

Concorrenza

Eseguire più flussi allo stesso tempo

Programmazione concorrente

- Un programma concorrente dispone di due o più flussi di esecuzione contemporanei
 - Per perseguire un obiettivo comune
 - Tali flussi possono essere eseguiti in parallelo (se il processore dispone di più core) e/o alternarsi nel tempo, sotto il controllo di uno schedulatore
- Ogni processo è **isolato** rispetto agli altri e vive nel suo spazio di indirizzamento (quello che succede nel processo A non ha nessuna interferenza con il processo B)
 - isolamento complica la cooperazione tra processi
 - isolamento imperfetto: possibili interferenze con il file system o con il sistema di rete
- All'atto della creazione, un processo dispone di un **unico flusso di esecuzione**
 - Thread principale
 - Esso può richiedere allo schedulatore (con richiesta di intervento del **Sistema Operativo**) la **creazione di altri thread**
- Un thread rappresenta una **computazione indipendente**
 - Basata su un proprio stack (pre-allocato all'atto della creazione del thread) collocato nello stesso spazio di indirizzamento in cui operano gli altri thread del processo (**condivisione** dello spazio di indirizzamento)
 - Tale computazione si svolge fino al proprio termine, potendo dare origine ad un risultato o ad un errore

Programmazione concorrente

- Il S.O. e/o le librerie di supporto allocano le risorse fisiche necessarie
 - Lo scheduler ripartisce, nel tempo, l'utilizzo dei core disponibili tra i diversi thread in modo non deterministico
 - Tutti i thread creati sono identificati in modo univoco e viene mantenuto, per quelli in uso, l'indicazione del loro stato di esecuzione
- La gestione dei thread può essere demandata direttamente al S.O.
 - In questo caso si parla di **thread nativi**
- ...oppure gestita da librerie a livello utente, con il supporto parziale del sistema operativo
 - Questi vengono definiti **green thread** o **fibre** e richiedono una certa forma di cooperazione da parte del codice che deve, in modo esplicito, invocare lo scheduler per cedere l'uso della CPU
- C++ e Rust offrono, nella propria libreria standard, supporto per i thread nativi
 - Entrambi offrono librerie di terze parti per il supporto di green thread

Thread nativi

- I diversi sistemi operativi offrono funzionalità simili (ma non identiche) per governare l'interazione di un programma con l'insieme dei thread che lo costituiscono
- Le funzioni supportate sono:
 - **Creazione** di un thread, indicando la funzione che rappresenta la computazione che deve essere svolta e la dimensione dello stack richiesto: questa operazione restituisce un **handle** opaco mediante il quale fare riferimento al thread
 - **Identificazione** del thread corrente, sotto forma di valore univoco a livello di sistema (TID)
 - **Attesa** della terminazione di un thread, a partire dalla sua handle, e accesso al suo stato finale (successo/fallimento): operazione **join**
- Tra le funzioni **non supportate**, spicca la richiesta di **cancellazione** di un thread
 - Questa può solo essere implementata in modo cooperativo dal thread stesso

Cosa implica la concorrenza

- **Riduzione del sovraccarico** dovuto alla comunicazione tra processi
 - Sebbene la suddetta parallelizzazione possa essere fatta anche creando processi separati, il costo di comunicazione e sincronizzazione tra processi è sensibilmente più alto di quello tra thread
 - I thread condividono infatti lo spazio di indirizzamento ed è possibile trasferire la proprietà di strutture dati da un thread ad un altro semplicemente comunicando il puntatore
 - Per ottenere un effetto analogo tra processi differenti, sarebbe necessario serializzare la struttura dati presente nel processo originale, trasferire una copia della rappresentazione ottenuta nel processo destinazione e qui ricostruire una copia della struttura dati

Cosa implica la concorrenza

- Possibilità di **sovrapporre temporalmente** attività di computazione e operazioni di I/O
 - I sistemi operativi tendono ad offrire API bloccanti che arrestano, di fatto, la prosecuzione di un thread fino a che il dato richiesto non è pronto (es.: `read(fd)`, `accept(socket)`, ...)
 - Suddividendo l'algoritmo in più thread, si può cercare sfruttare i tempi di attesa che un dato thread subisce a seguito delle operazioni di I/O per eseguire, in altri thread, operazioni utili al risultato
 - Occorre che la complessità aggiunta dalla suddivisione sia compensata da un effettivo guadagno in termini di tempi di esecuzione
- Regola generale:
 - Si usano i thread quando si devono *fare delle cose* insieme
 - Si usa la programmazione asincrona quando si deve *aspettare* insieme

Cosa implica la concorrenza

- Possibilità di sfruttare appieno le capacità di **elaborazione** delle CPU **multicore**
 - Vero parallelismo
 - Più flussi di esecuzione possono svolgersi contemporaneamente, riducendo così il tempo totale di elaborazione
- **Aumento** significativo **della complessità** del programma
 - Nuove fonti e tipologie di errore
 - **Non determinismo dell'esecuzione**
- La memoria **non** può più essere pensata come un "**deposito statico**"
 - I dati scritti al suo interno possono cambiare in conseguenza dell'attività di altri thread
- I thread devono **coordinare l'accesso** alla memoria
 - Tramite opportuni costrutti di sincronizzazione
 - La presenza di cache legate ai singoli core introduce non determinismo nell'ordinamento e nella visibilità delle azioni sulla memoria

Concorrenza in pratica

- Se, all'interno di un processo, sono presenti due o più thread, questi possono procedere indipendentemente nella propria computazione
 - Sebbene sia possibile creare thread che si ignorano reciprocamente e non necessitano alcuno scambio di informazione, sul piano pratico questo avviene molto raramente
- L'utilità di suddividere la computazione globale in più sotto-computazioni nasce, per lo più, dal fatto che ciascuna di esse contribuisce in qualche modo al risultato finale
 - Questo richiede che esista una forma di comunicazione/sincronizzazione tra thread differenti
- I meccanismi soggiacenti alla comunicazione/sincronizzazione interferiscono con le ottimizzazioni usate dai processori per migliorare l'esecuzione
 - Introducendo una serie di **complessità inattese** e lontane dal pensiero comune legato al modello di esecuzione sequenziale

Concorrenza in pratica

- In un sistema single-core, il concetto di thread è puramente una astrazione offerta dal sistema operativo
 - Il processore si limita ad alternare il proprio ciclo di esecuzione basato sulla successione delle micro-operazioni **fetch/decode/execute**, procedendo di istruzione in istruzione secondo la logica del codice macchina
- Il sistema operativo può intervenire in questa sequenza, grazie ad un'interruzione che attiva lo scheduler
 - Questo salva lo stato dei registri in una qualche area di memoria dedicata alle meta-informationi del thread corrente e li ripristina con il contenuto relativo ad un thread differente (**task switching**)
- Se la CPU è dotata di due o più core, non cambia molto
 - Ciascun core procede indipendentemente dagli altri e lo scheduler provvede a gestire le attività di tutti, allocando di volta in volta i core disponibili ad eseguire l'uno o l'altro thread, secondo le necessità

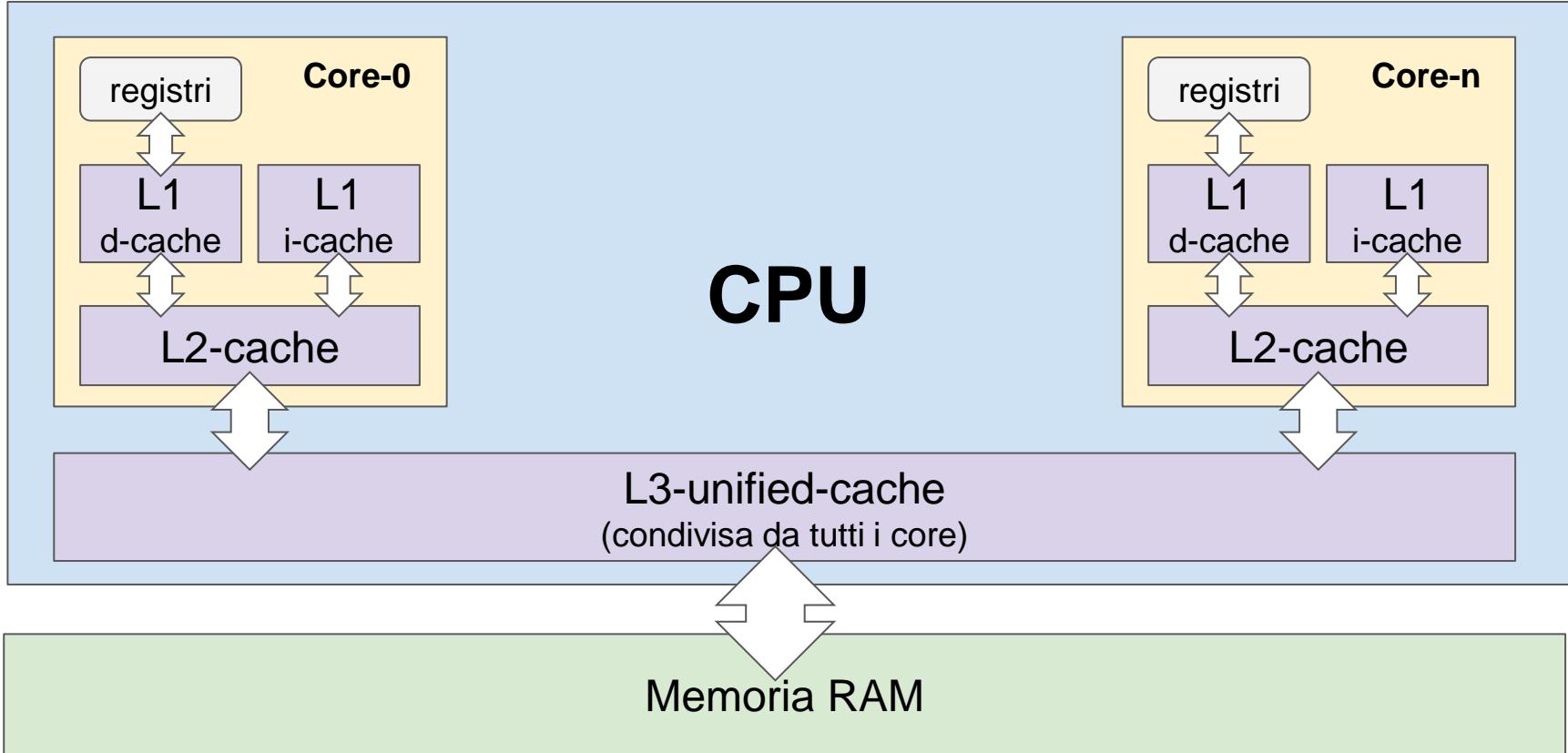
Concorrenza in pratica

- Poiché l'esecuzione di ciascun thread procede indipendentemente da quelle degli altri, un'eventuale necessità di comunicazione deve essere soddisfatta passando per l'utilizzo di un'area di memoria condivisa
 - In cui il thread T1 possa depositare le informazioni che intende comunicare al thread T2
- Sebbene i due thread utilizzino lo stesso spazio di indirizzamento e possano, in linea di principio, accedere al dato memorizzato, questa operazione risulta **più complessa** di quanto si possa pensare a prima vista
 - Per rendere la comunicazione utile sul piano pratico, può essere inoltre necessario avvalersi di pattern di interazione che definiscano con precisione i ruoli che le due o più parti coinvolte possono giocare

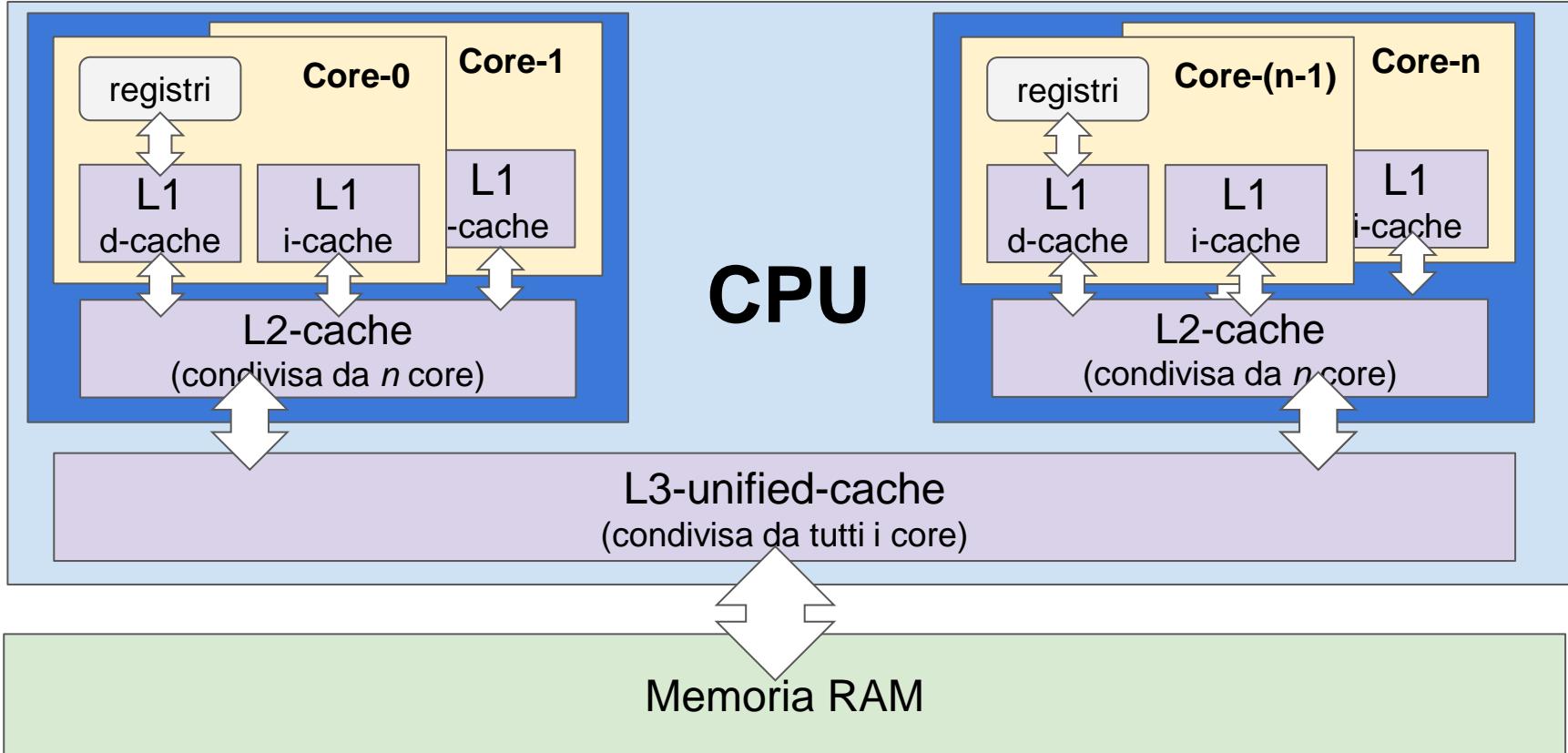
Modello di memoria

- Quando un thread legge il contenuto di una locazione di memoria, può trovare:
 - Il valore iniziale contenuto nel file 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 che permetta di definire l'ordine di esecuzione

Modello di memoria



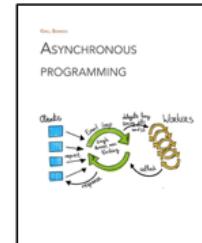
Modello di memoria (packed cores)



Tempi di accesso

System event	Actual Latency	Scaled Latency
One CPU cycle	0.4 ns	1 s
Level 1 cache access	0.9 ns	2 s
Level 2 cache access	2.8 ns	7 s
Level 3 cache access	2.8 ns	1 min
Min memory access (DDR DIMM)	~100 ns	4 min
Intel Optane DC persistent memory access	~350 ns	15 min
Intel Optane DC SSD I/O	< 10 µs	7 hrs
NVMe SSD I/O	~25 µs	17 hrs
SSD I/O	50-150 µs	1.5-4 days
Rotational disk I/O	1-10 ms	1-9 months
Internet SF to NYC	65 ms	5 years

da "Asynchronous programming" di Kirill Bobrov, settembre 2020



Problemi aperti

- Il problema è principalmente sulle variabili globali e su quelle istanza, meno sulle variabili locali (a meno che il loro indirizzo sia noto ad altri thread)
- **Atomicità**
 - Quali istruzioni devono avere effetti indivisibili?
 - o si vede che cosa c'era prima o si vede che cosa ci sarà dopo, ma non si vede il durante
- **Visibilità**
 - Sotto quali condizioni, le scritture compiute da un thread sono visibili da un secondo thread?
 - Problema legato alla cache: le scritture che un thread fa devono arrivare fino alla memoria principale per diventare visibili a tutti
- **Ordinamento**
 - Sotto quali condizioni gli effetti di più operazioni effettuate da un thread possono apparire ad altri thread in ordine differente?
 - Può succedere che aggiornamenti diversi abbiano dei tempi di propagazioni differenti e dunque la loro visibilità non mantenga lo stesso ordine.

Le risposte dei processori

- Ciascuna famiglia di processori offre una propria risposta ai problemi menzionati
 - La piattaforma **x86** adotta un modello **quasi sequenzialmente consistente** ed offre le istruzioni di tipo *fence*, che forzano il completamento delle operazioni di scrittura, bloccando temporaneamente gli altri core (e invalidando le loro cache) che dovessero cercare di accedere nel frattempo allo stesso segmento di indirizzi
 - Per contro, sulla piattaforma **ARM** viene usato un modello molto più lasco, basato su **liste di propagazione dei cambiamenti**, ed offre le istruzioni di tipo *memory barrier* che consentono l'ordinamento causale rispettivamente per il calcolo degli indirizzi, delle istruzioni, dei dati
 - Barrier di lettura, quando voglio leggere da una cella condivisa posso mettere una dipendenza di lettura e voglio che la cache sia invalidata prima di andare a leggere, in modo da essere sicuro di leggere dalla memoria principale
 - Barrier di scrittura, quando voglio scrivere voglio essere sicuro che il valore sia trasferito a destinazione e sia dunque flushed in memoria
 - Barrier completa, invalido la cache prima di leggere e faccio il flush dopo la scrittura
- Se tali istruzioni non vengono incluse all'interno del codice generato, **non è garantito un ordinamento predicibile** alle operazioni di **lettura e scrittura di dati condivisi**, in presenza di attività concorrenti
 - C++ e Rust condividono il modello di memoria su cui sono basati e annegano, nelle funzioni di libreria dei tipi dedicati alla concorrenza, tali istruzioni allo scopo di garantire le necessarie proprietà di funzionamento

