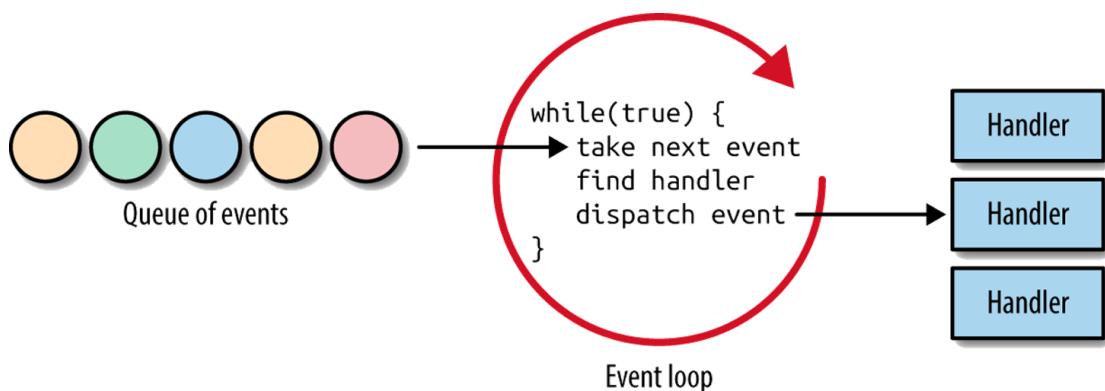


Elaborazione asincrona

“Si usano i thread quando occorre elaborare in parallelo, mentre si usa l'**esecuzione asincrona** quando occorre **attendere in parallelo**¹.

Il modo più diretto di implementare la terza strategia richiede che le **azioni conseguenti ad una data operazione bloccante siano racchiuse in un'apposita funzione** (callback)

- Tale funzione riceve come parametro il risultato dell'operazione bloccante e sarà invocata quando il sottosistema che è stato interrogato avrà fornito la propria risposta.
- A seconda del linguaggio utilizzato esistono soluzioni diverse a chi si occupa di fornire tale valore e in quale thread avverrà la chiamata.
 - In JavaScript, ad esempio, il modello di esecuzione prevede la presenza di una coda dei messaggi, in cui il driver del sottosistema interrogato (eseguito in un thread separato) provvede ad inserire il risultato.
 - Il thread principale esegue costantemente un ciclo, in cui attende la presenza di un messaggio, lo estrae dalla coda e lo elabora.
 - Quando la risposta arriverà, questa verrà naturalmente elaborata dal ciclo di elaborazione dei messaggi.



Un tipico modo per implementare operazioni asincrone è basarsi su **API ad eventi**, ovvero funzioni che permettono di **richiedere ad uno strato soggiacente** (come il SO o una piattaforma di esecuzione) **l'operazione che si intende eseguire**, passando una callback che dovrà essere invocata nel momento in cui lo strato soggiacente avrà terminato la sua parte di esecuzione.

```
val f1: AsyncRead = ...  
val f2: AsyncRead = ...  
read_async(f1, vec![], |buffer: &[u8]|{  
    // process buffer from file1...  
});  
read_async(f2, vec![], |buffer: &[u8]|{  
    // process buffer from file2  
});
```

¹ L. Palmieri, Zero to production in Rust, 2022

Questo modo di scrivere codice crea grossi problemi che ricadono sotto il nome di **callback hell**, ad esempio quando l'azione che si sta compiendo richiede, in cascata, una seconda azione asincrona.

