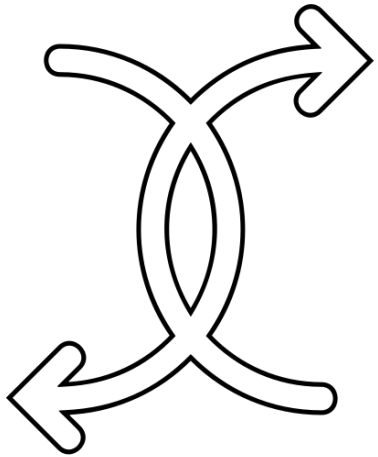


Concorrenza

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
- All'atto della creazione, un processo dispone di un **unico flusso di esecuzione**
  - Thread principale
  - Esso può richiedere allo schedulatore 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
  - 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)
- 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

- 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
- **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 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-informazioni 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

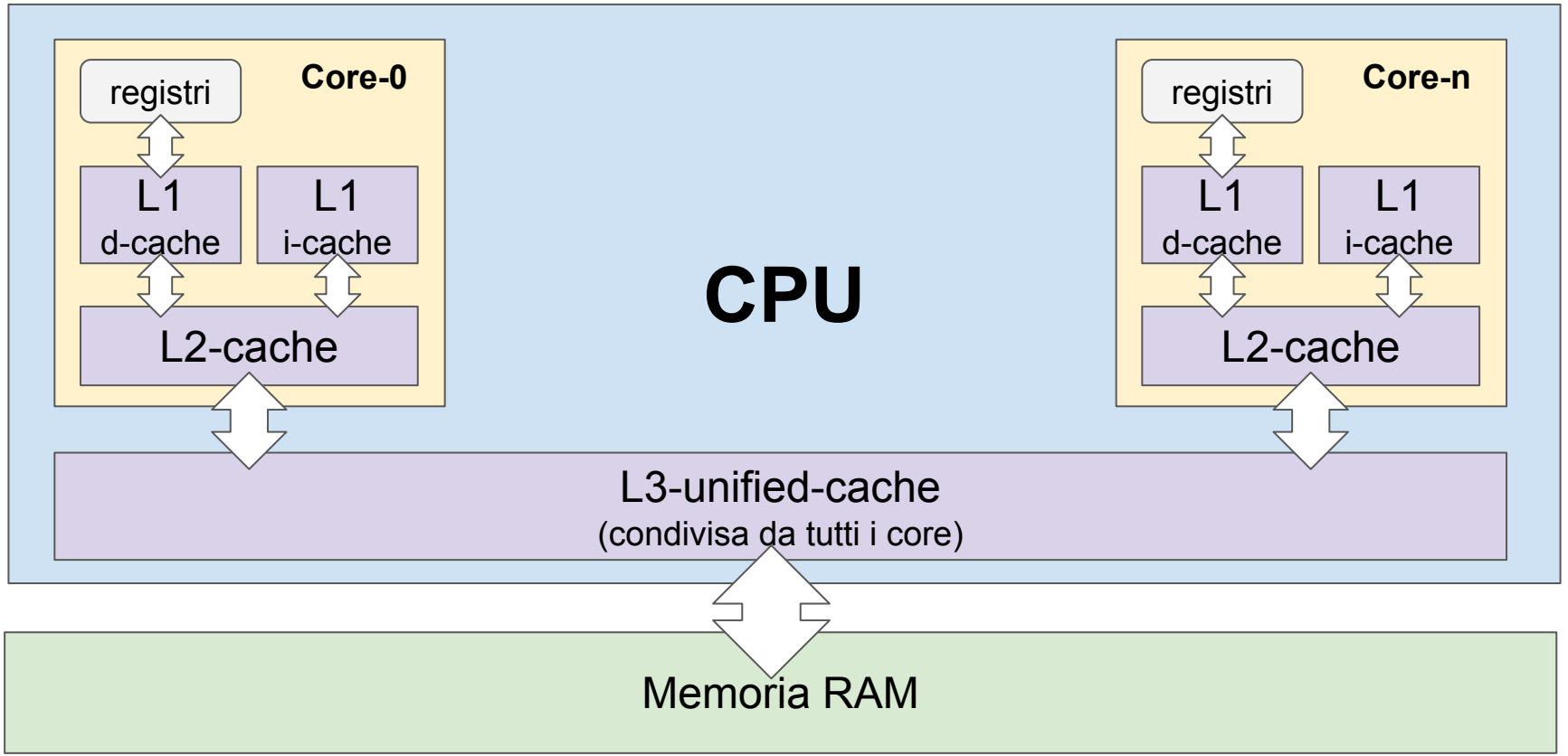
Concorrenza in pratica

- 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

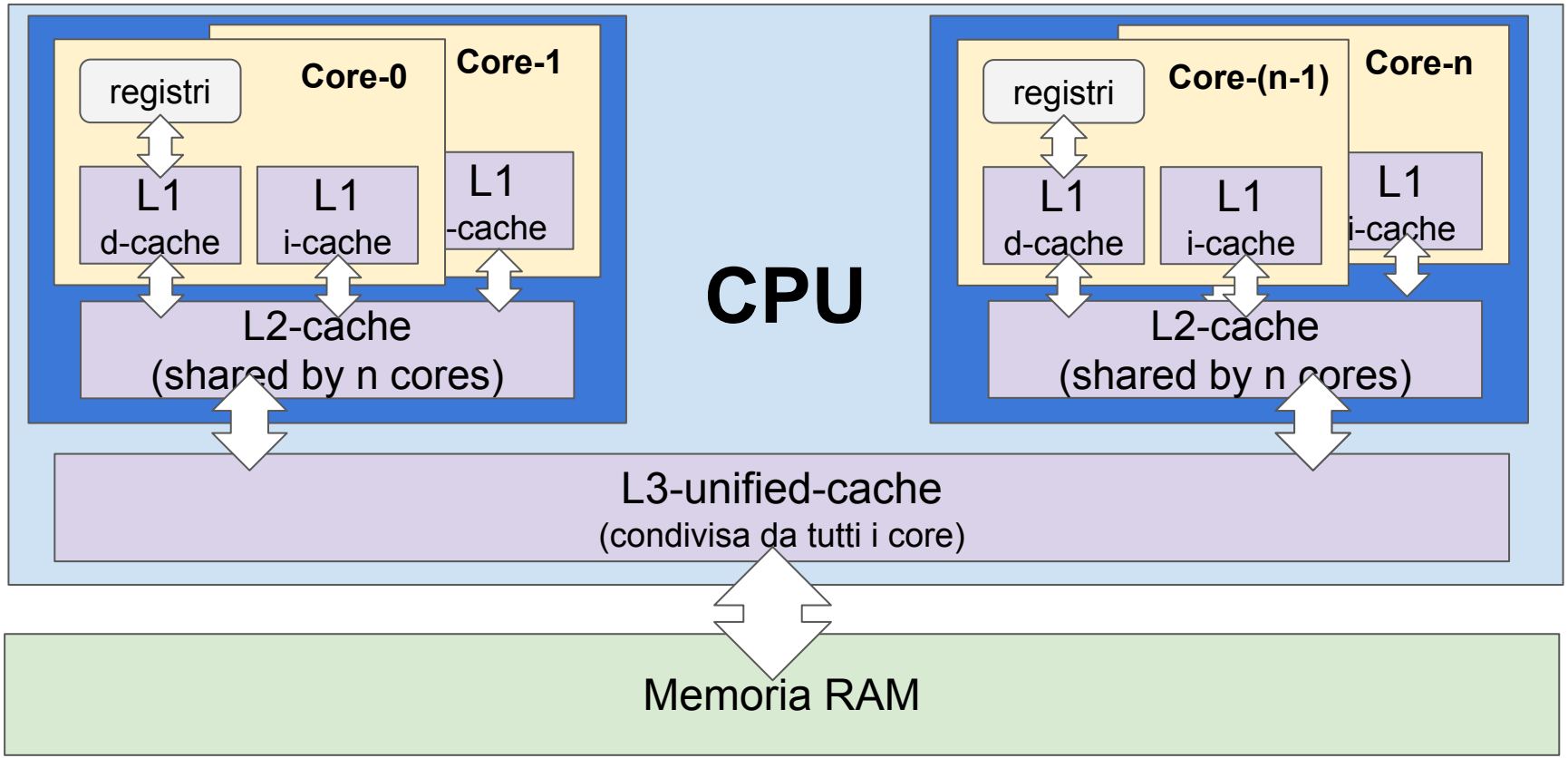
Modello di memoria

- 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

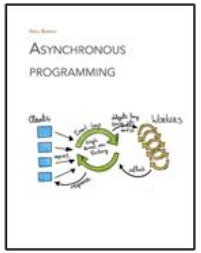


Modello di memoria (packed cores)



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	28 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

- **Atomicità**
  - Quali istruzioni devono avere effetti indivisibili?
  - 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)
- **Visibilità**
  - Sotto quali condizioni, le scritture compiute da un thread sono visibili da un secondo thread?
- **Ordinamento**
  - Sotto quali condizioni gli effetti di più operazioni effettuate da un thread possono apparire ad altri thread in ordine differente?

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 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 *barrier* che consentono l'ordinamento causale rispettivamente per il calcolo degli indirizzi, delle istruzioni, dei dati
- 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