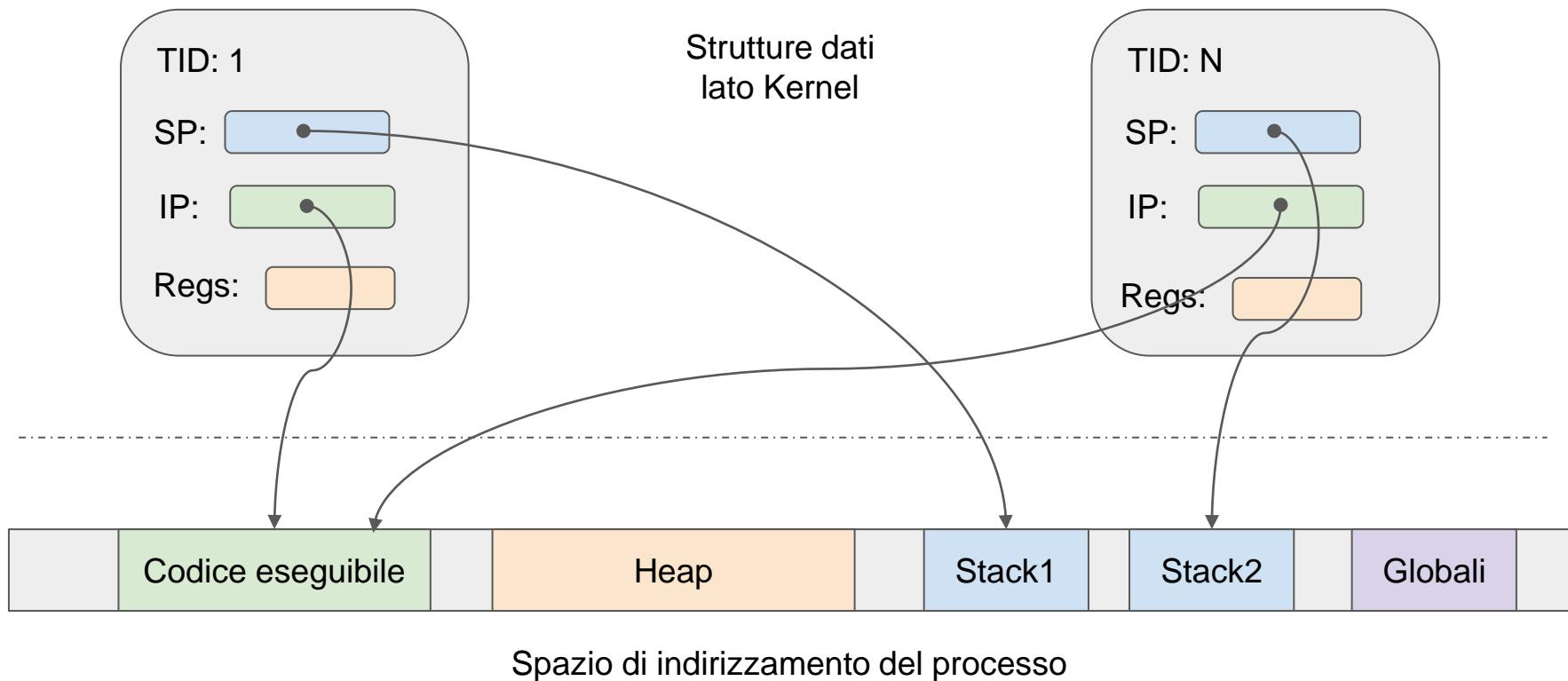
Errori

- **L'uso superficiale** dei costrutti di sincronizzazione porta a blocchi passivi (che si aspettano reciprocamente) o attivi (non riesco a smettere perché non esiste la possibilità di informarmi che devo smettere) del programma...
 - In ogni caso, fonte di guai per il programmatore
- **L'assenza** di costrutti di sincronizzazione, porta a risultati imprevedibili
 - Anche molto lontani da quanto sarebbe logico aspettarsi
- Si possono verificare **malfunzionamenti casuali**
 - Dovuti al comportamento non deterministico e asincrono dell'esecuzione concorrente
 - Estremamente difficili da riprodurre e da eliminare
- Gli errori possono manifestarsi cambiando la **piattaforma di esecuzione**
 - Oppure soltanto dopo numerose esecuzioni
 - Tipicamente emergono nel momento meno adatto

Thread e memoria

- Il sistema operativo mantiene, al proprio interno, una rappresentazione dei thread presenti all'interno di un dato processo
 - Ad ogni thread viene associato un identificativo univoco (TID - Thread ID), il suo stato di esecuzione (non schedulabile, schedulabile, in esecuzione sul core i, terminato con errore, terminato senza errore, ...) e le informazioni necessarie a salvare/ripristinare lo stato dei registri interni al processore
 - Provvede inoltre ad allocare, nello spazio di indirizzamento del processo, un blocco di indirizzi contigui destinato ad ospitare lo stack che governerà la sua esecuzione
- Tutti i thread presenti in un processo condividono:
 - Le variabili globali
 - Le costanti
 - L'area eseguibile in cui è contenuto il codice
 - Lo heap

Thread e memoria



Esecuzione e non determinismo

- L'esecuzione di ogni singolo thread procede secondo le normali regole sequenziali
 - Per cui è possibile prevedere cosa avvenga "prima" e cosa "dopo"
- Se più thread sono in esecuzione, non è possibile fare assunzioni sulle velocità relative di avanzamento
 - Se non ricorrendo a forme esplicite di sincronizzazione e comunicazione
- La sincronizzazione può riguardare il raggiungimento di un particolare stato da parte di un thread...
 - Abilitando di conseguenza altri a **procedere**
- ...oppure l'esigenza di un thread di eseguire azioni su aree condivise
 - Allo scopo di **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

Esecuzione e non determinismo

```
#include <thread>
#include <iostream>
#include <string>
using namespace std;
void run(string msg) {
    for (int j=0; j<10; j++) {
        cout << msg << to_string(j) << "\n";
        this_thread::sleep_for(chrono::nanoseconds(1));
    }
}
int main() {
    thread t1(run, "aaaa");
    thread t2(run, "bbbb");
    t1.join();
    t2.join();
}
```

C++

aaaa0
bbbb0
aaaa1
bbbb1
aaaa2
bbbb2
aaaa3
bbbb3
aaaa4
bbbb4
aaaa5
aaaa6
aaaa7
bbbb5
aaaa8
aaaa9
bbbb6
bbbb7
bbbb8
bbbb9

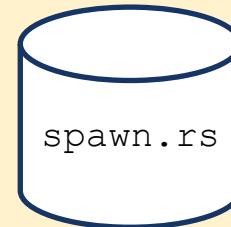
Esecuzione e non determinismo

```
use std::thread;
use std::time::Duration;

fn run(msg: &str) {
    for i in 0..10 {
        println!("{}{}", msg, i);
        thread::sleep(Duration::from_nanos(1));
    }
}

fn main() {
    let t1 = thread::spawn(|| {run("aaaa"); });
    let t2 = thread::spawn(|| {run("bbbb"); });
    t1.join().unwrap();
    t2.join().unwrap();
}
```

Rust



aaaa0
aaaa1
aaaa2
bbbb0
bbbb1
bbbb2
aaaa3
aaaa4
aaaa5
aaaa6
aaaa7
bbbb3
bbbb4
bbbb5
aaaa8
aaaa9
bbbb6
bbbb7
bbbb8
bbbb9

Esecuzione e non determinismo

- L'output dei programmi precedenti è **solo uno** dei possibili risultati
 - Se lo stesso programma viene eseguito più volte, si ottengono risultati differenti
- È assolutamente possibile (e a volte succede) che tutte le righe di uno dei thread precedano quelle dell'altro
 - In base al numero di core disponibili e alla durata del "quanto" di schedulazione adottato dai sistemi operativi
- L'unica certezza è che le righe che cominciano con "aaaa" sono tra loro ordinate in modo crescente
 - Così come le righe che cominciano con "bbbb"

Esecuzione e non determinismo

- I thread, nel programma precedente, non hanno punti di contatto, ma qualora vi siano punti di contatto (ossia variabili condivise) il **non determinismo** può dare origine a **comportamenti del tutto inattesi**
 - Questo può essere visto facilmente in C/C++, dove l'assenza di restrizioni da parte del borrow checker, non obbliga il programmatore ad avere cura degli aspetti di sincronizzazione
 - Per contro, **Rust si fa garante che un'intera gamma di possibili errori non possano verificarsi**

Esecuzione e non determinismo

```
#include <iostream>                                         C++
#include <thread>

int a = 0;          //Questo non è possibile in safe RUST

void run() {
    while (a >= 0) {
        int before = a;
        a++;
        int after = a;
        if (after-before != 1)
            std::cout << before << " -> " << after
                           << "(" << after-before << ")\\n";
    }
}
```

Esecuzione e non determinismo

C++

```
//creo due thread e ne attendo la terminazione

int main() {
    std::thread t1(run);
    std::thread t2(run);

    t1.join();
    t2.join();
}
```

Esecuzione e non determinismo

```
119944578 -> 119944552 (-26)
123397102 -> 123397584 (482)
128314912 -> 128314956 (44)
395835151 -> 395835236 (85)
396049424 -> 396098482 (49058)
412859791 -> 412859826 (35)
419214490 -> 419214537 (47)
419406880 -> 419406877 (-3)
433982464 -> 433982472 (8)
436005364 -> 436215900 (210536)
441453011 -> 441454010 (999)
446802106 -> 446802106 (0)
```

Domande

- Esaminando l'uscita del programma precedente, si vedono molti casi in cui la differenza tra **after** e **before** è superiore a **1**
 - In alcuni casi tale valore è anche molto grande: perché?
- Talora capita che la differenza sia **nulla** o **negativa**
 - Come è possibile, se entrambi i flussi incrementano sempre la variabile **a**?

????

Interferenza

- Si verifica quando più thread fanno accesso a uno stesso dato, **modificandolo**
- La sua presenza dà origine a **malfunzionamenti casuali**, molto difficili da identificare



Istruzione di tipo Read-Modify-Write non atomica

```
#include <iostream>
#include <thread>

int a=0;

void run() {
    while (true) {
        int before=a;
        a++;
        if (after-before!=1)
            std::cout<< before<< " -> " << after<< "("
                           << after-before<<")\n";
    }
}
```

Sembra un'azione
innocente, ma nasconde
due operazioni in
cascata:
int temp = a;
a = temp+1;

Thread 1

- temp = 900

```
void run() {  
    while (a >= 0) {  
        int before = a;  
        temp = a;  
        a = temp + 1;  
        int after = a;  
        ...  
    }  
}
```

Riprende da qua

Thread 2

- before = 1000

```
void run() {  
    while (a >= 0) {  
        int before = a;  
        temp = a;  
        a = temp + 1;  
        int after = a;  
        ...  
    }  
}
```

Riprende da qua

Alla fine del thread 1

a = 950

Temp = 950
After < before

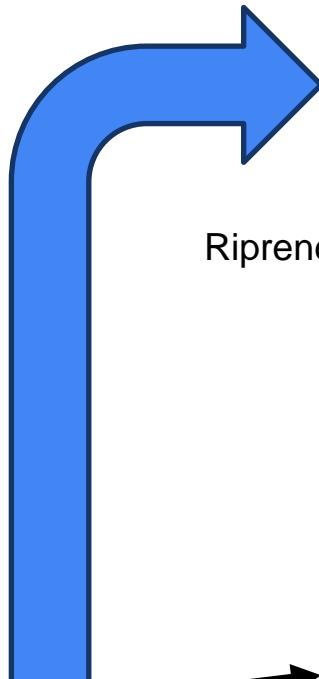
Thread 1

- $\text{temp} = 1000$

Riprende da qua

```
void run() {
    while (a >= 0) {
        int before = a;
        temp = a;
        a = temp + 1;
        int after = a;
        ...
    }
}
```

Alla fine del thread 1
 $a = 1100$



Thread 2

- $\text{before} = 900$

Riprende da qua

```
void run() {
    while (a >= 0) {
        int before = a;
        temp = a;
        a = temp + 1;
        int after = a;
        ...
    }
}
```

$\text{Temp} = 1100$
 $\text{After} > \text{before} + 1$

Sincronizzazione

- Se due thread cercano di accedere in lettura/scrittura ad una stessa variabile si verifica una **corsa critica** (*data race*)
 - In base a condizioni non controllabili dal programmatore (come la presenza di memoria cache, il momento in cui avviene un task switch, le ottimizzazioni fatte dai singoli processori con la predizione della prossima istruzione da eseguire, ...) 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 e 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

Sincronizzazione

- La certezza che l'operazione di accesso corrisponde alla variabile in memoria è garantito da **apposite istruzioni macchina** (*fence/barrier*)
 - Che dipendono dal processore responsabile dell'esecuzione del codice
- La certezza che l'accesso ai dati sia effettuato solo da un thread alla volta è garantito da **invarianti a livello sistema** che può essere fornita solo da meccanismi offerti dal sistema operativo (*mutex* e *condition variable*) che, controllando la schedulazione dei thread, può farsi carico che avvenga / non avvenga 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
- Le librerie standard dei diversi linguaggi di programmazione si fanno (talora) carico di standardizzare tale comportamento, offrendo API comuni a livello di codice sorgente
 - Come nei casi di C++11 e successivi e di Rust

Sincronizzazione

- La certezza che l'operazione di accesso sia garantita da **apposite istruzioni macchina** (fence/fenceq)
 - Che dipendono dal processore responsabile
- La certezza che l'accesso ai dati sia garantito da **invarianti a livello sistema** che può essere fornita solo da meccanismi offerti dal sistema operativo (*mutex* e *condition variable*) che, controllando la schedulazione dei thread, può farsi carico che avvenga / non avvenga 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
- Le librerie standard dei diversi linguaggi di programmazione si fanno (talora) carico di standardizzare tale comportamento, offrendo API comuni a livello di codice sorgente
 - Come nei casi di C++11 e successivi e di Rust

Mutex genera la barriera e garantisce che entrando nel blocco di codice venga invalidata la linea di cache corrispondente alla variabile condivisa e quando esco da quel blocco di codice venga fatta la flush della cache

Sincronizzazione

- La certezza che l'operazione di accesso corrisponda alle **apposite istruzioni macchina** (*fence/barrier*)
 - Che dipendono dal processore responsabile dell'esecuzione
- La certezza che l'accesso ai dati sia effettuato secondo **invarianti a livello sistema** che può essere fornita solo da meccanismi offerti dal sistema operativo (*mutex e condition variable*) che, controllando la schedulazione dei thread, può farsi carico che avvenga / non avvenga 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
- Le librerie standard dei diversi linguaggi di programmazione si fanno (talora) carico di standardizzare tale comportamento, offrendo API comuni a livello di codice sorgente
 - Come nei casi di C++11 e successivi e di Rust

Condition Variable fa attendere un thread fino a quando una condizione non si verifica senza consumare CPU e, appoggiandosi ad un mutex garantiscono una barriera di protezione e coerenza delle memorie.

Strutture native di sincronizzazione

Windows



- Strutture dati utente
 - **CriticalSection**
 - **SRWLock**
 - **ConditionVariable**
- Oggetti kernel
 - **Mutex**
 - **Event**
 - **Semaphore**
 - **Pipe**
 - **Mailslot**
 - ...

Linux



- Strutture dati utente
 - **pthread_mutex**
 - **pthread_cond**
- Oggetti kernel
 - **Semaphore**
 - **Pipe**
 - **Signal**
 - **Futex**



Correttezza

- Occorre fare in modo che **non capiti mai** che un thread "operi" su un dato, alterandone il contenuto
 - Mentre un altro sta già operando sullo stesso oggetto
- In particolare, non devono essere visibili **stati transitori** dell'oggetto
 - Dovuti al meccanismo di aggiornamento in cui solo una parte dell'informazione contenuta è cambiata
- Tutti gli oggetti condivisi mutabili devono godere di questa proprietà
 - Questi mantengono al proprio interno degli "invarianti" definiti a livello applicativo
 - Perché gli oggetti immutabili non sono soggetti a interferenza?
- Bisogna impedire che gli invarianti siano violati
 - Si effettuano le mutazioni (cambi di stato) con metodi che garantiscono la validità degli invarianti prima e dopo l'esecuzione e che bloccano 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



Correttezza

- In quasi tutti i linguaggi di programmazione, è compito del programmatore riconoscere **quando, dove e come** utilizzare la sincronizzazione
 - Un uso sbagliato porta a **risultati disastrosi**
- Occorre dimostrare la **correttezza formale** dell'algoritmo complessivo
 - Garantendo che, qualunque sia l'ordine di esecuzione scelto dal sistema operativo o la latenza introdotta dai sottosistemi di memoria, il programma resta corretto
 - Questo è possibile grazie alla presenza di primitive offerte dal sistema operativo / compilatore / librerie di esecuzione che garantiscono alcuni **invarianti di basso livello**
- Se un programma non è corretto, è inutile pensare di testarlo o ottimizzarlo
 - L'esito dei test è non deterministico: potrebbero passare per pura combinazione fortuita (o anche probabile) delle condizioni al contorno, ma non danno alcuna garanzia
 - Le ottimizzazioni non farebbero che esacerbare il problema, rendendone più difficile la comprensione



Correttezza

- 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 e replicare in errori di compilazione
 - Questo ha portato a definire questo aspetto di Rust come **fearless concurrency**
- Resta comunque responsabilità del programmatore la comprensione dell'algoritmo e la garanzia di terminazione dello stesso, così come la verifica dell'assenza di blocchi attivi e passivi che possono impedire il procedere dell'algoritmo

Accesso condiviso: i possibili problemi

- **Atomicità:** quali operazioni di memoria hanno effetti indivisibili?
 - Se due thread fanno **accesso alla stessa struttura dati**, rispettivamente in lettura e scrittura...
 - ...non c'è nessuna garanzia su quale delle due operazioni sia **eseguita per prima**
- **Visibilità:** la scrittura di una variabile può essere osservata da una lettura eseguita da un altro thread?
 - Se un thread legge un dato che un altro thread sta modificando...
 - ...il valore letto può essere **diverso** sia dal valore **iniziale** che da quello **finale**
- **Ordinamento:** sotto quali condizioni, sequenze di operazioni effettuate da un thread sono visibili nello stesso ordine da parte di altri thread?
 - Se, quando **osservato dall'esterno**, il comportamento di un singolo thread appare indistinguibile a seguito di una modifica alla sequenza delle istruzioni...
 - ...sia il compilatore che la CPU possono **invertire l'ordine di esecuzione** delle singole istruzioni

Accesso condiviso: le possibili soluzioni

- **Tipi Atomic**
 - Alla base di tutti i meccanismi di accesso condiviso ci sono istruzioni apposite, offerte dai singoli processori, volte a garantire operazioni di tipo Read-Modify-Write di tipo atomico (cioè, non interrompibili e non osservabili nei loro stati intermedi), su CPU single- e multi-core
 - Tali operazioni sono limitate a tipi semplici (booleani, interi, puntatori) e sono esposte dalle librerie standard di C++ e Rust attraverso opportune astrazioni, che incapsulano l'utilizzo di barriere di memoria
- **Mutex**
 - Per estendere le garanzie di atomicità e dipendenza causale a strutture dati più complesse, occorre introdurre il concetto di Mutex
 - Essi estendono il principio della mutua esclusione a thread differenti
 - Un mutex può essere libero o posseduto da un singolo thread
 - Se un secondo thread cerca di ottenere il possesso del mutex mentre è in uso da parte di 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 - senza consumare cicli di CPU - che si verifichi una condizione più complessa del semplice rilascio di un mutex da parte di un thread
 - 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
 - Una condition variable può essere usata solo in coppia con un mutex

Uso dei thread

- Le API dei S.O. permettono la gestione del ciclo di vita dei thread
 - Creazione e terminazione di thread
 - Meccanismi di sincronizzazione
 - Aree private di memoria
- I dettagli relativi a ciascuna piattaforma differiscono alquanto
 - Rendendo complessa la portabilità delle applicazioni
- La versione 2011 del linguaggio C++ ha introdotto una standardizzazione nella creazione e gestione dei thread
 - Tale standardizzazione, tuttavia, nello sforzo di uniformare i comportamenti, nasconde le peculiarità offerte dai singoli sistemi operativi per gestire i casi particolari connessi alla computazione, come cancellazione e fallimento
- Una standardizzazione analoga è offerta dalla libreria standard di Rust
 - Con una maggiore attenzione alla gestione del fallimento

Thread in C++

- La classe `std::thread` offre il supporto per la creazione di un thread nativo e la gestione del suo ciclo di vita
 - Il costruttore di tale classe accetta come parametro un oggetto callable (puntatore a funzione, oggetto funzionale, funzione lambda) e eventuali ulteriori parametri da passare a tale oggetto
 - Quando il costruttore ritorna, è presente il thread nativo creato al suo interno si trova nello stato `Runnable`, e può essere considerato schedulabile a tutti gli effetti
- L'oggetto thread inizializzato mantiene il riferimento opaco (handle) al thread nativo
 - Si può attendere la terminazione del thread nativo invocando il metodo bloccante `join()`
 - Oppure si può disgiungere l'oggetto dal thread nativo, invocando il metodo `detach()`
- Il distruttore dell'oggetto verifica, quando viene invocato, che il thread sia effettivamente terminato o sia stato distaccato
 - Se nessuna delle due opzioni è verificata, l'intero processo viene arrestato invocando la funzione `std::terminate()`
- Se la computazione svolta dal thread genera un'eccezione non gestita all'interno del thread stesso, l'intero processo viene terminato
 - Tramite la funzione `std::terminate()`