- Occorre che la callback indicata provveda ad invocare un'ulteriore operazione passando la relativa callback
- Le cose si complicano se tale operazione è ciclica e se occorre gestire eventuali errori. Occorre trasformare il codice in una **macchina a stati finiti**.
- Gli errori possono originarsi in momenti molto diversi:
 - All'atto dell'invocazione della funzione asincrona
 - Come conseguenza dell'elaborazione asincrona

```
let h1 = open_file_async("f1", FileMode::read)?;
let h2 = open_file_async("f2", FileMode::write)?;
let mut buffer = vec![];
read_async(h1, &mut buffer, |res1| {
    if (res1.is_ok()) {
        write_async(h2, res1.unwrap(), |res2| {
            if (res2.is_ok()) {
                //scrittura completata con successo
            } else {
                //scrittura fallita
            }
        });
    } else {
        //lettura fallita
    }
})?; // impossibilità di leggere
//Quando il programma arriva qui, non è ancora
//stato fatto nulla!
```

Esecuzione parziale

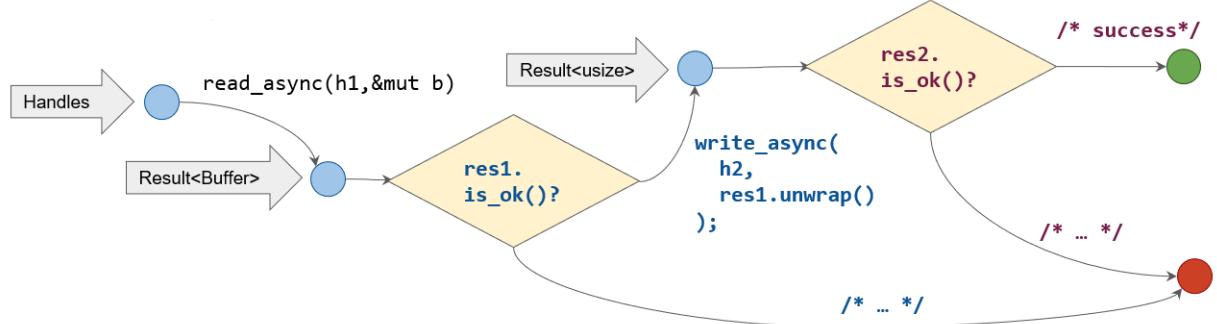
Un primo passo che aiuta a semplificare il problema è provare a riscrivere la **forma annidata** delle callback **in forma lineare**, appoggiandosi ad una struttura dati che mantenga le informazioni di stato (**Future**, che corrisponde alle Promise di JavaScript)

```
read_async(h1, &mut buffer, |res1| {
    if (buffer.is_ok()) {
        write_async(h2, res1.unwrap(), |res2| {
            if (res2.is_ok()) { /* success */ }
            else { /* ... */ }
        });
    } else {
        //lettura fallita
    }
});
```

```
read_async(h1, &mut buffer) ↗ Future<Result<[u8]>>
    .and_then(|res1| {
        if (res1.is_ok()) {
            write_async(h2, res1.unwrap());
        } else { /* ... */ }
    }) ↗ Future<Result<u8>>
        .and_then(|res2| {
            if (res2.is_ok()) { /* success */ }
            else { /* ... */ }
        }) ↗ Future<Result<()>>
            .map_error(|err| { /* ... */ });
```

Questa forma aiuta a vedere come le operazioni che si intendono eseguire possono essere viste come una **macchina a stati finiti**, che evolve “a strappi”.

- Quando raggiunge uno stato intermedio ritorna e, per continuare, sarà necessario riprendere l'esecuzione passando come parametro l'esito dell'operazione asincrona che ha causato la sospensione.



La macchina a stati finiti può essere implementata da una **chiusura**

- Essa racchiude il proprio **stato** e tutte le **variabili locali** di cui l'esecuzione ha bisogno.
- Quando viene eseguita ritorna un valore che indica se ha raggiunto uno degli stati finali o se si trova ancora in uno stato intermedio.
- Questo permette di non bloccare il thread corrente e procedere con l'esecuzione di altri task.

I **punti di blocco** sono tutti in corrispondenza degli **stati intermedi**.

- Questi sono immediatamente preceduti dall'indicazione di una funzione asincrona.
- Quando uno stato intermedio viene raggiunto, la chiusura ritorna.