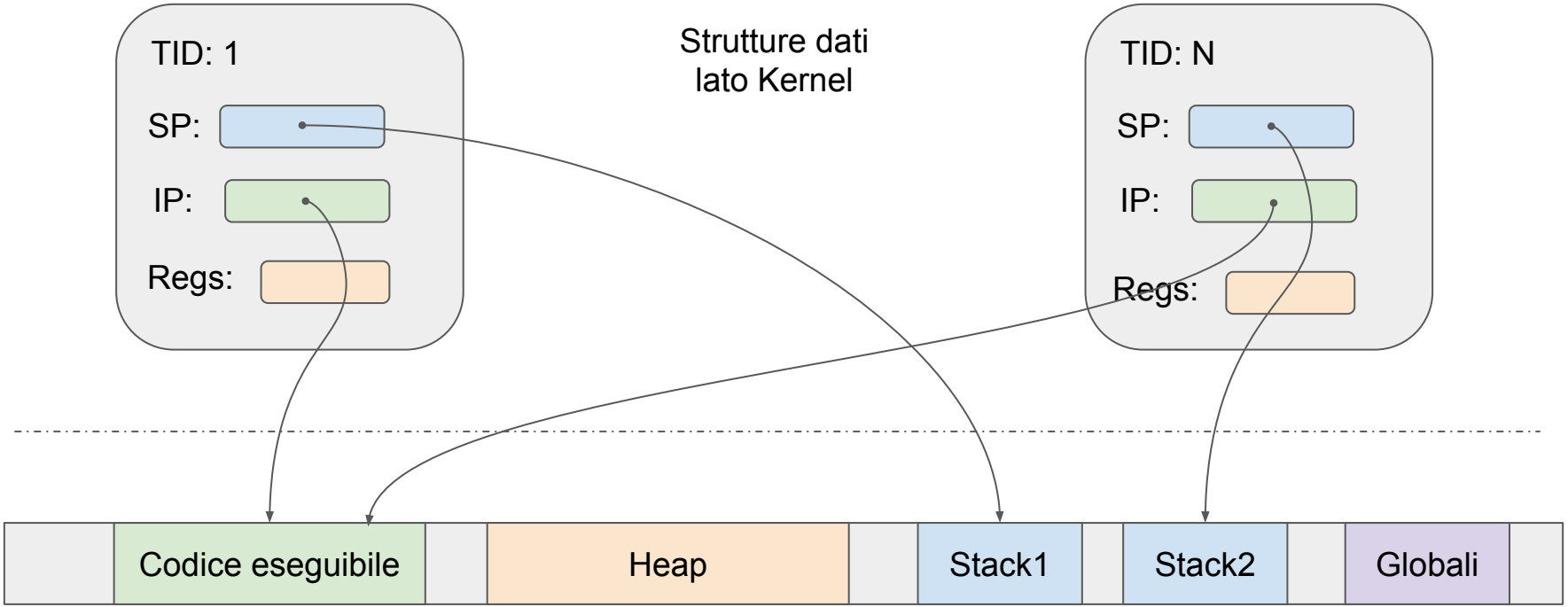
Errori

- L'**uso superficiale** dei costrutti di sincronizzazione porta a blocchi passivi o attivi 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



Spazio di indirizzamento del processo

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

C++

```
#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();
}
```

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

Esecuzione e non determinismo

Rust

```
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();
}
```

```
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"
- Il non determinismo dà origine a **comportamenti del tutto inattesi** in un contesto di elaborazione sequenziale
  - 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

C++

```
#include <iostream>
#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

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

int a=0;

void run() {
  while (a
    fore=a;
    a++;
    ter=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;

Sincronizzazione

- Se due thread cercano di accedere in lettura/scrittura ad una stessa variabile si verifica una **corsa critica**
  - 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

- Alcune delle condizioni citate sono garantite da **apposite istruzioni macchina**
  - Che dipendono dal processore responsabile dell'esecuzione del codice
- Altre richiedono la garanzia di **invarianti a livello sistema** che può essere fornita solo da meccanismi offerti dal sistema operativo
  - 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

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** e **dove** utilizzare la sincronizzazione
  - Un uso sbagliato porta a **risultati disastrosi**
- 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***