Thread in Rust

- Si crea un thread nativo in Rust attraverso la funzione `std::thread::spawn(...)`
 - Essa 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 (ovvero il tipo ritornato dalla funzione 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, nell'opzione `Ok`, il valore finale e nell'opzione `Err` il valore eventualmente passato alla macro `panic!`, nel caso in cui sia stata invocata nel corso della computazione del thread stesso
- Non si crea nessun rapporto di parentela tra thread creatore (quello in cui si invoca `spawn(...)`) e thread creato
 - Né occorre che l'uno sopravviva all'altro
 - Quando l'handle di un thread esce dello scope e viene rilasciata, non c'è più modo di avere notizie (dirette) sull'esito del thread creato, che acquisisce lo stato `detached`
- La standard library di Rust assegna ad ogni thread un identificatore accessibile attraverso la chiamata `thread::current().id()`

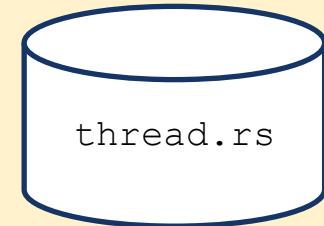
```
use std::thread;

fn main() {
    let t1 = thread::spawn(f);
    let t2 = thread::spawn(f);

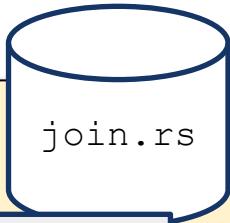
    println!("Hello from the main thread.");
    t1.join().unwrap();
    t2.join().unwrap();
    println!("End.");
}

fn f() {
    println!("Hello from another thread!");

    let id = thread::current().id();
    println!("This is my thread ID: {id:?}");
}
```



Output Locking: la macro `println!` utilizza la funzione `std::io::Stdout::Lock()` per garantire che la stringa stampata non sia interrotta.



join.rs

```
use std::thread;

fn main() {
    let data = vec![1, 2, 3, 4];
    // provare con
    // let data = vec![];
    // e vedere che cosa succede

    // Creiamo un nuovo thread e trasferiamo il possesso del vettore "data"
    let handle = thread::spawn(move || {
        let len: usize = data.len();
        let somma: usize = data.iter().sum();
        somma / len
    });

    println!("Il thread principale aspetta");
    let average = handle.join();
    match average {
        Ok(res) => {println!("La media è {:?}", res);},
        Err(err) => {println!("Errore {:?}", err);}
    }
    println!("Il thread principale termina.");
}
```

Cattura per movimento delle variabili che diventano di proprietà del thread

```
use std::thread;
use std::time::Duration;

fn main() {
    // Creiamo un nuovo thread
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("Ciao numero {} dal thread creato!", i);
            thread::sleep(Duration::from_millis(1));
        }
        // Restituiamo un valore dal thread
        "OK".to_string()
    });

    for i in 1..5 {
        println!("Ciao numero {} dal thread principale!", i);
        thread::sleep(Duration::from_millis(1));
    }

    // Attendiamo che il thread creato finisca e otteniamo il valore di ritorno
    match handle.join() {
        Ok(res) => {println!("Terminazione corretta {}", res);},
        Err(err) => {println!("Terminazione errata {:?}", err);}
    }
}
```



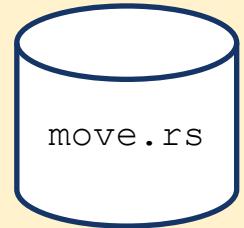
```
use std::thread;

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

    // Creiamo un nuovo thread e trasferiamo il possesso del vettore "data"
    let handle = thread::spawn(move || {
        let somma: i32 = data.iter().sum();
        println!("Ecco un vettore: {:?} la cui sommatoria è {}", data, somma);

    });
    println!("Il thread principale non ha accesso ai dati !");
    // println!("Ecco un vettore: {:?}", data);

    // Attendiamo che il thread creato finisca
    match handle.join() {
        Ok(_) => {println!("Terminazione corretta");},
        Err(err) => {println!("Terminazione errata {:?}", err);}
    }
}
```



Configurare un thread

- E' possibile configurare un thread prima di lanciare la sua esecuzione tramite la struct **std::thread::Builder**
 - Permette di assegnare al thread un nome a scelta 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>**

```
use std::thread;

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

let handler = builder.spawn(|| { /* codice */});

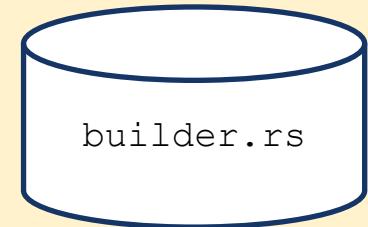
handler.unwrap().join().unwrap();
```

```
fn main() {
    use std::thread;

    let mut a = vec![1, 2, 3];
    let mut x = 0;

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

    let handler = builder
        .spawn( move || {
            a.push(4);
            x = a.iter().sum();
            println!("x: {}", x);
        });
    handler.unwrap().join().unwrap();
    println!("Fine");
}
```



```
fn main() {
    use std::thread;

    let a = vec![1, 2, 3];

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

    let handler = builder
        .spawn( move || {
            println!("{}", a[4]); // panica. Viene fornito il nome del thread.
        });
    handler.unwrap().join().unwrap();
    println!("Fine");
}
```

builder_panic.rs

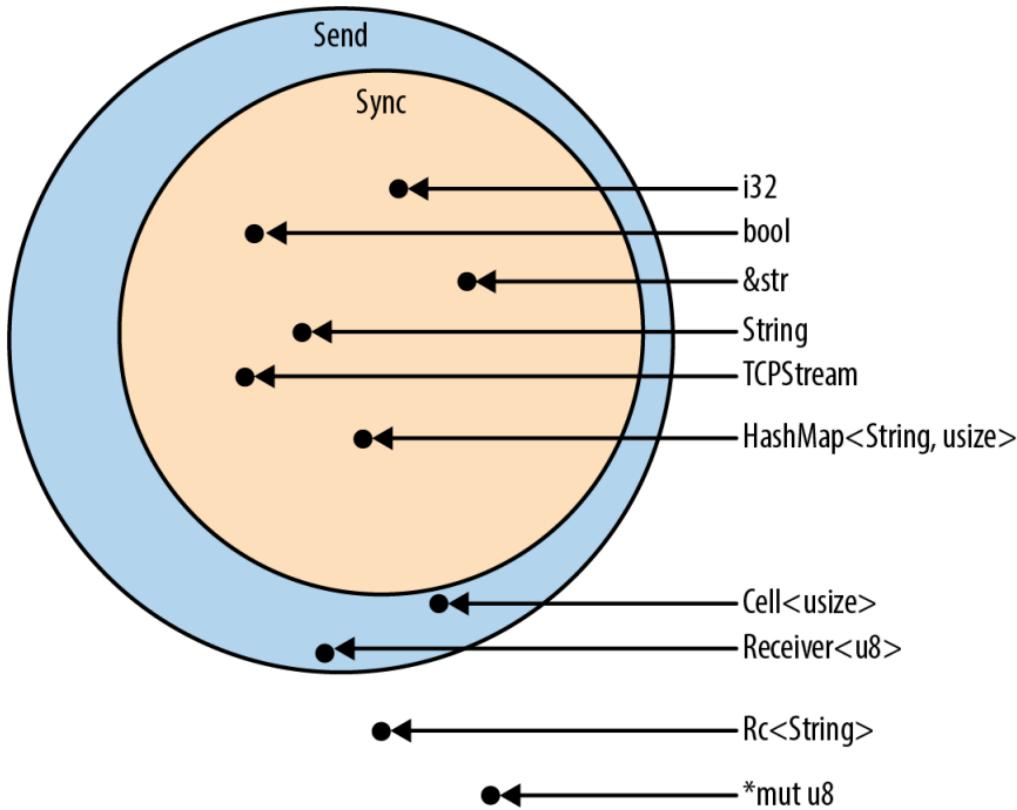
I tratti della concorrenza

- Nell'ambito del proprio sforzo 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** è 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** (un tipo che gode del tratto Send può essere ceduto ad un thread)
 - Il tratto **std::marker::Sync** è 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 (la reference in lettura ad un tipo che gode del tratto Send può essere passata ad un thread)
- Puntatori e riferimenti **non hanno** il tratto **Send**
 - L'esecuzione indipendente dei thread non consente infatti al borrow checker di fornire le proprie garanzie di correttezza

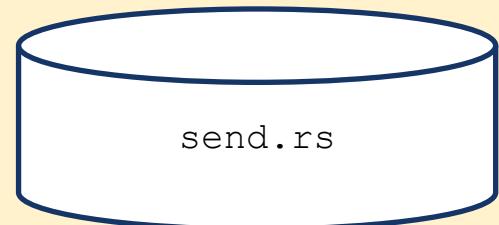
I tratti della concorrenza

- **pub unsafe auto trait Send { }**
 - Se un tipo dispone del tratto **Send**, è lecito passarlo **per valore** ad altri thread
 - L'uso del movimento (o della copia, là dove possibile) garantisce la non contemporaneità degli accessi
- I tipi composti (struct, tuple, enum, array) godono del tratto **Send** se tutti i loro campi lo posseggono
 - E' possibile forzare l'assegnazione/rimozione di tale tratto solo all'interno di un blocco unsafe: il programmatore deve essere consapevole della scelta adottata e farsi carico della relativa responsabilità
- **pub unsafe auto trait Sync { }**
 - Se un tipo dispone del tratto **Sync**, è lecito passarlo **come riferimento non mutabile** ad altri thread o, in altre parole, un tipo è **Sync** se è lecito accedervi in modo concorrente a partire da un riferimento non mutable
 - I tipi che implementano una mutabilità interna (come **Cell** e **RefCell**) non dispongono di questo tratto
 - Neanche **Rc** dispone di questo tratto
 - **Rc**, **Cell** e **RefCell** non possono essere usati in contesto multi-thread.

I tratti della concorrenza



```
struct Message {  
    content: String,  
    sender_id: u32,  
}  
  
fn main() {  
    let msg = Message {  
        content: "Ciao dal thread principale!".to_string(),  
        sender_id: 42,  
    };  
  
    // Crea un nuovo thread e invia il messaggio.  
    std::thread::spawn(move || {  
        println!("Messaggio ricevuto: {}", msg.content);  
    })  
        .join()  
        .unwrap();  
}
```



```
use std::cell::Cell;
use std::thread;

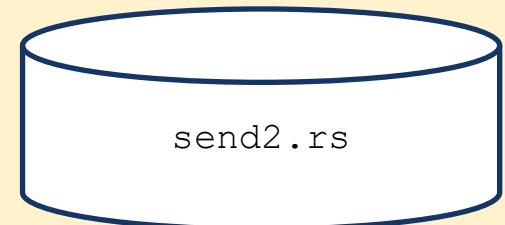
fn main() {
    // Crea un Cell contenente un intero
    let my_cell = Cell::new(42);

    // Clona il Cell per entrambi i thread
    let thread1_cell = my_cell.clone();
    let thread2_cell = my_cell.clone();

    // Thread 1: Incrementa il valore nel Cell
    let handle1 = thread::spawn(move || {
        thread1_cell.set(thread1_cell.get() + 1);
        println!("Thread1: Valore nel Cell: {}", thread1_cell.get());
    });

    // Thread 2: Legge il valore dal Cell
    let handle2 = thread::spawn(move || {
        println!("Thread2: Valore nel Cell: {}", thread2_cell.get()-1);
    });

    println!("Main Thread: Valore nel Cell: {}", my_cell.get());
    // Attendi che entrambi i thread terminino
    handle1.join().unwrap();
    handle2.join().unwrap();
}
```



```
fn main() {
    // Creiamo una variabile condivisa tra più thread.
    let shared_value = 42;

    // Creiamo due thread che accedono alla variabile condivisa
    let handle1 = std::thread::spawn(move || {
        println!("Thread 1: Valore condiviso = {}", shared_value);
        let a = shared_value + 1;
        println!("Thread 1: Valore calcolato = {}", a);
    });

    let handle2 = std::thread::spawn(move || {
        println!("Thread 2: Valore condiviso = {}", shared_value);
        let b = shared_value + 10;
        println!("Thread 2: Valore calcolato = {}", b);
    });

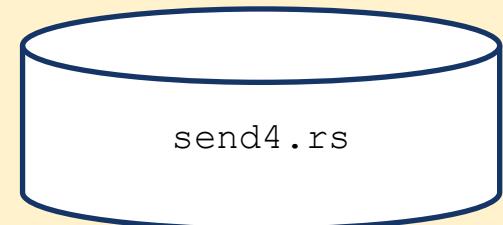
    handle1.join().unwrap();
    handle2.join().unwrap();
}
```



```
fn main() {
    // Creiamo una variabile condivisa tra thread
    let mut shared_data = SharedData { value: 42 };

    // Creiamo un thread che accede alla variabile
    let handle = thread::spawn(move || {
        shared_data.value += 1;
        println!("Il valore è: {}", shared_data.value); // Output: 43
    });

    // Attendiamo il completamento del thread
    handle.join().unwrap();
    println!("Il valore è: {}", shared_data.value); // Output: 42
}
```



I tratti della concorrenza

err_send.rs

- E' 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**
 - In mancanza di questa garanzia, il borrow checker genera un errore di compilazione, rilevando la propria impossibilità a garantire la correttezza di quanto si sta chiedendo di eseguire

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

    let jh = thread::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>`
```



I tratti della concorrenza

sendarc.rs

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

fn main() {
    let data1 = Arc::new(1);
    let data2 = data1.clone();
    let data3 = data1.clone();
    println!("t0: {}", *data1);

    let jh1 = thread::spawn(move || {
        println!("t1: {}", *data2);
    });
    let jh2 = thread::spawn(move || {
        println!("t2: {}", *data3);
    });
    jh1.join().unwrap();
    jh2.join().unwrap();
}
```

Scoped Thread

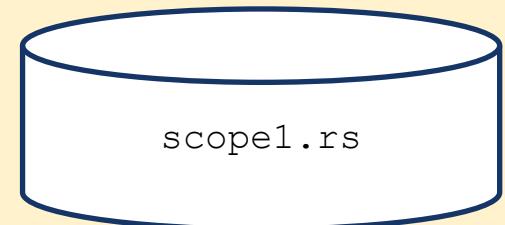
- Se un thread viene creato mediante la primitiva `std::thread::spawn(...)`, il compilatore non può fare assunzioni sulla sua durata
 - Di conseguenza, impedisce l'utilizzo di riferimenti condivisi tra la funzione lambda del thread e la funzione all'interno della quale il thread viene creato
- La libreria standard offre un ulteriore modo di creare un thread, tramite la funzione `std::thread::scope(|s: std::thread::Scope| { ... })`
 - Essa accetta come parametro una funzione 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 funzione 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

```
use std::thread;

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

    thread::scope(|s| {
        s.spawn(|| {
            println!("length: {}", numbers.len());
            // riferimento a variabile locale che non deve essere catturata con move
        });

        s.spawn(|| {
            for n in &numbers {
                println!("{}");
            }
        });
    });
}
```



```
use std::thread;
fn main()
{
    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 &v
for n in &v {println!("{}", n); }
x += v[0]+v[2]; // x è catturata come &mut
});
});

// Solo quando entrambi i thread saranno terminati si proseguirà
v.push(4); // non ci sono più riferimenti, si può modificare
println!("Vettore: {:?}", v);
println!("x: {:?}", x);
}
```

scope2.rs

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
 2. La condivisione di dati basata sullo **scambio di messaggi** che prevede uno o più mittenti ed un solo destinatario
- Sono inoltre disponibili librerie esterne che supportano ulteriori modelli
 - La libreria **actix** supporta il modello degli attori
 - La libreria **rayon** supporta il modello work stealing
 - La libreria **crossbeam** permette la condivisione di dati memorizzati nello stack del thread genitore con i thread figli

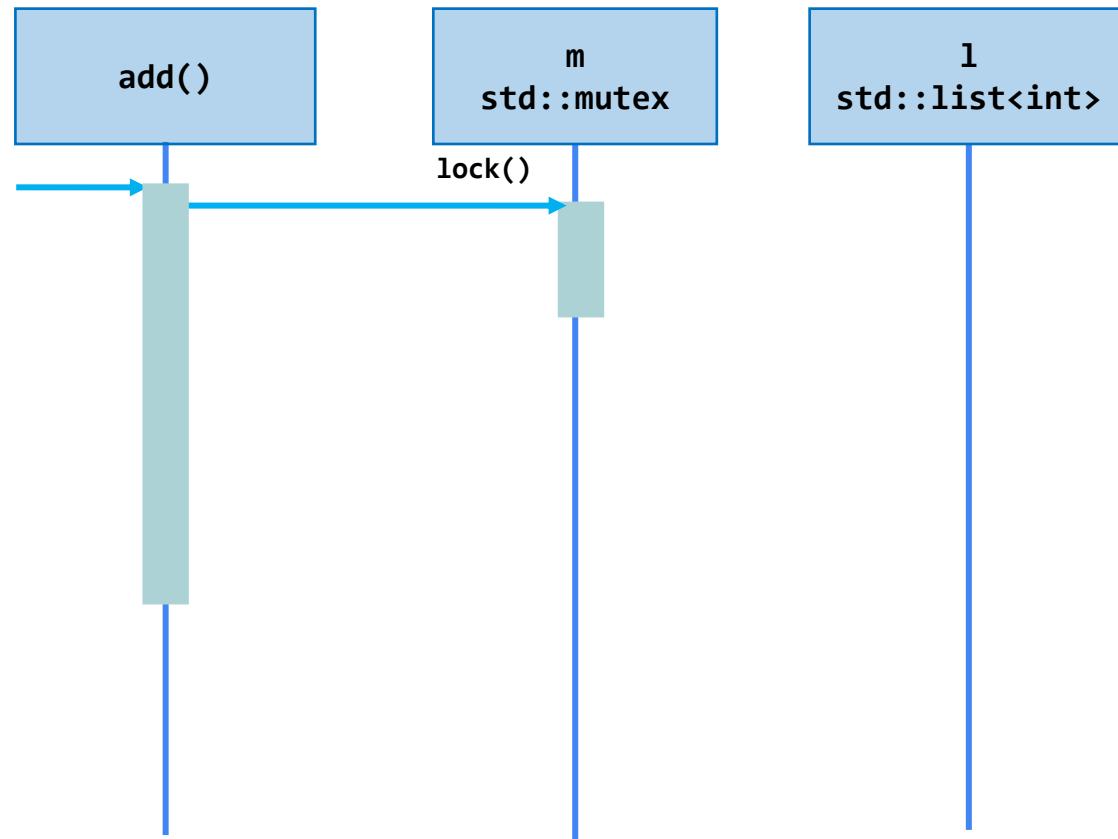
Condivisione dello stato

- Per poter condividere dati modificabili tra thread differenti, occorre disporre di un qualche meccanismo che permetta, ad un solo thread alla volta, di acquisire il permesso di modifica
 - Bloccando lo svolgimento degli altri thread che dovessero richiedere accesso alla stessa risorsa
- Il modo più semplice di ottenere questo comportamento è attraverso l'uso di un **mutex** (MUTual EXclusion lock)
 - Costrutti di questo tipo sono offerti nativamente dai sistemi operativi e sono riesportati in modo indipendente dalla piattaforma dalle librerie standard C++ e Rust
- Gli oggetti nativi offerti dai sistemi operativi offrono due metodi: **lock()** e **unlock()**
 - Invocando lock(), un thread richiede il possesso del mutex: se questo non può essere garantito al momento, perché il mutex è in uso ad un altro thread, **l'invocazione si blocca** fino a che il mutex non è stato rilasciato dall'attuale possessore
 - E' lecito invocare unlock() solo se il thread che lo esegue è l'attuale possessore del mutex