Perché la macchina a stati possa progredire occorrono due condizioni:

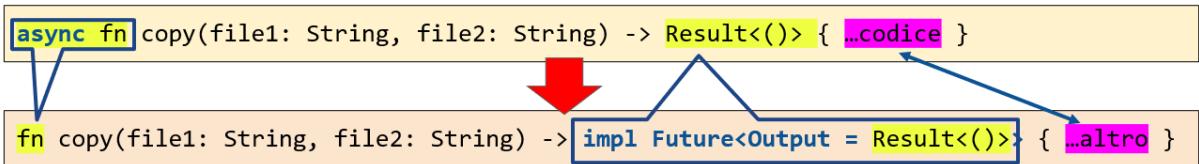
1. Che la funzione asincrona invocata scateni qualche attività (in che thread avviene?) che possa portare all'evoluzione dello stato
2. Che ci sia qualcuno che indichi che vi sono le condizioni affinché lo stato possa evolvere

15.2 - Async e await

Il compilatore Rust supporta esplicitamente la programmazione asincrona grazie all'introduzione di due parole chiave nel linguaggio (**async** e **await**) ed alla presenza di un tratto specifico (**Future**) nelle librerie core.

- Se una funzione o un blocco di **codice** sono preceduti dalla parola chiave **async**, il **compilatore** ne analizza il contenuto e **lo trasforma in una macchina a stati**.
- La funzione o il blocco vengono ridotti all'inizializzazione di tale macchina a stati.

- Il tipo restituito viene trasformato da **T** in un **tipo anonimo** che implementa il tratto **Future** al cui interno è implementata la macchina a stati precedentemente sintetizzata.



Se all'interno del codice della funzione è presente una chiamata ad un'altra funzione asincrona, per poter accedere al suo risultato occorre esplicitamente attenderlo, attraverso l'operatore `.await`

- Questo introduce un **nuovo stato** all'interno della macchina a stati associata alla funzione.

```
async fn copy(h1: FileHandle, h2: FileHandle) -> std::io::Result<()> {
    let mut buffer = vec![];
    h1.read_async(&mut buffer).await?; // Prosegue quando la Future
    h2.write_async(&buffer).await // sarà disponibile
}
```

Il tratto Future

```
pub trait Future {
    type Output;

    // Required method
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

- **Pin<T>** è un particolare smart pointer che impedisce che la scrittura venga mossa, permettendo l'uso di riferimenti relativi.
 - Serve quando ho dei puntatori a parti di me stesso: se venissi mosso, i bit verrebbero copiati ma i puntatori perderebbero di significato.
- **Context** incapsula un oggetto di tipo **Waker**, mediante il quale è possibile notificare all'esecutore che il metodo **poll(..)** può essere richiamato.

```
pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

```

pub struct Context<'a> {
    waker: &'a Waker,
    // Ensure we future-proof against variance changes by forcing
    // the lifetime to be invariant (argument-position lifetimes
    // are contravariant while return-position lifetimes are
    // covariant).
    _marker: PhantomData<fn(&'a () -> &'a ())>,
    // Ensure `Context` is `!Send` and `!Sync` in order to allow
    // for future `!Send` and / or `!Sync` fields.
    _marker2: PhantomData<*mut ()>,
}

```

Questo tratto viene implementato dagli oggetti che descrivono una computazione asincrona.

- Tale computazione può essere ancora in corso o essere terminata.
- Un oggetto che implementa questo tratto è **inerte**. Affinchè la computazione proceda, occorre che qualcuno ne invochi il metodo **poll(..)**.

Il tratto offre un singolo metodo, **poll(..)**, che **implementa la logica della macchina a stati** associata alla funzione.