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

- **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

Thread in Rust

```
use std::thread;

let thread_join_handle = thread::spawn(move || { //move trasferisce alla funzione
                                                 //il possesso di quanto catturato
     //computazione da eseguire

});

//altre attività

match thread_join_handle.join() {
     Ok(res) => { … },
     Err(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 */}).unwrap();

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

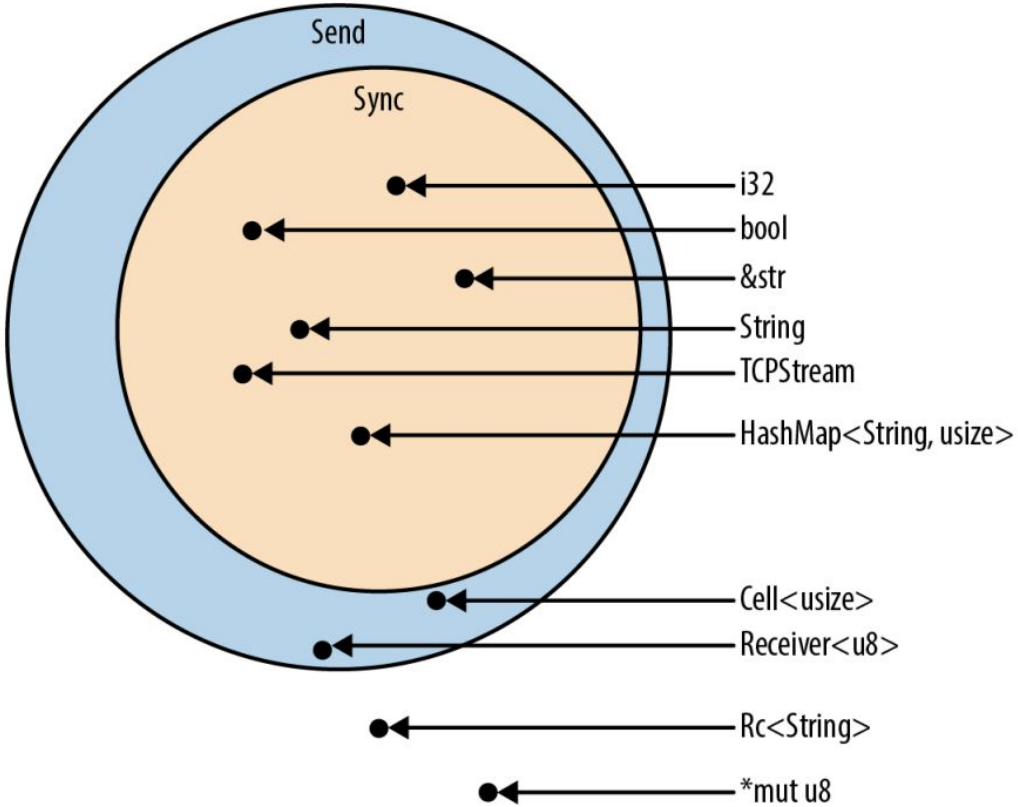
I tratti della concorrenza