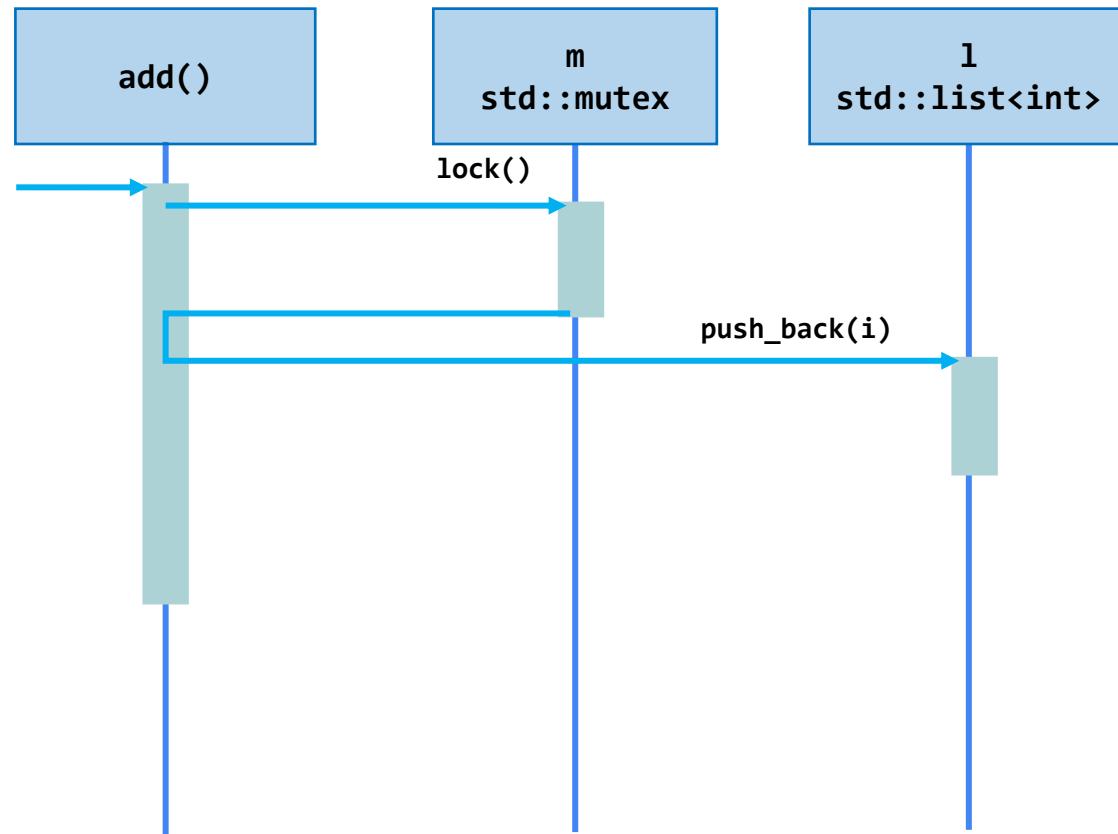
Condivisione dello stato

- Entrambi i metodi **lock()** e **unlock()** includono una barriera di memoria
 - Essa garantisce la visibilità delle operazioni eseguite fino a quel punto dagli altri thread
 - Permette di imporre la dipendenza causale
- Sul piano pratico, occorre associare ad ogni risorsa condivisa un mutex
 - Che deve essere **sempre** acquisito prima di fare accesso (sia in lettura che in scrittura) alla risorsa
- Nelle astrazioni base offerte dai sistemi operativi e nell'implementazione offerta in C++, **non c'è una corrispondenza sintattica tra un mutex e la struttura dati che questo protegge**
 - La relazione, in questi ambienti, è nella mente (e nelle intenzioni) del programmatore
- Un mutex può, in linea di principio, proteggere molte strutture diverse
 - Ma riduce il grado di parallelismo complessivo del programma

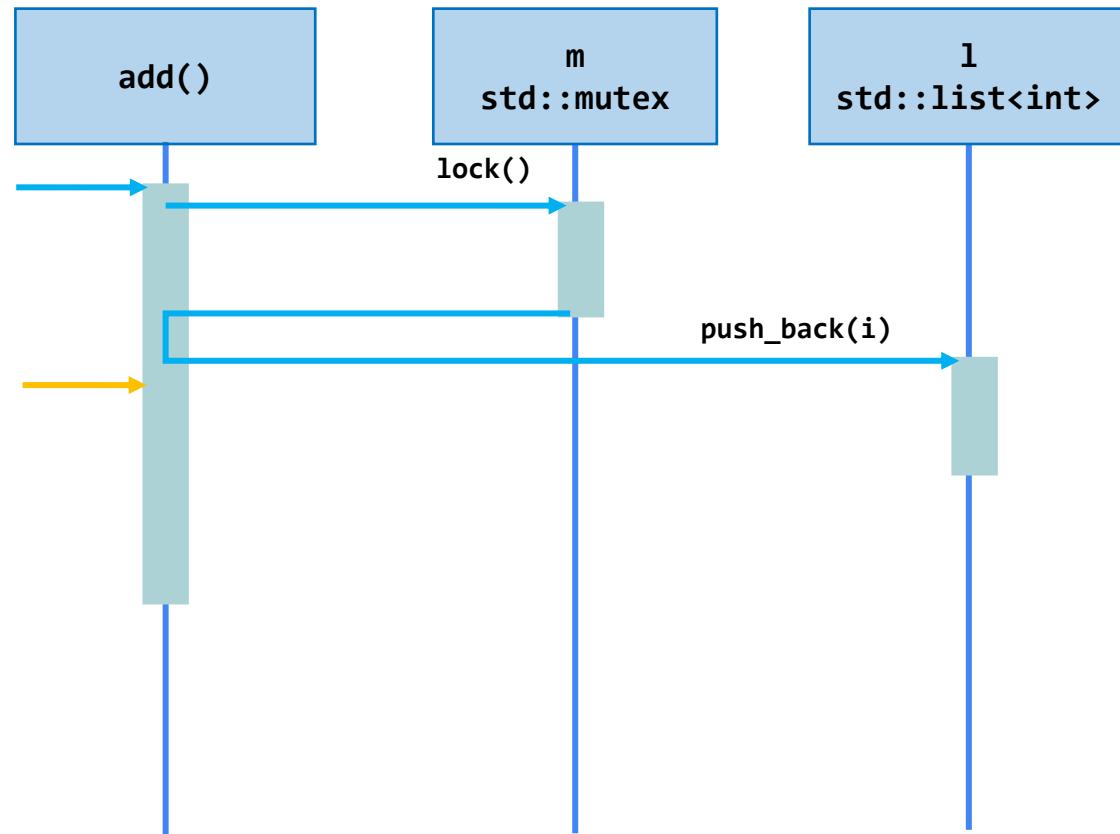
```
std::list<int> l;  
std::mutex m;  
  
void add(int i){  
    m.lock();  
  
    l.push_back(i);  
  
    m.unlock();  
}
```



```
std::list<int> l;  
std::mutex m;  
  
void add(int i){  
  
    m.lock();  
  
    l.push_back(i);  
  
    m.unlock();  
  
}
```



```
std::list<int> l;  
std::mutex m;  
  
void add(int i){  
    m.lock();  
  
    l.push_back(i);  
  
    m.unlock();  
}
```

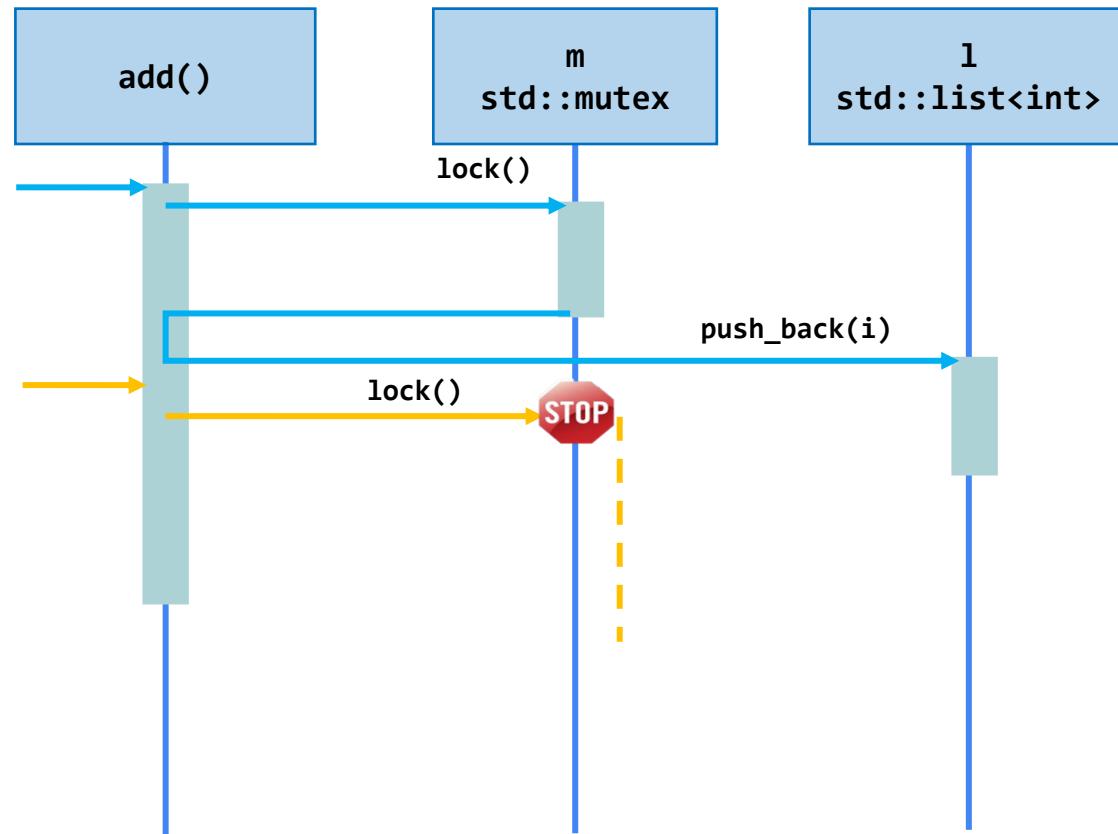


```

std::list<int> l;
std::mutex m;

void add(int i){
    m.lock();
    l.push_back(i);
    m.unlock();
}

```

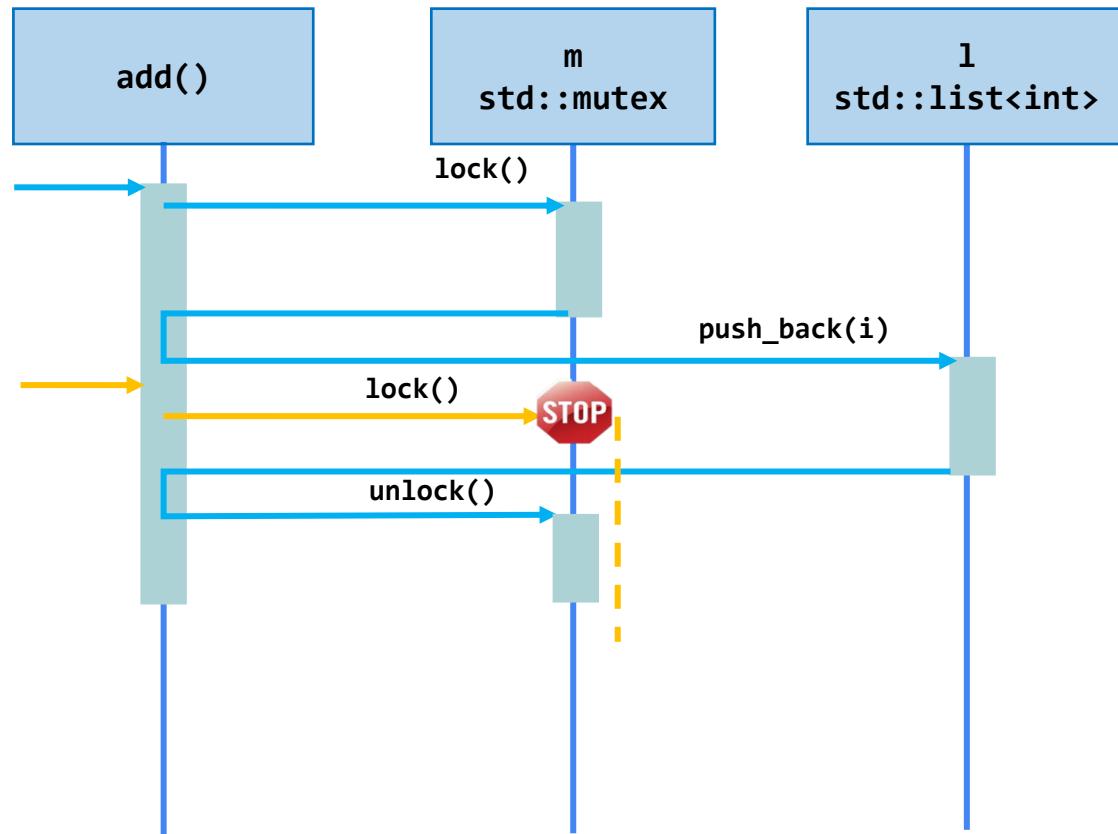


```

std::list<int> l;
std::mutex m;

void add(int i){
    m.lock();
    l.push_back(i);
    m.unlock();
}

```

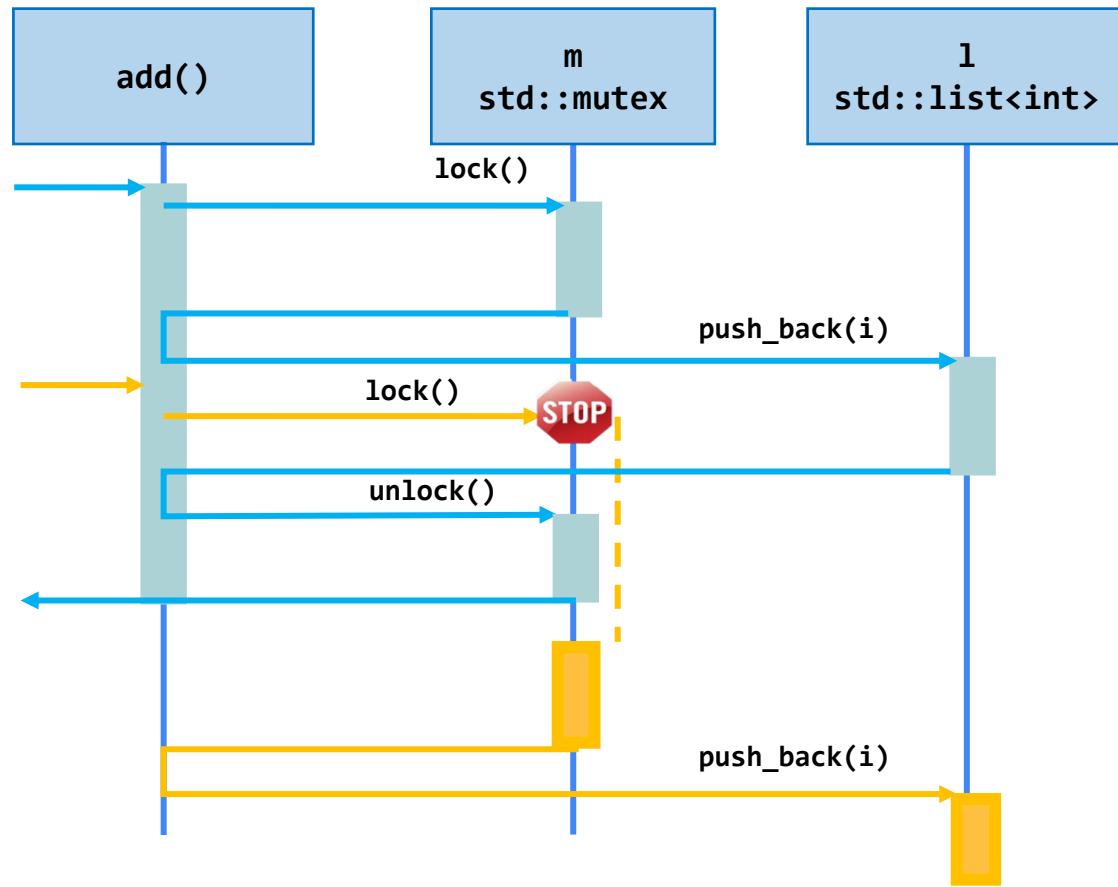


```

std::list<int> l;
std::mutex m;

void add(int i){
    m.lock();
    l.push_back(i);
    m.unlock();
}

```



Rilasciare i mutex

- Se un thread che è in possesso di un mutex termina (anche, ma non solo, per via di un errore) senza rilasciare il mutex, si crea un problema
 - I sistemi operativi tendono liberare il mutex
 - Però, quale stato hanno le risorse che il mutex protegge?
- Per evitare la situazione, in C++ si ricorre al paradigma RAII
 - La classe `std::lock_guard` incapsula un `std::mutex`: il costruttore lo acquisisce, il distruttore lo rilascia; non ci sono altri metodi
 - Questo garantisce che, mentre esiste l'istanza del `lock_guard`, si possegga il `mutex` e se - per qualsiasi motivo - cessa di esistere il `lock_guard`, il `mutex` sia comunque rilasciato

```
namespace std {  
  
    template <class T>  
    class lock_guard {  
  
        private:  
            T& m_lockable;  
  
        public:  
            lock_guard(T& lockable) :  
                m_lockable(lockable) {  
                lockable.lock();  
            }  
  
            ~lock_guard() {  
                m_lockable.unlock();  
            }  
    };  
}
```

C++

```
template <class T>  
class shared_vector {  
    std::vector<T> v;  
    std::mutex m;  
  
    public:  
        int size() {  
            std::lock_guard<std::mutex> l(m);  
            return v.size();  
        } // il rilascio avviene qui!  
  
        T front() {  
            std::lock_guard<std::mutex> l(m);  
            return v.front();  
        } // il rilascio avviene qui!  
  
        void push_back(T t) {  
            std::lock_guard<std::mutex> l(m);  
            v.push_back(t);  
        } // il rilascio avviene qui!  
};
```

C++

Valutazione

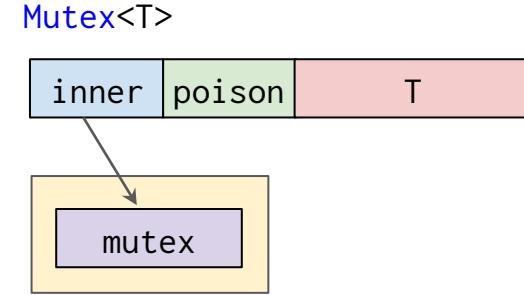
- L'uso del pattern RAI garantisce che il lock sia rilasciato automaticamente nel momento in cui l'oggetto **lock_guard** viene distrutto
 - Sia perché l'esecuzione ha raggiunto la normale fine del metodo
 - Sia perché si è verificata un'eccezione e c'è stata una contrazione dello stack
- Tuttavia, dall'uso 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
- L'uso corretto della sincronizzazione e le sue evoluzioni nel tempo, legate alla manutenzione della classe, **restano affidate** principalmente **ai commenti** eventualmente presenti nel codice
 - Senza che il compilatore possa fornire un supporto attivo per garantire il rispetto dei vincoli

Mutex in Rust

- L'accesso ad uno stato condiviso in Rust richiede l'utilizzo di **due blocchi** in cascata:
 - Il primo volto a permettere 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 che consente l'acquisizione in lettura/scrittura della struttura dati, realizzato alternativamente mediante il costrutto `std::sync::Mutex<T>`, con `std::sync::RwLock<T>` oppure ricorrendo ai **tipi atomici**
- Questa combinazione 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
 - Permettendo al compilatore di bloccare ogni tentativo di accesso non conforme

std::sync::Mutex<T>

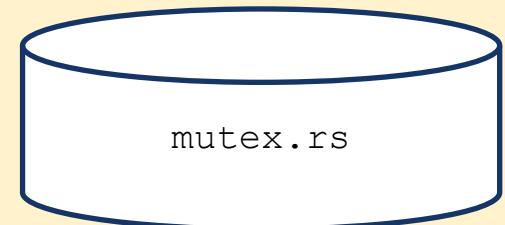
- Un oggetto di tipo Mutex incapsula un dato di tipo T oltre al riferimento ad un mutex nativo del sistema operativo
 - L'unico modo per accedere al dato è invocare il metodo **lock()**
 - Questo metodo restituisce un oggetto di tipo **LockResult<MutexGuard<T>>** e resta bloccato fino a che non è stato 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
- Se il metodo **lock()** ha successo, la risposta contiene un **MutexGuard<T>**
 - Tale oggetto implementa il tratto **Deref<T>** e si comporta come uno smart-pointer
 - Dereferenziandolo, si ottiene 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 **MutexGuard<T>** implementa il pattern RAI, ma - per come viene costruito - è necessariamente disponibile solo se si possiede il mutex