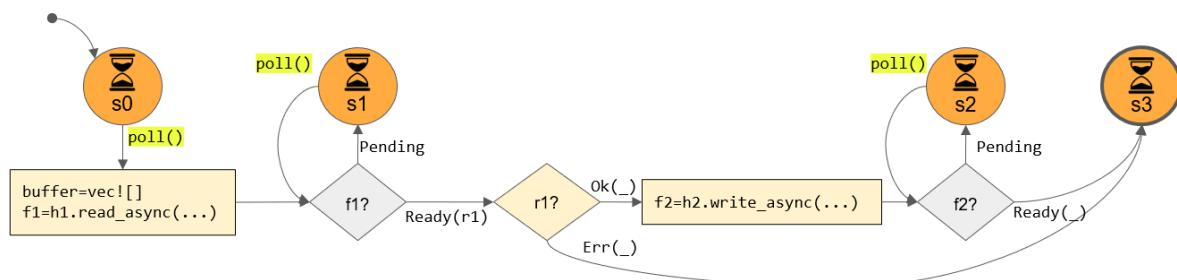
- L'oggetto che implementa il tratto mantiene al suo interno lo stato corrente.
- Restituisce una enum **Poll<T>** che può assumere due soli valori:
 - **Poll::Pending** - per indicare che la computazione è ancora in corso.
 - **Poll::Ready(val)** - per indicare che la computazione è terminata e ha come risultato **val**.

Generare la macchina a stati

```

async fn copy(h1: FileHandle, h2: FileHandle) -> std::io::Result<()> {
    let mut buffer = vec![];
    h1.read_async(&mut buffer).await?;
    h2.write_async(&buffer).await
}

```



Per ciascuno stato individuato, il compilatore sintetizza una struct in cui memorizzare le variabili locali necessarie per consentire l'esecuzione.

```
struct S0 {  
    h1: FileHandle,  
    h2: FileHandle,  
}
```

```
struct S1 {  
    h2: FileHandle,  
    buffer: Vec<u8>,  
    f1: impl Future<Output=Result<usize>>,  
}
```

```
struct S2 {  
    f2: impl Future<Output=Result<usize>>,  
}
```

```
struct S3 {}
```

Inoltre genera una enum che raccoglie i possibili stati e implementa il tratto Future.

```
enum CopySM {  
    s0(S0),  
    s1(S1),  
    s2(S2),  
    s3(S3)  
}
```

```
impl Future for CopySM {  
    type Output = std::io::Result<()>  
    fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Self::Output> {  
        loop {match self {  
            CopySM::s0(state) => { ... },  
            CopySM::s1(state) => { ... },  
            CopySM::s2(state) => { ... },  
            CopySM::s3(state) => { ... },  
        } }  
    }  
}
```

```
CopySM::s0(state) => {  
    let mut buffer = vec![];  
    let f1 = state.h1.read_async(&mut buffer);  
    let state = S1 {h2: state.h2, buffer, f1 };  
    *self = CopySM::S1(state);  
}
```

```
struct S0 {  
    h1: FileHandle,  
    h2: FileHandle,  
}
```

```

CopySM::s1(state) => {
    match (state.f1.poll(cx)) {
        Poll::Pending => return Poll::Pending,
        Poll::Ready(r1) =>
            if r1.is_ok() {
                let f2 = state.h2.write_async(&mut state.buffer);
                let state = S2{ f2 };
                *self = CopySM::s2(state);
            } else {
                *self = CopySM::s3(S3);
                return Poll::Ready(r1);
            }
    }
}

```

```

struct S1 {
    h2: FileHandle,
    buffer: Vec<u8>,
    f1: impl Future<Output=Result<usize>>,
}

```

```

CopySM::s2(state) => {
    match (state.f2.poll(cx)) {
        Poll::Pending => return Poll::Pending,
        Poll::Ready(r2) =>
            *self = CopySM::s3(S3);
            return Poll::Ready(r2);
    }
}

```

```

struct S2 {
    f2: impl Future<Output=Result<usize>>,
}

```

```

CopySM::s3(_) => {
    panic!("poll() was invoked again after Poll::Ready has been returned");
}

```

```
struct S3 {}
```

Il codice generato dal compilatore per la funzione si riduce all'inizializzazione della macchina a stati.

```

fn copy(h1: FileHandle, h2: FileHandle) ->
    impl Future<Output = std::io::Result<()>> {
    CopySM::s0(
        S0 { h1, h2 }
    )
}

```

Gestire l'esecuzione

Chi invoca una funzione asincrona, ottiene come risultato un **Future** nel proprio stato iniziale.

- Affinché possa capirne qualcosa, occorre che ne venga invocato il metodo **poll(..)**.

Se la funzione asincrona è chiamata all'interno di un'altra funzione asincrona, diventa automaticamente parte della macchina a stati del chiamante.

- Come succede per le funzioni `read_async(..)` e `write_async(..)` mostrate nell'esempio
- Sarà responsabilità del chiamante della funzione esterna gestire il **Future** risultante.

Se si invoca una funzione asincrona all'interno di una funzione "normale", occorre gestire il risultato di tipo **Future** in modo esplicito.

- Per farlo occorre disporre di un **Executor**

Se il compilatore supporta la generazione automatica dei tipi che implementano la macchina a stati associata ad una funzione asincrona e la riscrittura delle funzioni in modo opportuno, **nessun supporto** è invece offerto dal linguaggio **per la gestione dell'esecuzione**.

- Il programmatore può scegliere quale libreria adottare nel proprio progetto, in base alle specifiche necessità.
- Sono disponibili diverse librerie alternative per questo scopo:
 - **Tokio** - l'ambiente più diffuso, con supporto per connessioni di rete, database, etc..
 - **soml** - un ambiente semplificato, a basso impatto sulle risorse, adatto a sistemi embedded.
 - **async-std** - ambiente che offre la controparte asincrona delle librerie standard bloccanti.

I diversi ambienti di esecuzione non sono equivalenti.

- Tokio implementa un ciclo reattivo proprio, basato sul crate ad alte prestazioni **mio** (Metal I/O), che non è compatibile con i tratti usati dagli altri due.

Un runtime può basarsi su un singolo ciclo reattivo e/o utilizzare un thread-pool cui delegare l'esecuzione di più **Future** in parallelo.

- In questo caso, è possibile che l'elaborazione di una funzione asincrona inizi in un thread ma sia continuata in un thread differente.
- Questo implica che tutti i valori utilizzati nella funzione asincrona il cui uso si estende in più stati devono implementare il tratto **Send**, mentre i riferimenti devono implementare il tratto **Sync**.

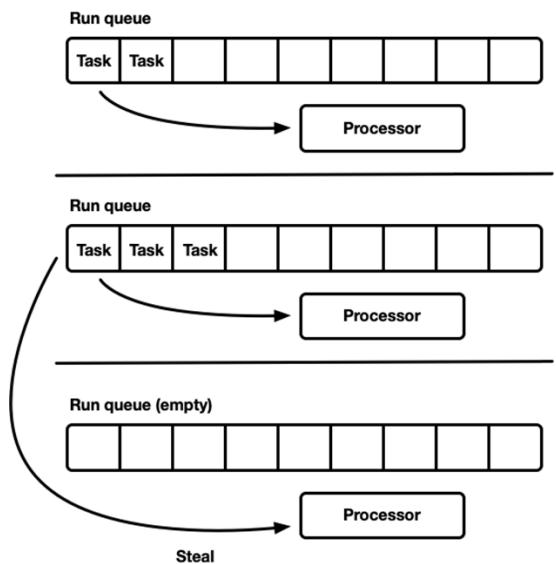
15.3 - Il framework Tokio

Tokio opera utilizzando un certo numero di code di esecuzione (default: n° di core).