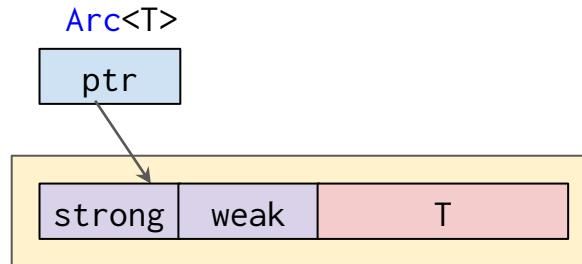
```
use std::sync::Mutex;
use std::thread;

fn main() {
    let n = Mutex::new(0);
    thread::scope(|s| {
        for _ in 0..10 {
            s.spawn(|| {
                let mut guard = n.lock().unwrap();
                for _ in 0..100 {
                    *guard += 1;
                }

                println!("Alla fine del thread:{} n = {:?}", 
                        thread::current().id(), guard);
            });
        }
    });
}
```



std::sync::Arc<T>



- Un oggetto di tipo **Mutex** può avere un solo possesore
 - Per superare questo vincolo, lo si incapsula all'interno di 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*
 - E' possibile duplicare un oggetto di questo tipo attraverso il metodo `clone()`
 - Tale metodo si limita a duplicare il puntatore al blocco sullo heap, avendo cura di incrementare (in modo atomico) 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

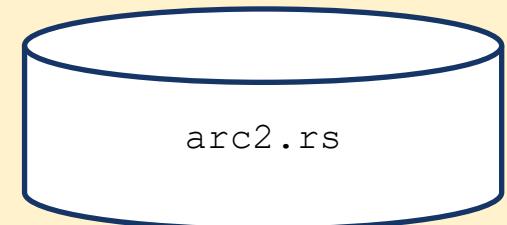
```
use std::sync::Mutex;
use std::sync::Arc;
use std::thread;
fn main() {
    // Creiamo un vettore condiviso protetto da un mutex
    let shared_data = Arc::new(Mutex::new(Vec::new()));

    // Cloniamo l'Arc per poterlo condividere tra più thread
    let shared_data_clone = Arc::clone(&shared_data);

    // Creiamo un thread per aggiungere dati al vettore condiviso
    let handle1 = thread::spawn(move || {
        let mut data = shared_data_clone.lock().unwrap();
        data.push(1);
    });
    // Cloniamo nuovamente l'Arc per un altro thread
    let shared_data_clone = Arc::clone(&shared_data);
    // Creiamo un secondo thread per aggiungere dati al vettore condiviso
    let handle2 = thread::spawn(move || {
        let mut data = shared_data_clone.lock().unwrap();
        data.push(2);
    });
    handle1.join().unwrap();
    handle2.join().unwrap();
    println!("{}: {:?}", shared_data.lock().unwrap());
}
```



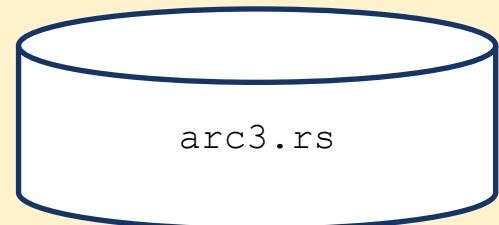
```
use std::sync::{Arc, Mutex};  
use std::thread;  
  
fn main() {  
    let counter = Arc::new(Mutex::new(0));  
    let mut handles = vec![];  
  
    for _ in 0..10 {  
        let counter = Arc::clone(&counter);  
        let handle = thread::spawn(move || {  
  
            let mut num = counter.lock().unwrap();  
  
            *num += 1;  
            println!("Scrivo: {}", num);  
        });  
        handles.push(handle);  
  
    }  
    for handle in handles {  
        handle.join().unwrap();  
    }  
    println!("\nResult: {}", *counter.lock().unwrap());  
}
```



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

fn main() {
    let shared_data = Arc::new(Mutex::new(Vec::new()));
    let mut threads = vec![];
    for i in 1..10 {
        let 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>
            println!("{}:{}", i);
            v.push(i);                      //quando v esce dall scope, il lock
            }) );                          //viene rilasciato
    }
    for t in threads { t.join().unwrap(); }

    //v contiene i numeri da 1 a 9
    println!("\nResult: {:?}", *(shared_data.lock().unwrap()));
}
```



Mutex avvelenato

- Il campo "poison" in un Mutex è un meccanismo per gestire situazioni in cui un thread proprietario di un blocco del Mutex termina in modo anomalo senza rilasciare il blocco. Quando ciò accade, il Mutex entra in uno stato di "veleno" (poisoned), il che significa che il blocco non è stato rilasciato correttamente.
- Questo stato è segnalato al prossimo thread che cerca di ottenere il blocco.
- Quando un MutexGuard tenta di riacquisire il blocco dopo che è stato rilasciato in modo anomalo, il MutexGuard solleva un errore per segnalare la situazione di "poisoning".
- Questo aiuta a identificare e gestire situazioni in cui un thread potrebbe aver lasciato il blocco del Mutex in uno stato inconsistente, aiutando a garantire l'integrità dei dati condivisi.

```
use std::time::Duration;
use std::sync::{Arc, RwLock};
use std::thread;
fn main() {
    let data = Arc::new(RwLock::new(vec![1, 2, 3]));
    let data_clone1 = Arc::clone(&data);
    let data_clone2 = Arc::clone(&data);
    let data_clone3 = Arc::clone(&data);
    let reader1 = thread::spawn(move || { // Thread in lettura 1
        let guard = data_clone1.read().unwrap();
        println!("Thread lettura 1: {:?}", *guard);
    });
    let reader2 = thread::spawn(move || { // Thread in lettura 2
        thread::sleep(Duration::from_secs(1));
        let guard = data_clone2.read();
        match guard {
            Ok(guard) => { println!("Valore letto con successo: {:?}", guard); }
            Err(poison_error) => {
                // Gestiamo l'errore derivante dal Mutex in stato "poisoned"
                println!("Reader 2 - errore: il Mutex è stato avvelenato");}
        }
    });
    let writer = thread::spawn(move || { // Thread in scrittura (che provoca uno stato "poisoned")
        let mut guard = data_clone3.write().unwrap();
        guard.push(4); // Prova a inserire un elemento nella struttura dati
        panic!("Oops, ho fatto un errore!"); // Simula un errore (ad esempio, un panic)
    });
    reader1.join().unwrap();
    reader2.join().unwrap();
    writer.join().unwrap_err();
}
```

poison.rs



Leggere un dato in una struttura avvelenata

- Una volta che un Mutex è stato avvelenato (cioè un thread ha panicato mentre deteneva il Mutex), rimarrà in uno stato di errore permanente. Questo è fatto per garantire che altri thread non accedano a dati potenzialmente inconsistenti
- E' però possibile accedere al dato protetto dal Mutex (in lettura o scrittura) anche nel caso di Mutex avvelenato attraverso la lettura dell'errore, mediante il metodo `into_error()` che fornisce

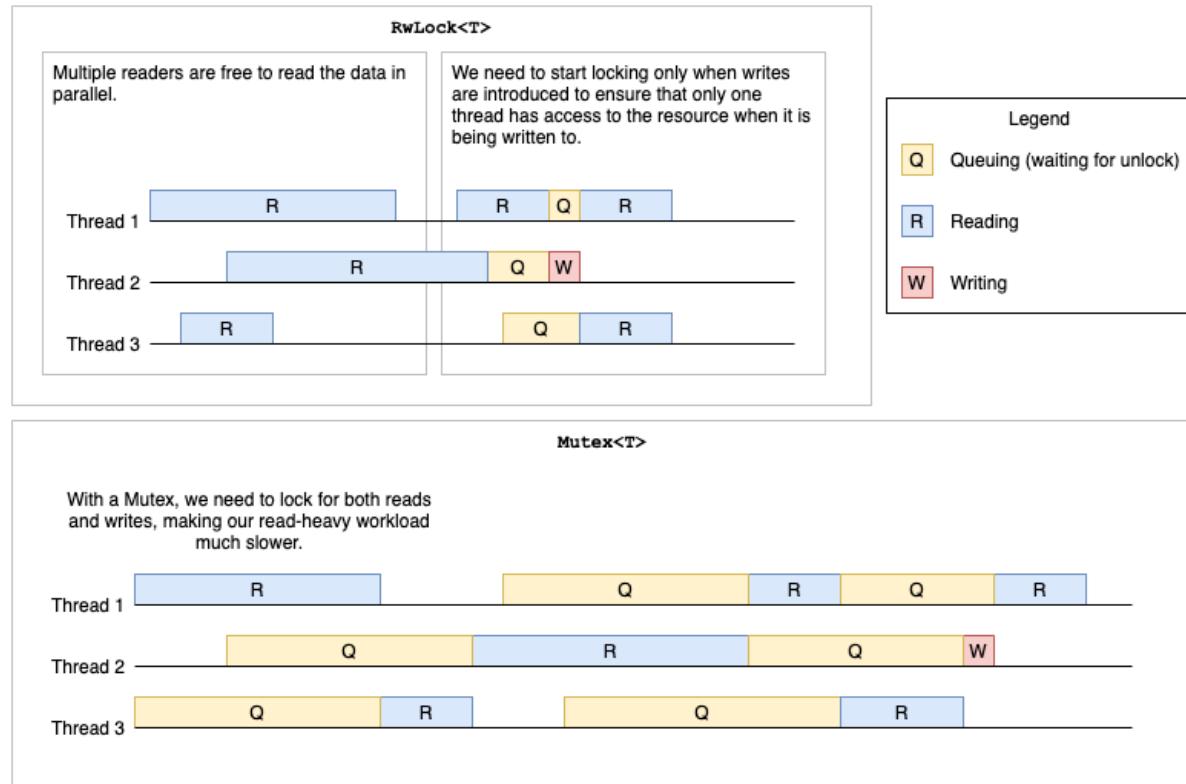
```
use std::sync::{Arc, Mutex};
use std::thread;
fn main() {
    let data = Arc::new(Mutex::new(0));
    let data_cloned = Arc::clone(&data);
    let _ = thread::spawn(move || {
        let mut num = data_cloned.lock().unwrap();
        *num += 1;
        panic!("Oops! Il thread ha avvelenato il mutex.");
    }).join();
    let result = data.lock(); // Tentiamo di acquisire il mutex nel thread principale
    match result {
        Ok(guard) => { println!("Mutex non avvelenato. Valore: {}", *guard); }
        Err(poisoned) => {
            // Il mutex è avvelenato. Recuperiamo il valore
            let mut guard = poisoned.into_inner();
            println!("Mutex avvelenato. Valore recuperato: {}", *guard);
            // Possiamo modidificare il dato del mutex
            *guard += 1;
            println!("Stato del mutex resettato. Nuovo valore: {}", *guard);
        }
    }
    let result = data.lock(); // Tentiamo di acquisire nuovamente il mutex nel thread principale
    match result {
        Ok(guard) => { println!("Mutex non avvelenato. Valore: {}", *guard); }
        Err(poisoned) => {
            // Il mutex rimane avvelenato, ma possiamo recuperare il valore
            let mut guard = poisoned.into_inner();
            println!("Mutex avvelenato. Valore recuperato: {}", *guard);
            // Possiamo ancora modificare il dato
            *guard += 1;
            println!("Stato del mutex resettato. Nuovo valore: {}", *guard);
        }
    }
}
```

poison2.rs



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 offre il metodo **read()** per accedere, in modo condiviso, in lettura ed il metodo **write()** per accedere in modo esclusivo in scrittura



std::sync::RwLock<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 se un thread che lo possedeva 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;

fn main() {
    // Creiamo una struttura dati condivisa protetta da un RwLock
    let data = Arc::new(RwLock::new(vec![1, 2, 3, 4, 5]));

    // Creiamo 10 thread che leggono dalla struttura dati condivisa
    let mut threads = vec![];

    for i in 0..10 {
        // Cloniamo l'arc per ogni thread in modo che ciascuno abbia un riferimento condiviso
        let data_clone = Arc::clone(&data);

        // Creiamo il thread
        let thread = thread::spawn(move || {
            // Otteniamo un blocco di lettura (non esclusivo) sul RwLock
            let guard = data_clone.read().unwrap();
            println!("Thread {}: {:?}", i, *guard);
        });

        threads.push(thread);
    }

    // Attendo che tutti i thread terminino
    for thread in threads {
        thread.join().unwrap();
    }
}

```



```

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

fn main() {
    // Creiamo una struttura dati condivisa protetta da un RwLock
    let data = Arc::new(RwLock::new(vec![1, 2, 3]));

    // Cloniamo l'arc per ogni thread in modo che ciascuno abbia un riferimento condiviso
    let data_clone1 = Arc::clone(&data);
    let data_clone2 = Arc::clone(&data);

    // Thread che legge dalla struttura dati condivisa
    let reader = thread::spawn(move || {
        // Otteniamo un blocco di lettura (non esclusivo) sul RwLock
        let guard = data_clone1.read().unwrap();
        println!("Thread lettore: {:?}", guard);
    });

    // Thread che scrive sulla struttura dati condivisa
    let writer = thread::spawn(move || {
        // Otteniamo un blocco di scrittura (esclusivo) sul RwLock
        let mut guard = data_clone2.write().unwrap();
        guard.push(4);
        println!("Thread scrittore: {:?}", guard);
    });

    // Attendo che entrambi i thread terminino
    reader.join().unwrap();
    writer.join().unwrap();
}

```



```

use std::time::Duration;
use std::sync::{Arc, RwLock};
use std::thread;
fn main() {
    let data = Arc::new(RwLock::new(vec![1, 2, 3]));
    let data_clone1 = Arc::clone(&data);
    let data_clone2 = Arc::clone(&data);
    let data_clone3 = Arc::clone(&data);
    let reader1 = thread::spawn(move || { // Thread in lettura 1
        let guard = data_clone1.read().unwrap();
        println!("Thread lettura 1: {:?}", *guard);
    });
    let reader2 = thread::spawn(move || { // Thread in lettura 2
        thread::sleep(Duration::from_secs(1));
        let guard = data_clone2.read().unwrap();
        // La lettura di un RwLock avvelenato causa un panic
        println!("Thread lettura 2: {:?}", *guard);
    });
    let writer = thread::spawn(move || { // Thread in scrittura (che provoca uno stato "poisoned")
        let mut guard = data_clone3.write().unwrap();
        guard.push(4); // Prova a inserire un elemento nella struttura dati
        panic!("Oops, ho fatto un errore!"); // Simula un errore (ad esempio, un panic)
    });
    reader1.join().unwrap();
    reader2.join().unwrap_err();
    writer.join().unwrap_err();
}

```



```

use std::time::Duration;
use std::sync::{Arc, RwLock};
use std::thread;
fn main() {
    let data = Arc::new(RwLock::new(vec![1, 2, 3]));
    let data_clone1 = Arc::clone(&data);
    let data_clone2 = Arc::clone(&data);
    let data_clone3 = Arc::clone(&data);
    let reader1 = thread::spawn(move || { // Thread in lettura 1
        let guard = data_clone1.read().unwrap();
        println!("Thread lettura 1: {:?}", *guard);
    });
    let reader2 = thread::spawn(move || { // Thread in lettura 2
        thread::sleep(Duration::from_secs(1));
        let guard = data_clone2.read();
        match guard {
            Ok(guard) => { println!("Valore letto con successo: {:?}", guard); }
            Err(poison_error) => {
                println!("Reader 2: RwLock è stato avvelenato. Contiene {:?}", poison_error.into_inner());
            }
        }
    });
    let writer = thread::spawn(move || { // Thread in scrittura (che provoca uno stato "poisoned")
        let mut guard = data_clone3.write().unwrap();
        guard.push(4); // Prova a inserire un elemento nella struttura dati
        panic!("Ops, ho fatto un errore!"); // Simula un errore (ad esempio, un panic)
    });
    reader1.join().unwrap();
    reader2.join().unwrap();
    writer.join().unwrap_err();
}

```



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 (**AtomicBool**, **AtomicIsize**, **AtomicUsize**, **AtomicI8**, **AtomicU8**, **AtomicI16**, **AtomicU16**, **AtomicI32**, **AtomicU32**, **AtomicI64**, **AtomicU64**)
 - Ciascun tipo è associato ad operazioni che, se usate correttamente, permettono di sincronizzare gli aggiornamenti di tali valori (*read and modify*) tra thread differenti
 - L'operazione di aggiornamento diventa indivisibile, evitando *undefined behavior*

Tipi atomici: load e store

- Permettono operazioni di lettura (**load**) e scrittura (**store**) con associata barriera di memoria

```
impl AtomicI32 {  
    pub fn load(&self, ordering: Ordering) -> i32;  
    pub fn store(&self, value: i32, ordering: Ordering);  
}
```

Ordering

- Ogni operazione atomica definisce un argomento di tipo std::sync::atomic::Ordering che specifica come le operazioni atomiche sincronizzano la memoria tra i thread. Le possibili varianti sono:
 - Relaxed: Questa è la variante più debole. Non ha vincoli di ordinamento
 - Release: applicabile solo per operazioni che eseguono una scrittura e determina che tutte le operazioni non atomiche eseguite prima dell'operazione atomica **non** vengano riordinate **dopo** di essa
 - Acquire: applicabile solo per operazioni che eseguono una lettura e determina che tutte le operazioni non atomiche eseguite dopo l'operazione atomica **non** vengano riordinate **prima** di essa
 - AcqRel: applicabile solo per operazioni che combinano sia letture che scritture e garantisce sia l'acquire che il release (nessuna operazione non atomica che precede l'istruzione atomica è riordinata dopo di essa e nessuna operazione non atomica che segue l'istruzione è riordinata prima di essa)
 - SeqCst: impone che tutti i thread vedono tutte le operazioni sequenzialmente consistenti nello stesso ordine.

```

use std::sync::{Arc, Mutex};
use std::sync::atomic::{AtomicBool, Ordering};
use std::thread;
const BUFFER_SIZE: usize = 10;
fn main() {
    let buffer = Arc::new(Mutex::new(Vec::with_capacity(BUFFER_SIZE)));
    let producer_finished = Arc::new(AtomicBool::new(false));
    let buffer_clone = Arc::clone(&buffer);
    let producer_finished_clone = Arc::clone(&producer_finished);
    let producer_handle = thread::spawn(move || {
        for i in 0..10 {
            thread::sleep(std::time::Duration::from_nanos(10));
            let mut buffer = buffer_clone.lock().unwrap();
            buffer.push(i); // Aggiungi un valore al buffer
            println!("Produttore: prodotto {}", i);
        }
        producer_finished_clone.store(true, Ordering::Release);
    });
    let buffer_clone = Arc::clone(&buffer);
    let consumer_handle = thread::spawn(move || {
        loop {
            let mut buffer = buffer_clone.lock().unwrap();
            let len = buffer.len();
            if len > 0 {
                let value = buffer.remove(0); // Se ci sono elementi nel buffer, consumane uno
                println!("Consumatore: consumato {}", value);
            } else if producer_finished.load(Ordering::Acquire) {
                break; // Se il produttore ha finito, esce dal ciclo
            }
            // Altrimenti, rilascia il lock e attendi finché ci sono elementi nel buffer o il produttore ha finito
        }
        println!("Tutti gli elementi sono stati consumati");
    });
    producer_handle.join().unwrap();
    consumer_handle.join().unwrap();
}