- Ogni coda è gestita da un ciclo reattivo (Processor) che estrae ed esegue i task presenti.
- Quando un Processor esaurisce i task della propria coda, prova a rubarne alcuni ad altre code, su base euristica.

L'intero algoritmo è ottimizzato per ridurre al minimo la sincronizzazione.

- Prevalentemente usando oggetti atomici



Per impostare un progetto con Tokio, oltre ad aggiungere l'opportuna dipendenza nel file Cargo.toml, indicando anche quali funzionalità si intendono attivare per il crate, occorre strutturare il punto di ingresso del programma in modo opportuno.

```
[dependencies]
tokio = {version = "1.23.0", features = ["full"]}
```

```
#[tokio::main(flavor = "multi_thread", worker_threads = 4)] //o altro...
async fn main() {
    //... creazione di future e attesa relativa
}
```

```
tokio::task::spawn(f: T) → JoinHandle<T::Output>
    where T: Future + Send + 'static,
          T::Output: Send + 'static
```

- Funzione che **definisce un task**, eseguito in modo concorrente con altri task, la cui esecuzione inizia subito (a differenza di un Future di cui occorre invocare l'operatore `.await`)
- L'oggetto passato come parametro può essere il risultato dell'invocazione di una funzione `async`, o essere un blocco `async` passato per valore.

```
#[tokio::main]
async fn main() {
    let task = tokio::spawn(async { println!("Hello, Tokio!"); });
    task.await.unwrap()
}
```

La macro `join!(f1: impl Future, .., fn: impl Future)` può essere invocata all'interno di una funzione/blocco async e forza l'attesa fino a che tutti i suoi parametri non sono completati.

- Restituisce una tupla contenente i risultati associati ai suoi parametri.

Una versione semplificata, `try_join!(..)` può essere usata quando le espressioni passate come parametro hanno come valore di ritorno `Result<T, E>`.

- In questo caso viene restituito un oggetto `Result` che contiene, se tutti i `Future` hanno avuto successo, una tupla con i relativi risultati, oppure, in corrispondenza del primo fallimento, l'errore corrispondente.

```
async fn do_stuff_async() { ... }
async fn more_async_work() { ... }

#[tokio::main]
async fn main() {
    let (first, second) = tokio::join!(
        do_stuff_async(),
        more_async_work()
    );
    // do something with the values
}
```

La macro `select!(..)` permette di attendere su più rami asincroni, eseguiti nell'ambito dello stesso thread, quello che termina per primo, cancellando l'esecuzione dei restanti.

- Può essere usata solo all'interno di funzioni/blocchi asincroni
- Al suo interno è possibile inserire condizione della forma:
`<pattern> = <async expression> (, if <precondition>)? => <handle>
else => <expression>`
- Il ramo `else`, se presente, viene valutato solo se nessuno dei rami precedenti ha avuto successo.

```
async fn do_stuff_async() { ... }
async fn more_async_work() { ... }

#[tokio::main]
async fn main() {
    tokio::select! {
        _ = do_stuff_async() => {
            println!("do_stuff_async() completed first")
        }
        _ = more_async_work() => {
            println!("more_async_work() completed first")
        }
    };
}
```

Gestione del tempo

La funzione `tokio::time::sleep(d: Duration).await` sospende l'esecuzione del task corrente per un tempo pari alla durata indicata.

- Durante l'attesa non viene consumata nessuna risorsa, se non la memoria necessaria a descrivere l'oggetto Future corrispondente.
- Allo scadere del tempo, l'esecuzione procede normalmente.

La funzione `tokio::time::timeout(d: Duration, f: F).await` attende per un tempo massimo pari alla durata indicata che il Future passato come secondo parametro si completi e restituisce un valore di tipo `Result<T, Elapsed>`.

- Se l'esecuzione si completa in tempo, il risultato è positivo e contiene il valore restituito dal Future, altrimenti riporta un errore di tipo `Elapsed`.

Eseguire compiti computazionalmente intensi

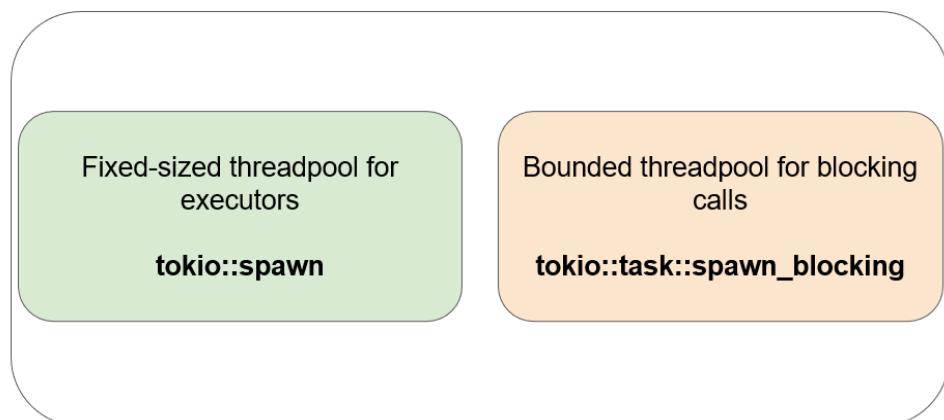
Se, in una funzione asincrona, occorre eseguire un compito computazionalmente intenso, la cui durata possa superare il centinaio di microsecondi, è bene richiedere che venga eseguito in un thread apposito.

- In questo modo si evita di introdurre latenza nell'elaborazione degli altri task.
- Si utilizza la funzione `tokio::task::spawn_blocking(f: FnOnce() → R)`

Occorre evitare di lanciare troppi task di questo tipo.

- Durante la loro esistenza, infatti, tenderebbero a richiedere l'uso di CPU introducendo contesa con i thread deputati ad elaborare le code dei messaggi, aumentando la latenza complessiva del sistema.
- Può essere opportuno condizionare l'esecuzione alla disponibilità di risorse globali (usando, ad esempio, un semaforo) o usare esecutori ad hoc, come quelli offerti dalla libreria **Rayon**

Tokio's Runtime



Condividere dati tra task

Se due task hanno bisogno di condividere una struttura dati, questa deve essere opportunamente protetta.

- A seguito del meccanismo di *work stealing* adottato dallo schedulatore, è infatti possibile che l'esecuzione avvenga in thread differenti.
- Occorre pertanto adottare le stesse precauzioni e strategie già viste con la programmazione multithread.

Tokio mette a disposizione una ricca serie di primitive asincrone nel modulo **tokio::sync**

- Alcune basate sulla condivisione dello stato (**Barrier**, **Mutex**, **Notify**, **RwLock**, **Semaphore**)
- Altre basate sulla comunicazione di messaggi (canali **oneshot**, **mpsc**, **broadcast**, **watch**)

Arc/Mutex - stato condiviso

```
use tokio::sync::Mutex;
use std::sync::Arc;
#[tokio::main]
async fn main() {
    let data = Arc::new(Mutex::new(0));
    let mut v = vec![];
    for _ in 0..4 {
        let data = Arc::clone(&data);
        v.push(tokio::spawn(async move {
            let mut lock = data.lock().await;
            *lock += 1;
        }));
    }
    for h in v { let _ = join!(h); }
    assert_eq!(*data.lock().await, 4);
}
```

Canali oneshot - Invio di un solo messaggio

```
async fn some_computation() -> String { "Some result".to_string() }

#[tokio::main]
async fn main() {
    let (tx, rx) = oneshot::channel();

    tokio::spawn(async move {
        let res = some_computation().await;
        tx.send(res).unwrap();
    });
}

// Do other work while the computation is happening in the background

// Wait for the computation result
let res = rx.await.unwrap();
}
```