```



Tipi atomici: fetch_modify

- Funzionalità di tipo Read-Modify-Write: modificano la variabile atomica, ma anche leggono (*fetch*) il valore originale in una singola istruzione atomica

```
impl AtomicI32 {  
    pub fn fetch_add(&self, v: i32, ordering: Ordering) -> i32;  
    pub fn fetch_sub(&self, v: i32, ordering: Ordering) -> i32;  
    pub fn fetch_or(&self, v: i32, ordering: Ordering) -> i32;  
    pub fn fetch_and(&self, v: i32, ordering: Ordering) -> i32;  
    pub fn fetch_nand(&self, v: i32, ordering: Ordering) -> i32;  
    pub fn fetch_xor(&self, v: i32, ordering: Ordering) -> i32;  
    pub fn fetch_max(&self, v: i32, ordering: Ordering) -> i32;  
    pub fn fetch_min(&self, v: i32, ordering: Ordering) -> i32;  
    pub fn swap(&self, v: i32, ordering: Ordering) -> i32; // "fetch_store"  
}
```

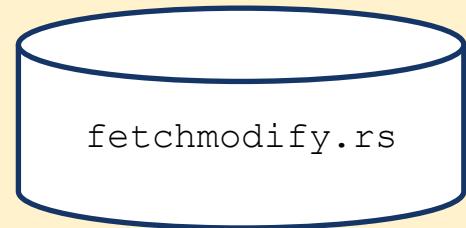
```
use std::sync::atomic::Ordering::Relaxed;

fn main() {
    use std::sync::atomic::AtomicI32;

    let a = AtomicI32::new(100);
    let b = a.fetch_add(23, Relaxed);
    let c = a.load(Relaxed);

    assert_eq!(b, 100);
    assert_eq!(c, 123);

}
```



Tipi atomici: compare_exchange

- Funzionalità di tipo Read-Modify-Write: Questa operazione controlla, atomicamente, se il valore atomico è uguale a un determinato valore e solo in tal caso lo sostituisce con un nuovo valore,
- Restituirà il valore precedente e ci dirà se lo ha sostituito o meno.

```
impl AtomicI32 {  
    pub fn compare_exchange(  
        &self,  
        expected: i32,  
        new: i32,  
        success_order: Ordering,  
        failure_order: Ordering  
    ) -> Result<i32, i32>;  
}
```

- compare_exchange si comporta così

```
impl AtomicI32 {
    pub fn compare_exchange(&self, expected: i32, new: i32)
                           -> Result<i32, i32> {
        let v = self.load();
        if v == expected {
            // Value is as expected.
            // Replace it and report success.
            self.store(new);
            Ok(v)
        } else { // The value was not as expected.
            // Leave it untouched and report failure.
            Err(v)
        }
    }
}
```

Tipi atomici

- Sebbene siano tutti *thread-safe* (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 all'interno di un elemento di tipo **Arc<T>** oppure dichiararli come variabili globali, attraverso la parola chiave **static**
- implementano il meccanismo di mutabilità interna (in modo analogo a **Cell<T>**)
 - Poiché 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::thread;
fn main() {
    const NUM_THREADS: usize = 4;
    const NUM_INCREMENTS: usize = 100_000;

    let counter = Arc::new(AtomicUsize::new(0));
    let mut handles = vec![];
    for _ in 0..NUM_THREADS {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            for _ in 0..NUM_INCREMENTS {
                counter.fetch_add(1, Ordering::Relaxed);
            }
        });
        handles.push(handle);
    }
    for handle in handles {
        handle.join().unwrap();
    }
    println!("Final counter value: {}", counter.load(Ordering::Relaxed));
}
```



```
use std::sync::Arc;
use std::thread;
use std::sync::atomic::{AtomicBool, Ordering};
fn main() {
    let running = Arc::new(AtomicBool::new(true));
    let running_clone = Arc::clone(&running);
    let handle = thread::spawn(move || {
        while running_clone.load(Ordering::Relaxed) {
            println!("Working...");
            thread::sleep(std::time::Duration::from_secs(1));
        }
        println!("Thread exiting...");
    });
    // Simulate main thread doing some work
    thread::sleep(std::time::Duration::from_secs(5));

    // Set the running flag to false to signal the thread to exit
    running.store(false, Ordering::Relaxed);

    // Wait for the spawned thread to finish
    handle.join().unwrap();
}
```



arcatomic2.rs



- E' possibile in Rust definire variabili globali mutabili ?

- Le variabili globali possono essere solo immutabili
- Variabili globali mutabili possono essere solo inserite in blocchi **unsafe**.

```
static mut COUNTER:i32 = 0;

fn increment_counter() {
    COUNTER += 1;
    println!("New value of COUNTER: {}", COUNTER);
}

fn main() {
    for _ in 1..10 {
        increment_counter();
    }
    println!("Final value of COUNTER: {}", COUNTER);
}
```

```
error[E0133]: use of mutable static is unsafe and requires unsafe function or block
--> src/main.rs:5:5
 |
5 |     COUNTER += 1;
 |     ^^^^^^^^ use of mutable static
```

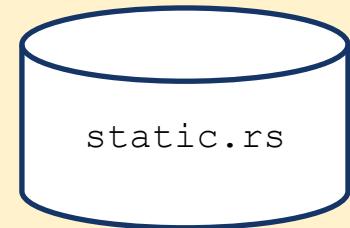
```

use std::sync::atomic::{AtomicUsize, Ordering};
use std::thread;
static COUNTER: AtomicUsize = AtomicUsize::new(0);

fn increment_counter() {
    let new_value = COUNTER.fetch_add(1, Ordering::Relaxed) ;
    println!("Thread ID: {:?}", New value of COUNTER: {},",
std::thread::current().id(), new_value);
}

fn main() {
    let mut handles = vec![];
    // Creare 10 thread e farli eseguire la funzione increment_counter
    for _ in 0..10 {
        let handle = thread::spawn(increment_counter);
        handles.push(handle);
    }
    // Attendere la terminazione dei thread
    for handle in handles {
        handle.join().unwrap();
    }
    println!("Final value of COUNTER: {}", COUNTER.load(Ordering::Relaxed) );
    // Output: Final value of COUNTER: 10
}

```



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

static COUNTER: AtomicUsize = AtomicUsize::new(0);

fn update_counter(new_value: usize) {
    let mut expected_value = 0;
    // Provare a impostare il valore di counter sul nuovo valore
    if let Err(actual_value) = COUNTER.compare_exchange(expected_value, new_value, Ordering::Relaxed,
                                                       Ordering::Relaxed) {
        panic!("Fallimento in compare_exchange(): contiene {} anziché {}",actual_value, expected_value);
    }
}

fn main() {
    let thread = thread::spawn(move || {
        update_counter(1);
    });

    thread.join().unwrap();

    println!("Final value of COUNTER: {}", COUNTER.load(Ordering::Relaxed));
}
```



```

use std::time::Duration;
use std::sync::atomic::Ordering::Relaxed;
use std::thread;
use std::sync::atomic::AtomicUsize;

fn main() {
    let num_done = AtomicUsize::new(0);
    thread::scope(|s| {
        // A background thread to process all 100 items.
        s.spawn(|| {
            for i in 0..100 {
                println!("{}", i);
                num_done.store(i + 1, Relaxed);
            }
        });
        // The main thread shows status updates, every 10 nanoseconds.
        loop {
            let n = num_done.load(Relaxed);
            if n == 100 { break; }
            println!("Working.. {}/100 done");
            thread::sleep(Duration::from_nanos(10));
        }
    });
    println!("Done!");
}

```



Perché non devo incapsularlo in un Arc e non l'ho definito static, ma funziona?



`std::sync::Weak<T>`

- 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 - sulla falsariga di quanto avviene con `std::rc::Weak<T>` - 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()`

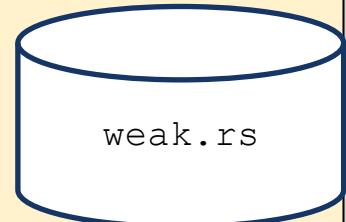
```
use std::sync::{Arc, Weak};

fn main() {
    let data = Arc::new("hello".to_owned());
    let weak_ref = Arc::downgrade(&data); // Create a weak reference

    // Try to upgrade the weak reference
    if let Some(upgraded) = weak_ref.upgrade() {
        println!("Weak reference still alive: {}", upgraded.to_uppercase());
    } else {
        println!("Weak reference is no longer valid.");
    }

    // Dropping the strong reference
    drop(data);

    // Attempt to upgrade again (should return None)
    if let Some(upgraded) = weak_ref.upgrade() {
        println!("Weak reference still alive: {}", upgraded);
    } else {
        println!("Weak reference is no longer valid.");
    }
}
```



Attese condizionate

- Spesso un thread deve aspettare uno o più risultati intermedi prodotti altri thread
 - Per motivi di efficienza, l'attesa non deve consumare risorse e deve terminare non appena un dato è disponibile
- La presenza di dati condivisi richiede come minimo l'utilizzo di un mutex
 - Per garantire l'assenza di interferenze tra i due thread che devono fare accesso ai dati
- Il polling ha due limiti
 - 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 gestire queste situazioni, i sistemi operativi offrono il concetto di **condition variable**
 - Strutture dati di sincronizzazione che permettono di bloccare l'esecuzione di un thread, così da evitare il consumo di CPU, nell'attesa che qualcosa succeda

Attese condizionate

- L'uso di una **condition variable** 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 condition variable
- Il pattern di utilizzo prevede che esista una espressione booleana il cui valore possa essere usato per determinare se occorre attendere o meno
 - La valutazione di tale espressione deve avvenire 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 *condition variable* può determinare un fallimento in fase di esecuzione

Attese condizionate

- Il linguaggio C++ offre la classe **std::condition_variable**
 - Essa viene usata in coppia con un oggetto di tipo **std::mutex** racchiuso all'interno di un oggetto di tipo **std::unique_lock<std::mutex>**
- Rust offre la struct **std::sync::Condvar**
 - La sua semantica è totalmente allineata con la corrispondente classe C++
 - La struttura dei suoi metodi facilita il collegamento con il dato protetto dal mutex, rendendo più naturale il suo utilizzo

Condvar - metodi principali

- **pub fn new() -> Condvar**
 - Crea una nuova istanza
- **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, riaccquisisce il lock e restituisce una nuova guardia
- **pub fn notify_one(&self)**
 - Sveglia un thread a caso tra quelli in attesa sulla condition variable
- **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

```

use std::sync::{Arc, Mutex, Condvar};
use std::thread;
use std::time::Duration;

fn main() {
    // Create an Arc pair containing a Mutex and a Condvar
    let pair = Arc::new( (Mutex::new(false), Condvar::new()) );
    let pair2 = Arc::clone(&pair);

    // Inside our lock, spawn a new thread and wait for it to start
    thread::spawn(move || {
        let (lock, cvar) = &*pair2;
        let mut started = lock.lock().unwrap();
        thread::sleep(Duration::from_secs(5));
        *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();

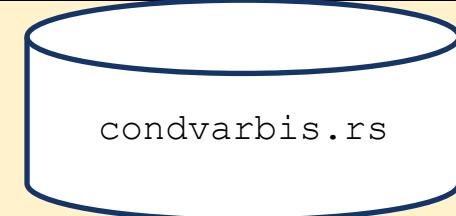
    println!("Waiting ...");
    started = cvar.wait(started).unwrap();
    println!("Thread started!");
}

```



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

```
fn main() {  
    // Create an Arc pair containing a Mutex and a Condvar  
    let pair = Arc::new( (Mutex::new(false), Condvar::new()) );  
    let pair2 = Arc::clone(&pair);  
  
    // Inside our lock, spawn a new thread and wait for it to start  
    thread::spawn(move || {  
        let (lock, cvar) = &*pair2;  
        let mut started = lock.lock().unwrap();  
        thread::sleep(Duration::from_secs(5));  
        // *started = true;  
        cvar.notify_one();  
    });  
  
    // Wait for the thread to start up  
    let (lock, cvar) = &*pair;  
    let mut started = lock.lock().unwrap();  
  
    println!("Waiting ...");  
    started = cvar.wait(started).unwrap();  
    println!("Thread started!");  
}
```



Senza questa
istruzione,
funzionerebbe ?



Meccanismo di funzionamento

- Concettualmente, 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 S.O. 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 (sotto il controllo del mutex)



```
use std::sync::{Arc, Mutex, Condvar};
use std::thread;
fn main() {
    const NUM_THREADS: usize = 5;
    let data = Arc::new((Mutex::new(0), Condvar::new()));
    let mut handles = vec![];
    for i in 0..NUM_THREADS {
        let data_clone = Arc::clone(&data);
        let handle = thread::spawn(move || {
            let (lock, cvar) = &*data_clone;
            let mut data_guard = lock.lock().unwrap();
            println!("Thread {} in attesa...", i);
            // Attendi fino a quando il dato non è diverso da 0
            if *data_guard == 0 { // dopo capiremo che e' meglio avere while come e' nella versione che trovate in rete
                data_guard = cvar.wait(data_guard).unwrap();
            }
            println!("Thread {} svegliato! Il dato è: {}", i, *data_guard);
        });
        handles.push(handle);
    }
    // Facciamo avanzare i thread dopo 2 secondi
    thread::sleep(std::time::Duration::from_secs(2));
    // Cambiamo il dato condiviso e svegliamo tutti i thread
    {
        let (lock, cvar) = &*data;
        let mut data_guard = lock.lock().unwrap();
        *data_guard = 42;
        cvar.notify_all();
    }
    for handle in handles {
        handle.join().unwrap();
    }
}
```

```
use core::time::Duration;
use std::thread;
use std::sync::{Arc, Mutex, Condvar};
fn main() {
    let pair = Arc::new((Mutex::new(Vec::new()), Condvar::new()));
    let pair2 = pair.clone();
    let t = thread::spawn(move || {
        let (m, cv) = &*pair2;
        for i in 0..100 {
            let mut v = m.lock().unwrap();
            v.push(i);
            cv.notify_all();
        }
    });
    let (m, cv) = &*pair;
    let mut round = 0;
    while round != 100 {
        let mut v = m.lock().unwrap();
        if round == v.len() // dopo capiremo che e' meglio avere while come e' nella versione che trovate in rete
        {
            v = cv.wait(v).unwrap();
        }
        println!("Mentre dormivo sono stati prodotti {} elementi ", v.len() - round);

        for i in round .. v.len()
        {
            println!("{}", v[i]);
        }
        round = v.len();
    }
    t.join().unwrap();
}
```



Notifiche spurie

- È possibile che un thread in attesa su una condition variable sia risvegliato in assenza di un'esplicita notifica
 - Problema delle cosiddette **notifiche spurie**
- Occorre, al ritorno dal metodo **wait()**, controllare se la condizione attesa è verificata
 - Per semplificare tale verifica, esiste una versione del metodi 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, ri-acquisisce il lock e valuta la funzione **condition**: se questa restituisce **true**, si riaddormenta, altrimenti esce dall'attesa

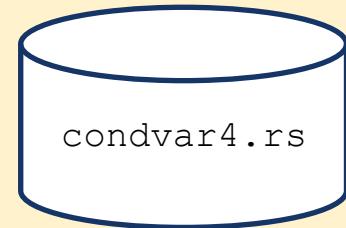
```

use std::{
 sync::{Arc, Condvar, Mutex},
 thread::{sleep},
 time::Duration,
};

struct Counter {
 value: Mutex<u32>,
 condvar: Condvar,
}

fn main() {
 let counter = Arc::new(Counter {
 value: Mutex::new(0),
 condvar: Condvar::new(),
 });
 let counter_clone = counter.clone();
 let counting_thread = std::thread::spawn(move || loop {
 sleep(Duration::from_millis(100));
 let mut value = counter_clone.value.lock().unwrap();
 *value += 1;
 counter_clone.condvar.notify_all();
 if *value >= 15 {
 break;
 }
 });
 // Wait until the value more or equal to 15
 let mut value = counter.value.lock().unwrap();
 value = counter.condvar.wait_while(value, |val| *val < 15).unwrap();
 println!("Condition met. Value is now {}.", *value);
 // Wait for counting thread to finish
 counting_thread.join().unwrap();
}

```



condvar4.rs

# Notifiche perse

- 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 di 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
  - Esso infatti è equivalente al seguente blocco di codice:

```
while condition(&mut *guard) {
 guard = self.wait(guard)?;
}
```

**Ok(guard)**

```

use std::sync::{Arc, Mutex, Condvar};
use std::thread;
use std::time::Duration;

fn main() {
 // Create an Arc (atomic reference counting) pair containing a Mutex and a Condvar
 let pair = Arc::new((Mutex::new(false), Condvar::new()));
 let pair2 = Arc::clone(&pair);

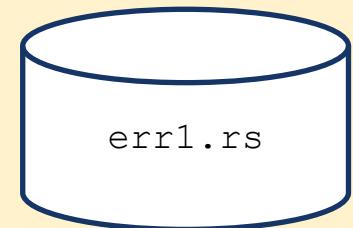
 // Inside our lock, spawn a new thread and wait for it to start
 thread::spawn(move || {
 let (lock, cvar) = &*pair2;
 let mut started = lock.lock().unwrap();
 *started = true; // We notify the Condvar that the value has changed

 cvar.notify_one();
 });

 // Wait for the thread to start up
 let (lock, cvar) = &*pair;

 println!("Waiting ...");
 thread::sleep(Duration::from_secs(1));
 let mut started = lock.lock().unwrap();
 started = cvar.wait(started).unwrap();
 println!("Thread started!");
}

```



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

fn main() {
 // Create an Arc (atomic reference counting) pair containing a Mutex and a Condvar
 let pair = Arc::new((Mutex::new(false), Condvar::new()));
 let pair2 = Arc::clone(&pair);

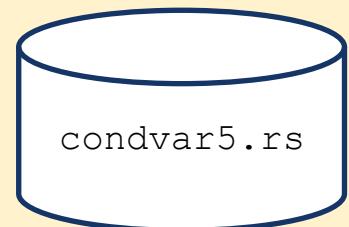
 // Inside our lock, spawn a new thread and wait for it to start
 thread::spawn(move || {
 let (lock, cvar) = &*pair2;
 let mut started = lock.lock().unwrap();
 *started = true; // We notify the Condvar that the value has changed

 cvar.notify_one();
 });

 // Wait for the thread to start up
 let (lock, cvar) = &*pair;

 println!("Waiting ...");
 thread::sleep(Duration::from_secs(1));
 let mut started = lock.lock().unwrap();

 while !*started
 {
 started = cvar.wait(started).unwrap();
 }
 println!("Thread started!");
}
```



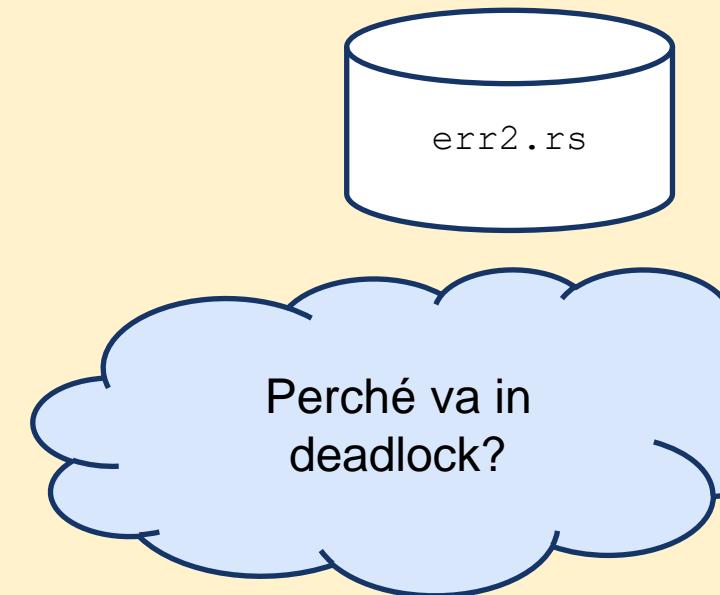
```

use std::{
 sync::{Arc, Condvar, Mutex},
 thread::{sleep},
 time::Duration,
};

struct Counter {
 value: Mutex<u32>,
 condvar: Condvar,
}

fn main() {
 let counter = Arc::new(Counter {
 value: Mutex::new(0),
 condvar: Condvar::new(),
 });
 let counter_clone = counter.clone();
 let counting_thread = std::thread::spawn(move || loop {
 sleep(Duration::from_millis(100));
 let mut value = counter_clone.value.lock().unwrap();
 *value += 1;
 counter_clone.condvar.notify_all();
 if *value > 15 {
 break;
 }
 });
 // Wait until the value more or equal to 15
 let mut value = counter.value.lock().unwrap();
 value = counter.condvar.wait_while(value, |val| *val < 15).unwrap();
 println!("Condition met. Value is now {}.", *value);
 // Wait for counting thread to finish
 counting_thread.join().unwrap();
}

```



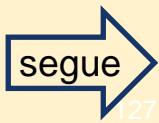
# Attesa temporizzata

- Altri metodi permettono di limitare il tempo massimo di attesa, permettendo al thread di risvegliarsi anche in assenza del verificarsi della condizione
- Come `wait`, il lock specificato verrà riacquisito all'uscita della funzione `wait_timeout*`, indipendentemente dal fatto che il timeout sia scaduto o meno.
- ```
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_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 ri-addormentarsi se la funzione condition restituisce false
```