Canali mpsc - Multiple Produce Single Consumer

```
async fn some_computation(i: u32) -> String { format!("Value {}", i) }

#[tokio::main]
async fn main() {
    let (tx, mut rx) = mpsc::channel(100);

    tokio::spawn(async move {
        for i in 0..10 {
            let res = some_computation(i).await;
            tx.send(res).await.unwrap();
        }
    });

    while let Some(res) = rx.await.unwrap() { println!("{}: {}", i, res); }
}
```

Canali broadcast - comunicazione multi-molti

```
#[tokio::main]
async fn main() {
    let (tx, mut rx1) = broadcast::channel(16);
    let mut rx2 = tx.subscribe();

    tokio::spawn(async move {
        assert_eq!(rx1.recv().await.unwrap(), 10);
        assert_eq!(rx1.recv().await.unwrap(), 20);
    });
    tokio::spawn(async move {
        assert_eq!(rx2.recv().await.unwrap(), 10);
        assert_eq!(rx2.recv().await.unwrap(), 20);
    });
    tx.send(10).unwrap();
    tx.send(20).unwrap();
}
```

Canali watch - pattern Observer

```
#[tokio::main]
async fn main() {
    let (tx, mut rx) = watch::channel("value 0");

    for i in 0..2 {
        let mut rx = rx.clone();
        tokio::spawn(async move {
            while rx.changed().await.is_ok() {
                println!("received: {:?}", *rx.borrow());
            }
        });
    }
    let d = Duration::from_secs(1);
    tx.send("value 1").unwrap(); tokio::time::sleep(d).await;
    tx.send("value 2").unwrap(); tokio::time::sleep(d).await;
}
```

Implementare un semplice server HTTP

```
use tokio::io::AsyncWriteExt;
use tokio::net::{TcpListener, TcpStream};
use tokio::task;
#[tokio::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1:8181").await.unwrap();
    loop {
        let (stream, _) = listener.accept().await.unwrap();
        tokio::spawn(handle_connection(stream));
    }
}

#[tokio::main]
async fn main() {
    let (tx, mut rx1) = broadcast::channel(16);
    let mut rx2 = tx.subscribe();
    tokio::spawn(async move {
        assert_eq!(rx1.recv().await.unwrap(), 10);
        assert_eq!(rx1.recv().await.unwrap(), 20);
    });
    tokio::spawn(async move {
        assert_eq!(rx2.recv().await.unwrap(), 10);
        assert_eq!(rx2.recv().await.unwrap(), 20);
    });
    tx.send(10).unwrap();
    tx.send(20).unwrap();
}
```

Prestazioni a confronto

I costi di creazione e cambiamento di contesto tra attività asincrone e thread sono analizzati nel sito <https://github.com/jimblandy/context-switch>.

- Il sito riporta la metodologia, il codice e i risultati ottenuti su un elaboratore specifico, con il relativo SO.
- Gli ordini di grandezza sono comunque interessanti (tabella)

Operazione	async	thread
Creazione di task	0.3 µs	17µs
Cambio di contesto	0.2 µs	1.7µs
Uso di memoria	300÷500 Byte	> 9.5 KByte

Dall'analisi di questi numeri emerge che l'uso di costrutti asincroni può ridurre l'utilizzo della memoria.

- Specialmente in questi contesti in cui si tenderebbe a preallocare un elevato numero di thread per poter fare fronte a carichi di lavoro improvvisi (server di rete)

È meno oneroso creare task asincroni rispetto a thread.

- Inoltre il tempo necessario a cambiare contesto è minore, ma solo nei casi migliori: se il cambio di contesto è dovuto alla disponibilità di dati su una periferica su cui si intende eseguire I/O, il costo è analogo.

Operazioni di I/O asincrone, che richiedano un'attesa, dovranno essere ritentate in seguito.

- Questo richiede l'esecuzione di (almeno) due system-call, invece che una sola come nel caso sincrono.