```

use std::sync::{Arc, Mutex, Condvar};
use std::thread;
use std::time::Duration;
struct SharedData{
 mutex:Mutex<bool>,
 cv:Condvar
}
impl SharedData{
 //Metodo costruttore
 pub fn new(condition:bool)->Self{
 SharedData { mutex:Mutex::new(condition), cv:Condvar::new() }
 }
 pub fn change_and_notify(&self){
 let mut data=self.mutex.lock().unwrap();
 *data = true;
 //Mandiamo la notifica alla condvar
 self.cv.notify_one();
 }
 pub fn looper(&self){
 loop {
 //Il thread aspetta per una notifica. Nel caso siano passati 100 misisecondi smette di aspettare
 let lock: (MutexGuard(bool), WaitTimeoutResult) = self.cv.wait_timeout(
 self.mutex.lock().unwrap(), Duration::from_millis(100)).unwrap();
 if *lock.0 == true {
 //Il thread ha ricevuto una notifica dato che il valore è stato cambiato, quindi esco
 print!("Il valore è cambiato quindi posso uscire dal ciclo correttamente ");
 if !lock.1.timed_out() {
 println!("e non c'è stato time-out");
 }
 break
 }
 if lock.1.timed_out() {
 println!("E' scaduto il timer ma il valore non è cambiato. Ricomincio il ciclo");
 }
 }
 }
}

```



```
fn main(){
 let shared = Arc::new(SharedData::new(false));
 let shared2 = Arc::clone(&shared);

 let mut handles = vec![]; //vettore dei threads creati per poi fare le dovute join

 handles.push(thread::spawn(move|| {
 shared2.looper();
 }));

 handles.push(thread::spawn(move|| {
 //Aspetto prima di mandare la notifica
 thread::sleep(Duration::from_secs(1));
 shared.change_and_notify();
 }));

 //Join finali
 for handle in handles {
 handle.join().expect("Error");
 }
}
```



condvar6.rs

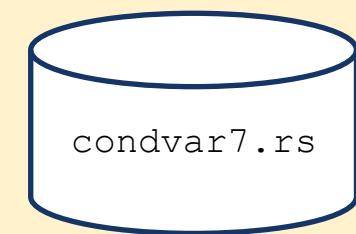
```
use std::sync::{Arc, Mutex, Condvar};
use std::thread;
use std::time::Duration;

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

 thread::spawn(move|| {
 let (lock, cvar) = &*pair2;
 let mut pending = lock.lock().unwrap();
 *pending = true;
 thread::sleep(Duration::from_secs(1));
 // We notify the condvar that the value has changed.
 cvar.notify_one();
 });
}

// wait for the thread to start up
let (lock, cvar) = &*pair;
let result = cvar.wait_timeout_while(
 lock.lock().unwrap(),
 Duration::from_millis(100),
 |&mut pending| !pending,
).unwrap();
if result.1.timed_out() {
 // timed-out without the condition ever evaluating to false.
 println!("Time out with {}", *result.0);
}
else {
 println!("Not time out with {}", *result.0);
}

}
```



condvar7.rs

# Barrier

- Le barriere (**Barrier**) sono strumenti di sincronizzazione che consentono a un certo numero di thread di attendersi l'un l'altro a un determinato punto di sincronizzazione.
- In Rust, la libreria standard **std::sync::Barrier** offre una Barrier che può essere utilizzata per questo scopo
- Un thread che arriva al punto di sincronizzazione chiama la funzione **wait()**.

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

fn main() {
 let num_threads = 10;

 // Crea una barriera per sincronizzare num_threads thread
 let barrier = Arc::new(Barrier::new(num_threads));

 let mut handles = Vec::with_capacity(num_threads);

 for i in 0..num_threads {
 let barrier_clone = Arc::clone(&barrier);
 let handle = thread::spawn(move || {
 println!("Thread {} sta facendo un po' di lavoro", i);

 // Simula il lavoro con una sleep crescente per thread
 thread::sleep(std::time::Duration::from_secs((i+1).try_into().unwrap()));

 println!("Thread {} è arrivato alla barriera", i);
 // Attende che tutti i thread raggiungano la barriera
 barrier_clone.wait();

 println!("Thread {} ha superato la barriera", i);
 });
 handles.push(handle);
 }
 for handle in handles { // Attende che tutti i thread completino il loro lavoro
 handle.join().unwrap();
 }
}
```



# Modelli di concorrenza

**Struttura dati condivisa:** ci si sincronizza per comunicare  
**Scambio di messaggi:** si comunica per sincronizzarsi.



# Scambio di messaggi

- RUST ci mette a disposizione una primitiva elementare che si chiama channel
- Un channel è un tubo a 2 estremità: quella di ingresso e quella di uscita. All'interno del channel possono essere presenti dei messaggi.
- Chi conosce l'estremità di ingresso può produrre un dato e cederlo al canale. Il dato viene prodotto e affidato al canale. Sta nel canale fino a quando qualcun altro, nell'estremità di uscita non cerca di leggerlo. Chi cerca di leggerlo acquisisce il possesso di quel dato e ci fa quello che vuole
- Il ricevitore legge il valore solo se quell'altro lo ha prodotto.
- Contemporaneamente ottengo 2 informazioni:
  - So che l'altro ha fatto un pezzo
  - So anche che pezzo ha fatto.

- Prima di creare i 2 thread che hanno bisogno di parlarsi preparo il channel
- Do l'estremità di ingresso al thread sender
- Do l'estremità di uscita al thread receiver
- Il canale conserva l'ordine con cui sono stati scritti i messaggi.
- Chi legge, legge alla velocità che può. Finchè non c'e' nulla nel tubo, rimane bloccato, senza consumare CPU.
- Non ho bisogno di inventarmi la condition variable, perche' se il canale è vuoto, l'operazione di lettura mi sospende automaticamente. Appena qualcuno inietta qualcosa nel canale, se io sono in attesa, mi sveglio e lo prendo. Se non sono in attesa, sta nel canale.
- Vado nel canale, prendo il dato e poi torno nel canale a vedere se c'è un altro dato.

- Il Channel ci offre un modo per far sapere al receiver che non potrà più ricevere nulla perché non ci sono più sender: non appena tutti i sender vanno via, il receiver, se era anche bloccato in una attesa, esce con un errore che dice che non ci sono più sender e quindi può smettere, senza aver bisogno di sapere a priori di quanti pezzi dovrà aspettare.
- Parallelamente, anche il sender riceve una indicazione. Se l'unico receiver andasse via (ossia viene distrutto e se ne fa il drop), un tentativo di invio fallisce poiché il channel sa che non c'è più la possibilità di ricevere il dato.

- Il producer crea un dato che sta possedendo, quando lo vogliono comunicare al ricevitore, lo cedono al canale. Questo permette al producer di non avere nessuna idea dell'identità del consumer. L'unica cosa che conosce il producer è il canale. Il canale lo renderà disponibile appena possibile.
- Il canale non è limitato. Io posso mettere nel canale quanti messaggi voglio (unbounded). Se quell'altro non sta leggendo perché è occupato a fare cose stanno nel canale. E' come se fosse un tubo che si allunga, si allunga, si allunga per contenere le cose.
- Appena il ricevitore chiede di leggere, legge il primo (quello più vecchio).
- Il canale mantiene l'ordine di ricezione. I messaggi vengono letti nell'ordine con cui sono stati inseriti.

- Quando l'ultimo sender esce di scena e il canale diventa vuoto, il receiver riceve un error (RecvError). Questo permette di capire, da parte del receiver, che lui deve finire.
- Non basta che siano morti tutti i sender, se ho ancora dei messaggi nel canale, questi vengono svuotati.
- Il canale mi garantisce che tutto quello che lui ha preso in carico verrà smaltito e lo consegnerà (ammesso che vi sia un receiver).
- Paralleamente il sender sa che tutto quello che mette nel canale finirà a destinazione, a patto che ci sia una destinazione. Se per qualche motivo la destinazione cessa di esistere, perché viene fatto il drop, ciò che c'è già nel canale, va perduto (perché nessuno è più in grado di leggerlo) e non posso più aggiungere nulla. Ogni eventuale tentativo di aggiunta ulteriore da origine ad un SendError che mi informa che non c'è più un receiver

# 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 (tupla) formata da una `struct Sender<T>` ed una `struct Receiver<T>`
  - Tutti i dati inviati tramite il metodo `send(...)` della 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 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
- L'implementazione fornità gode della proprietà ***multiple producer - single consumer*** ovvero permette di creare più cloni dell'oggetto `sender`
  - Mentre obbliga ad avere una singola copia dell'oggetto `receiver`

# Condivisione di messaggi

- 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)
- Un numero arbitrario di messaggi può essere scambiato sul canale
  - A condizione che il ricevitore sia attivo
  - Se il ricevitore viene deallocato, eventuali tentativi di invio falliscono con la generazione di un valore di tipo **SendError<T>**
  - Se tutti i trasmettitori vengono deallocati, tentativi di lettura sul ricevitore falliscono con la generazione di un valore di tipo **RecvError**

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

fn main() {
 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 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);
 }
}
```



```

use std::sync::mpsc::{Sender, channel, Receiver};
use std::thread;
struct SharedMsg{
 tx:Sender<String>,
 rx:Receiver<String>
}
impl SharedMsg{
 pub fn new()->Self{
 //Creazione del canale
 let (tx,rx)=channel::<String>();
 SharedMsg { tx: tx, rx: rx }
 }
}
fn main() {
 let mut handles = vec![];
 let shared=SharedMsg::new();
 let tx1=shared.tx.clone();
 handles.push(thread::spawn(move|| {
 if tx1.send("Ciao".to_string()).is_err()==true { println!("Errore nell'invio del messaggio"); }
 }));
 let tx2=shared.tx.clone();
 handles.push(thread::spawn(move|| {
 if tx2.send("Come stai?".to_string()).is_err()==true{ println!("Errore nell'invio del messaggio"); }
 }));
 // Occorre chiudere il canale,
 // Senza questo il receiver continuerebbe a rimanere in attesa di messaggi
 drop(shared.tx);
 while let Ok(msg)=shared.rx.recv(){
 println!("Messaggio ricevuto:");
 println!("{}",&msg);
 }
 for handle in handles {
 handle.join().expect("Error");
 }
}

```



```

use std::sync::mpsc::{channel, Receiver, Sender};
use std::thread;
use std::sync::{Arc, Mutex};
use std::time::Duration;

#[derive(Debug)]
pub struct Task {
 id: usize,
 testo: String,
}

fn main() {
 //Definisco un canale con il quale dialogare, in cui vi sono più producer e 1 consumer
 let (task_sender, task_receiver) = channel::<Task>();

 // Per condividere il risultato tra thread in mutua esclusione, ed essenzialmente per tener traccia di quante task
 //sono state effettuate
 let result_counter = Arc::new(Mutex::new(0));

 // Faccio partire 2 thread
 for i in 0..2 {
 let tx = task_sender.clone();
 thread::spawn(move || {
 for j in 0..5 {
 let task = Task {
 id: i * 5 + j,
 testo: format!("Task {} from Producer {}", i * 5 + j, i),
 };
 tx.send(task).unwrap(); //invio la task
 thread::sleep(Duration::from_millis(200)); //Aspetto per renderlo più deterministico
 }
 });
 }
}

```

channelmutex.rs

segue

```
let result_counter_t = Arc::clone(&result_counter);
thread::spawn(move || {
 loop {
 //Per ricevere il dato e poi stamparlo mi metto in loop sul receiver
 match task_receiver.recv() {
 Ok(task) => {
 println!("Consumer received task: {:?}", task);

 thread::sleep(Duration::from_millis(400));

 // Incremento il mutex per tener traccia delle task effettuate
 {
 let mut counter = result_counter_t.lock().unwrap();
 *counter += 1;
 }
 }
 Err(_) => break, //Esco dal loop quando il channel si chiude
 }
 }
});;

// Aspetto che finiscano tutti i sender
thread::sleep(Duration::from_secs(3));
// Chiudo il canale con il drop del sender
drop(task_sender);

thread::sleep(Duration::from_secs(1));

let result_counter = Arc::clone(&result_counter);

// Stampo il numero di task facendo lock sul mutex
let counter = result_counter.lock().unwrap();
println!("Total tasks processed: {}", counter);
})
```



channelmutex.rs

# Canali sincroni

- La funzione `std::sync::mpsc::sync_channel<T>(bound: usize)` restituisce invece 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 di scrittura
  - Le restanti operazioni offerte da `SyncSender<T>` hanno semantica simile alle corrispondenti offerte da `Sender<T>`

# Canali sincroni

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

fn main() {

 let (sender, receiver) = sync_channel(1);
 // this returns immediately

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

 println!("{}", receiver.recv().unwrap());
 println!("{}", receiver.recv().unwrap());
}
```



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

fn main() {
 let (sender, receiver) = mpsc::sync_channel(0);

 // Questo thread tenta di inviare un valore al canale, ma essendo la capacità 0,
 // si bloccherà fino a quando il valore non verrà letto dal receiver.
 thread::spawn(move || {
 sender.send("Sto trasmettendo un messaggio e ci aspettiamo
 all'appuntamento").unwrap();
 println!("Valore inviato.");
 });

 // Il receiver legge il valore dal canale.
 let received_value = receiver.recv().unwrap();
 println!("Valore ricevuto: {}", received_value);
}
```



rendezvous.rs

# La libreria Crossbeam

- Libreria ben documentata e attivamente mantenuta 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 là dove possibile, oppure ricorrendo all'uso di un lock interno per strutture dati più articolate
  - **Strutture dati concorrenti** - le 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

```
use crossbeam::atomic::AtomicCell;
use std::sync::Arc;
use std::thread::JoinHandle;

fn main() {
 let cell = Arc::new(AtomicCell::<i16>::new(0)) ;

 // Crea due thread che incrementano il valore nella cella
 let handles: Vec<JoinHandle<()>> = (0..4).map(|_| {
 let cell = Arc::clone(&cell) ;
 std::thread::spawn(move || {
 for _ in 0..1000 {
 cell.fetch_add(1);
 }
 })
 }).collect();

 // Attendi che entrambi i thread abbiano terminato
 for handle in handles {
 handle.join().unwrap();
 }

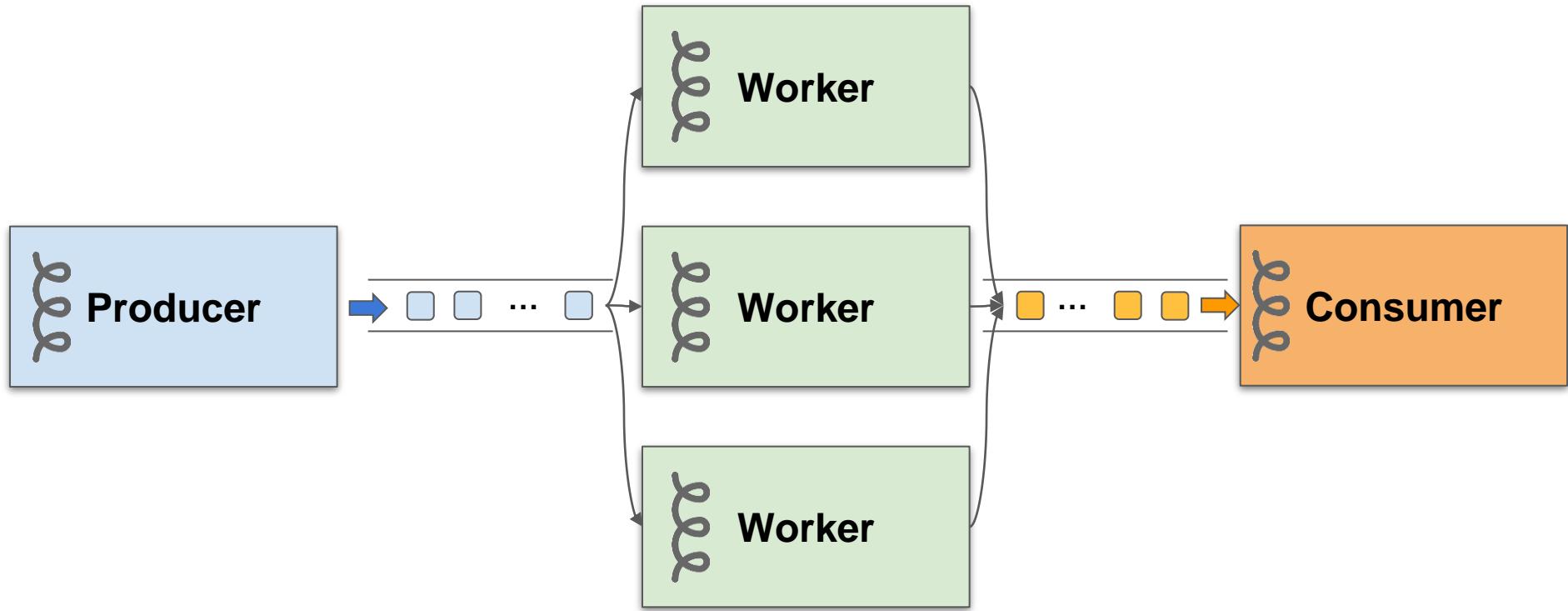
 // Stampa il valore finale nella cella
 println!("Final value: {}", cell.load());
}
```

atomiccell.rs

# Uso della libreria Crossbeam

- 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
  - **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
  - **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

# Fan-Out / Fan-In



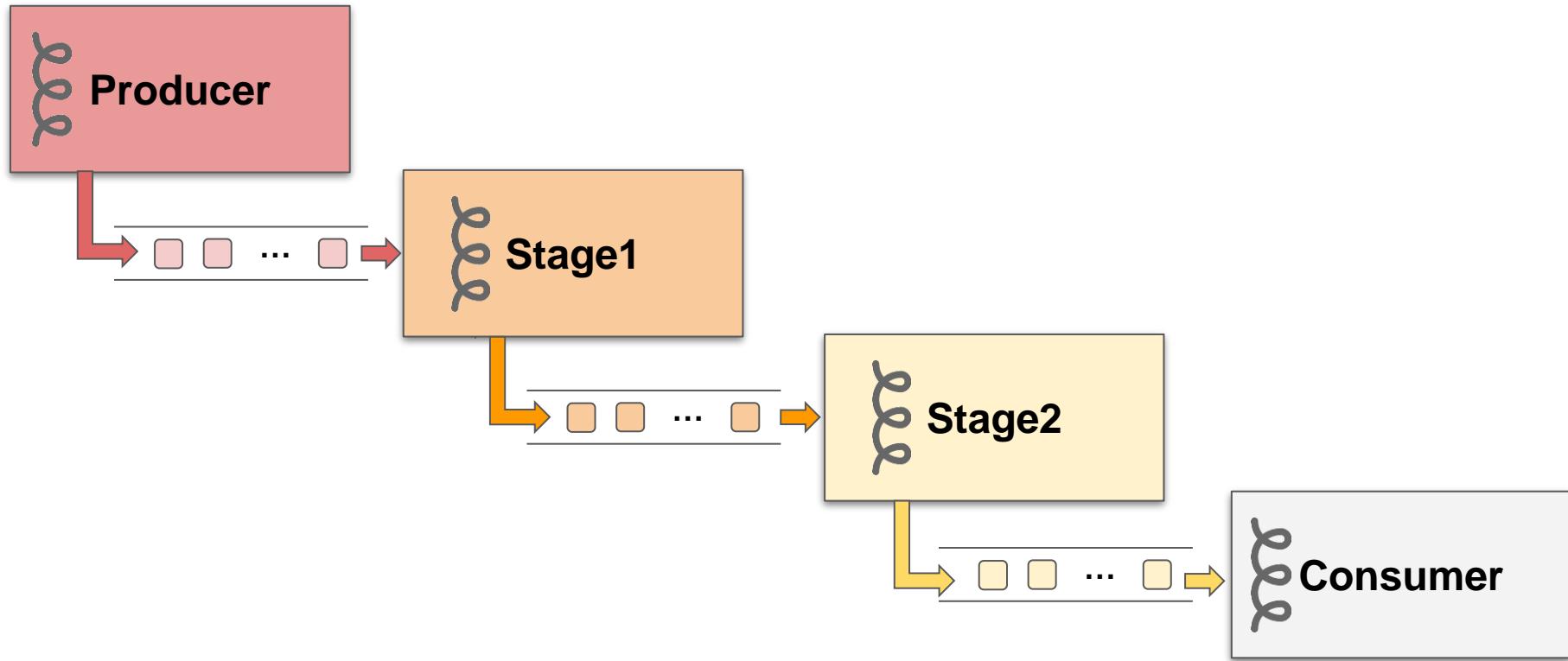
```
use std::thread;
use crossbeam_channel::{bounded, Receiver, Sender};
fn worker(id: usize, rx: Receiver<i32>, tx: Sender<String>) {
 while let Ok(value) = rx.recv() {
 tx.send(format!("W{} ({})", id, value)).unwrap();
 }
}

fn main() {
 let (tx_input, rx_input) = bounded::<i32>(10);
 let (tx_output, rx_output) = bounded::<String>(10);
 let mut worker_handles = Vec::new();
 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)));
 }
 for i in 1..=10
 {
 tx_input.send(i).unwrap();
 }
 drop(tx_input);
 drop(tx_output);

 while let Ok(result) = rx_output.recv() {
 println!("Received result: {}", result);
 }
 for handle in worker_handles { handle.join().unwrap(); }
}
```

fanoutfanin.rs

# Pipeline



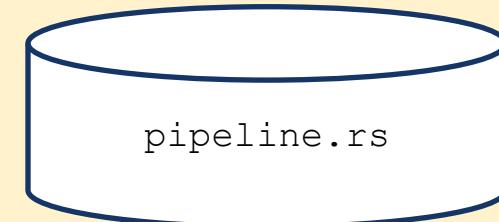
```
use std::thread;
use crossbeam_channel::{bounded, Receiver, Sender};
fn stage_one(rx: Receiver<i32>, tx: Sender<String>) {
 while let Ok(value) = rx.recv() {
 tx.send(format!("S1({})", value)).unwrap();
 }
}
fn stage_two(rx: Receiver<String>, tx: Sender<String>) {
 while let Ok(value) = rx.recv() {
 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);
 let (tx_output, rx_output) = bounded::<String>(10);

 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));

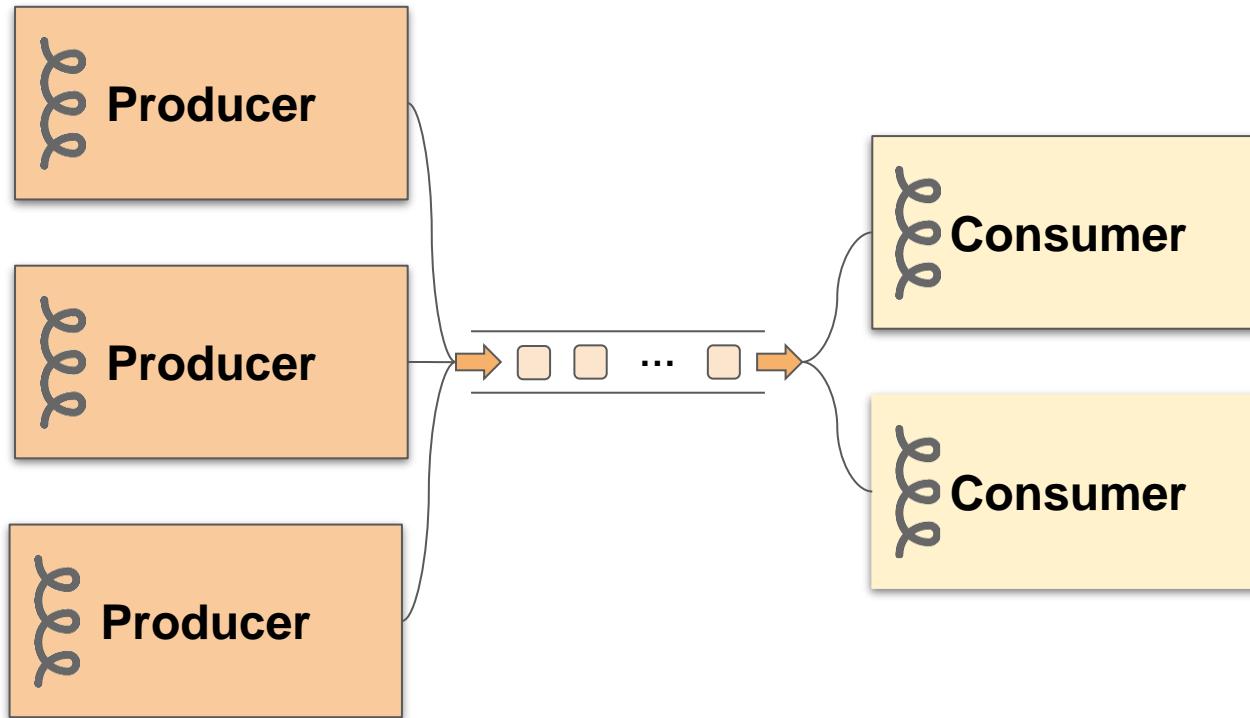
 for i in 1..=10 { tx_input.send(i).unwrap(); }
 drop(tx_input);

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

 stage_one_handle.join().unwrap();
 stage_two_handle.join().unwrap();
}
```



# Producer/consumer



```
use std::thread;
use crossbeam_channel::{bounded, Receiver, Sender};

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();
 for i in 0..3 {
 let tx = tx.clone();
 handles.push(thread::spawn(move || producer(i, tx)));
 }
 for i in 0..2 {
 let rx = rx.clone();
 handles.push(thread::spawn(move || consumer(i, rx)));
 }
 drop(tx);
 for handle in handles { handle.join().unwrap(); }
}
```

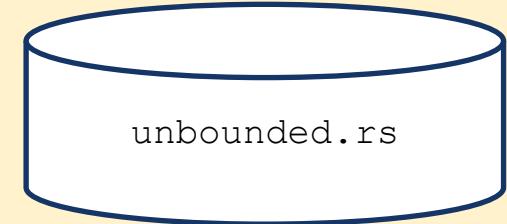


```
use std::thread;
use crossbeam_channel::{unbounded};

fn main() {
 let (s, r) = unbounded();
 let mut vettore: Vec<i32> = Vec::new();

 thread::spawn(move || {
 // It can send any number of messages into the channel without blocking.
 for i in 0..50 {
 s.send(i).unwrap();
 }
 });

 while let Ok(result) = r.recv() {
 vettore.push(result);
 }
 println!("{}: {:?}", vettore);
}
```

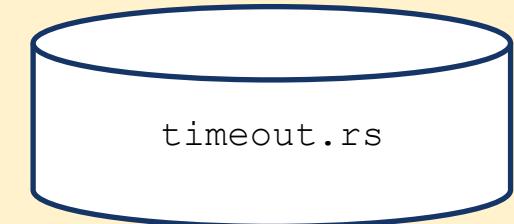


# After e tick

- La funzione **after** crea un ricevitore che consegna un messaggio dopo una certa durata di tempo
- La funzione **tick** crea un ricevitore che consegna un messaggio ad una determinata frequenza di trasmissione.

# Time out

```
use crossbeam_channel::select;
use crossbeam::channel;
use std::thread;
use std::time::{Duration, Instant};
fn main() {
 let (s, r) = channel::bounded(1);
 let after = channel::after(Duration::from_secs(5)); // Timeout dopo 5 secondi
 thread::spawn(move || {
 // Simula un'operazione di calcolo intensivo
 let start = Instant::now();
 while start.elapsed() < Duration::from_secs(8) {
 // Fa qualcosa di computazionalmente intenso
 }
 let _ = s.send("Operazione completata").unwrap();
 });
 select! {
 recv(r) -> msg => println!("{}: {}", msg.unwrap(),), // Riceve il messaggio
 recv(after) -> _ => println!("Timeout! Operazione non completata"), // Timeout
 }
 let (s, r) = channel::bounded(1);
 let after = channel::after(Duration::from_secs(5)); // Timeout dopo 5 secondi
 thread::spawn(move || {
 // Simula un'operazione di calcolo intensivo
 let start = Instant::now();
 while start.elapsed() < Duration::from_secs(1) {
 // Fa qualcosa di computazionalmente intenso
 }
 let _ = s.send("Operazione completata").unwrap();
 });
 select! {
 recv(r) -> msg => println!("{}: {}", msg.unwrap(),), // Riceve il messaggio
 recv(after) -> _ => println!("Timeout! Operazione non completata"), // Timeout
 }
}
```



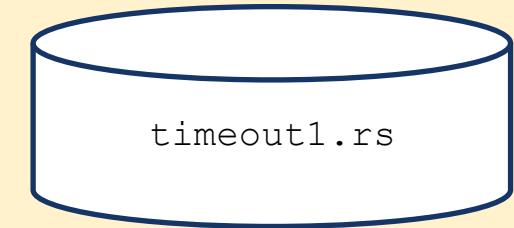
select! è un costrutto fornito dalla crate crossbeam che permette di ascoltare più canali contemporaneamente e di eseguire azioni basate su quale canale riceve un valore per primo.

```

use crossbeam_channel::select;
use crossbeam::channel;
use std::thread;
use std::time::{Duration, Instant};
fn main() {
 let (s, r) = channel::bounded(1);
 let after = channel::after(Duration::from_secs(5)); // Timeout dopo 5 secondi

 thread::spawn(move || {
 for i in 0..10 {
 // Simula un ciclo di operazioni di calcolo intensivo
 let start = Instant::now();
 while start.elapsed() < Duration::from_secs(i) {
 // Fa qualcosa di computazionalmente intenso
 }
 let _ = s.send("Operazione completata").unwrap();
 }
 });
 loop {
 select! {
 recv(after) -> _ => {
 println!("Timeout! Operazione non completata"); // Timeout
 break;
 }
 recv(r) -> msg => println!("{}", msg.unwrap()), // Riceve il messaggio
 }
 }
}

```



```
use std::time::{Duration, Instant};
use crossbeam_channel::tick;
fn main()
{
let start = Instant::now();
let ticker = tick(Duration::from_millis(100));

for _ in 0..5 {
 ticker.recv().unwrap();
 println!("Tempo trascorso: {:?}", start.elapsed());
}
}
```

tick1.rs

```
use std::thread;
use std::time::Duration;
use crossbeam::channel;

fn main() {
 let (s, r) = channel::unbounded();
 let tick = channel::tick(Duration::from_secs(1));
 // Genera un impulso ogni secondo

 let mut counter = 0;
 thread::spawn(move || {
 for _ in 0..5 {
 let _ = tick.recv(); // Riceve l'impulso
 s.send("Impulso ricevuto").unwrap();
 }
 });

 for _ in 0..5 {
 println!("{}", r.recv().unwrap());
 // Stampa "Impulso ricevuto" ogni secondo
 counter += 1;
 }
 println!("Ho ricevuto {} impulsi", counter);
}
```



# TdE

- La struttura MultiChannel implementa il concetto di canale con molti mittenti e molti ricevitori
- I messaggi inviati a questo tipo di canale sono composti da singoli byte che vengono recapitati a tutti i ricevitori attualmente collegati.
- Riferimenti a tipi:

```
use std::result::Result;
use std::sync::mpsc::{Receiver, SendError};
```
- Metodi:
  - **new() -> Self**
  - crea un nuovo canale senza alcun ricevitore collegato
  - **subscribe(&self) -> Receiver<u8>**
  - collega un nuovo ricevitore al canale: da quando questo metodo viene invocato, gli eventuali byte inviati al canale saranno recapitati al ricevitore.
  - Se il ricevitore viene eliminato, il canale continuerà a funzionare inviando i propri dati ai ricevitori restanti (se presenti), altrimenti ritornerà un errore
  - **send(&self, data: u8) -> Result<(), SendError<u8>>**
  - invia a tutti i sottoscrittori un byte
  - se non c'è alcun sottoscrittore, notifica l'errore indicando il byte che non è stato trasmesso

```
use std::result::Result;
use std::sync::mpsc::{channel, Receiver, Sender, SendError};
use std::sync::Mutex;
use std::thread;
pub struct MultiChannel {
 channels: Mutex<Vec<Sender<u8>>>,
}
impl MultiChannel {
 fn new() -> Self {
 MultiChannel {
 channels: Mutex::new(Vec::new()),
 }
 }
 fn send(&self, data: u8) -> Result<(), SendError<u8>> {
 let channels = self.channels.lock().unwrap();
 if channels.is_empty() {
 return Err(SendError(data));
 }
 for i in 0..channels.len() {
 channels[i].send(data).unwrap();
 }
 Ok(())
 }
 fn subscribe(&self) -> Receiver<u8> {
 let (tx, rx) = channel();
 let mut channels = self.channels.lock().unwrap();
 channels.push(tx);
 rx
 }
}
```



```
fn main() {
 let mut handles = Vec::new();
 {
 let multi_channel = MultiChannel::new();

 // Subscriber 1
 let rx1 = multi_channel.subscribe();
 handles.push(thread::spawn(move || {
 while let Ok(data) = rx1.recv() {
 println!("Subscriber 1 received: {}", data);
 }
 }));

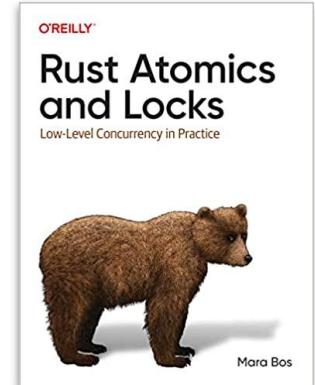
 // Subscriber 2
 let rx2 = multi_channel.subscribe();
 handles.push(thread::spawn(move || {
 while let Ok(data) = rx2.recv() {
 println!("Subscriber 2 received: {}", data);
 }
 }));

 // Send data
 multi_channel.send(10).unwrap();
 multi_channel.send(20).unwrap();
 multi_channel.send(30).unwrap();
 }
 for handle in handles { handle.join().unwrap(); }
}
```



# Per saperne di più

- The C11 and C++11 Concurrency Model, 2014, Mark Batty
  - <https://www.cs.kent.ac.uk/people/staff/mjb211/docs/toc.pdf>
  - Tesi di dottorato in cui viene formalizzato il modello di memoria dei linguaggi C++11/C11
- LLVM Atomic Instructions and Concurrency Guide -
  - <https://llvm.org/docs/Atomics.html>
  - Modello di accesso concorrente alla memoria offerto da LLVM su cui si basa Rust
- The Little Book of Semaphores
  - <https://greenteapress.com/seashores/LittleBookOfSemaphores.pdf>
- Green threads explained in 200 lines of code
  - <https://cfsamson.gitbook.io/green-threads-explained-in-200-lines-of-rust/>
- Rust Atomics and Locks - Low-Level Concurrency in Practice
  - Mara Bos - O'Reilly 2023 - ISBN: 978-1-098-11944-7
  - Trattazione dettagliata e efficace del modello di concorrenza in Rust



# Only for Rustaceans



- Il "work stealing" è una tecnica di bilanciamento del carico utilizzata in sistemi di calcolo parallelo per migliorare l'efficienza e l'utilizzo dei processori.
- Il work stealing è un metodo in cui i thread che rimangono inattivi rubano lavoro dai thread che sono ancora occupati. Questo meccanismo aiuta a bilanciare il carico di lavoro tra i thread, assicurando che tutti i processori disponibili siano utilizzati in modo efficiente
- Come funziona:
  - Il task è suddiviso in unità più piccole (sub-task) che possono essere eseguite in parallelo.
  - Ogni thread riceve una coda di lavoro contenente questi sub-task.
  - Ogni thread esegue i task dalla propria coda.
  - Se un thread esaurisce i task nella propria coda, diventa inattivo e tenta di rubare task dalle code di altri thread.
  - Un thread inattivo sceglie casualmente un altro thread e cerca di rubare un task dalla coda di quest'ultimo. Il furto di solito avviene dalla coda opposta rispetto alla direzione normale di estrazione del task, per minimizzare il conflitto (es. FIFO per il thread proprietario e LIFO per il ladro).
- Rayon è una libreria in Rust che realizza il work stealing per gestire il parallelismo. Rayon facilita la parallelizzazione delle operazioni su collezioni e altre strutture dati, ottimizzando automaticamente la distribuzione dei task tra i thread utilizzando il work stealing.

# Libreria Rayon

- Uno degli strumenti che offre per il parallelismo è la funzione `join()`, che permette di eseguire due task in parallelo e aspettare che entrambi finiscano prima di procedere.
- La funzione `join` prende due chiusure e le esegue in parallelo. Attende che entrambe finiscano e restituisce una tupla con i risultati di entrambe le chiusure.
- `join` fornisce un modo semplice per eseguire due task in parallelo senza dover gestire esplicitamente i thread
- Rayon ottimizza automaticamente l'uso delle risorse hardware, rendendo l'esecuzione parallela efficiente.

```
extern crate rayon;

use rayon::prelude::*;
use rayon::join;

fn main() {
 // Definiamo due compiti che eseguiremo in parallelo
 let task1 = || {
 println!("Esecuzione del task 1");
 // Simuliamo un lavoro computazionale
 let sum: i32 = (1..=25_000).sum();
 println!("Il risultato del task 1 è: {}", sum);
 sum
 };

 let task2 = || {
 println!("Esecuzione del task 2");
 // Simuliamo un lavoro computazionale diverso
 let product: i32 = (1..=12).product();
 println!("Il risultato del task 2 è: {}", product);
 product
 };

 // Eseguiamo i due compiti in parallelo
 let (result1, result2) = join(task1, task2);

 // Utilizziamo i risultati
 println!("Il risultato combinato è: {}", result1 + result2);
}
```



- Rayon permette di personalizzare il livello di parallelismo configurando il pool di thread.
- Ogni thread ha una propria coda di task.
- Il metodo `par_iter()` è un iteratore che distribuisce automaticamente il lavoro su più thread, utilizzando il pool di thread gestito da Rayon
- Quando si utilizza `par_iter` (o altre funzioni parallele di Rayon), il lavoro viene suddiviso in task più piccoli che vengono distribuiti tra i thread del pool.
- I thread eseguono i task dalla propria coda. Se un thread esaurisce i task, tenta di rubare task dalle code di altri thread.
- Questo bilanciamento dinamico aiuta a mantenere tutti i thread occupati, migliorando l'efficienza complessiva e si adatta bene a un numero variabile di processori e thread, rendendolo adatto per applicazioni su sistemi multi-core.
- Si può creare e configurare un pool di thread personalizzato e usarlo per eseguire operazioni parallele

```

use rayon::prelude::*;
use rayon::ThreadPoolBuilder;

fn main() {
 let pool = ThreadPoolBuilder::new().num_threads(4)
 .stack_size(100_000)
 .thread_name(|index| format!("my number - {} ", index))
 .build().unwrap();

 pool.install(|| {
 let v: Vec<i32> = (1..20_000).collect();

 let sum: i32 = v.par_iter()
 .map(|&x| x * 2)
 .sum();

 println!("Somma: {}", sum);
 });
}

```



- `build()` costruisce il pool di thread con le impostazioni specificate
- `install()` esegue un blocco di codice nel contesto del pool di thread