- 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**
  - Il tratto **`std::maker::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
- 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 conscio 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
  - I tipi che implementano una mutabilità interna (come `Cell` e `RefCell`) non dispongono di questo tratto

I tratti della concorrenza



I tratti della concorrenza

- 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

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

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

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

Modelli di concorrenza

- La libreria standard di Rust supporta due modelli base per la realizzazione di programmi concorrenti
  1. La condivisione di dati basata su sincronizzazione degli accessi ad una struttura dati condivisa, a cui tutti i thread interessati possono accedere in lettura e scrittura
  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

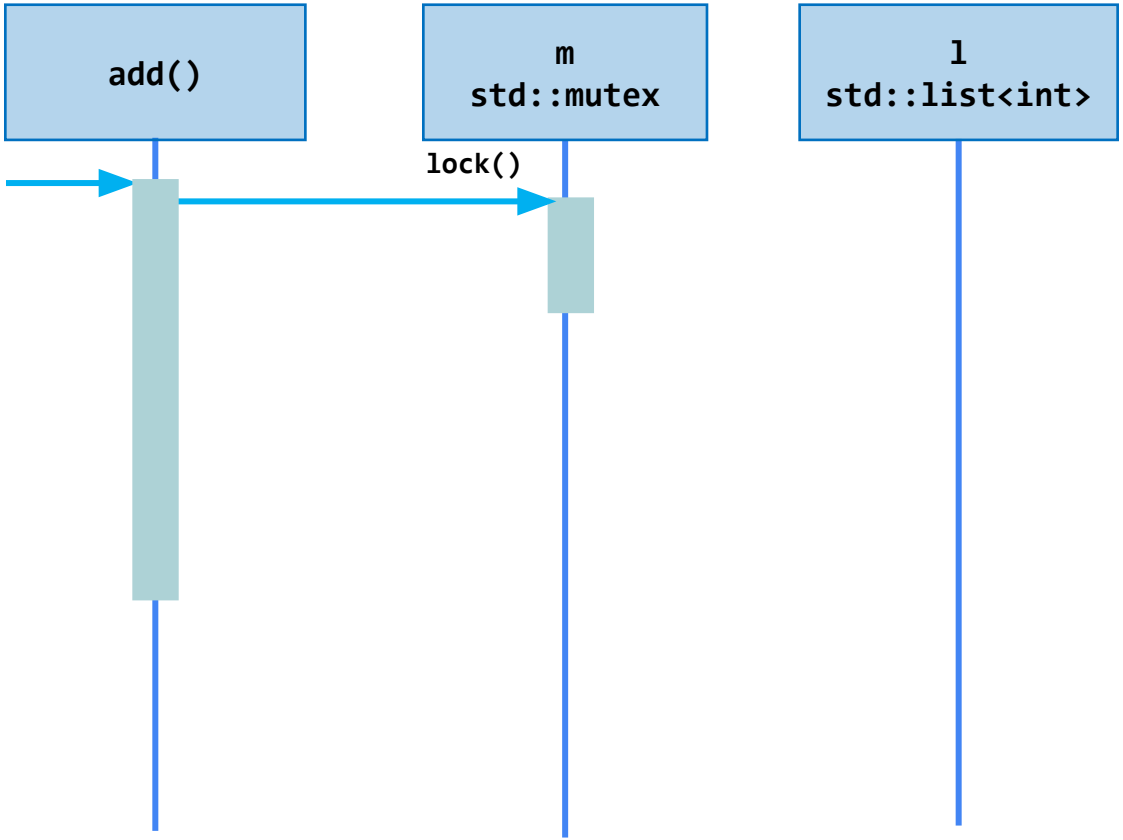
Condivisione dello stato

- 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, come minimo, 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

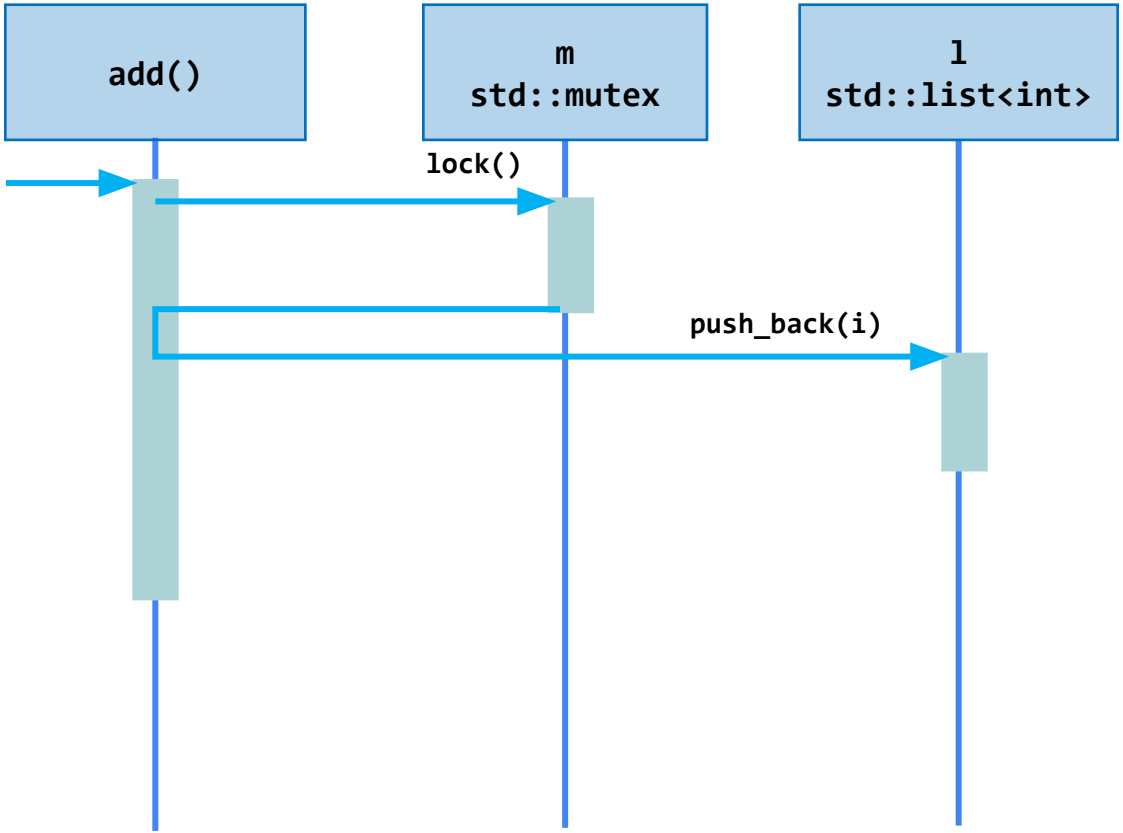
Condivisione dello stato

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

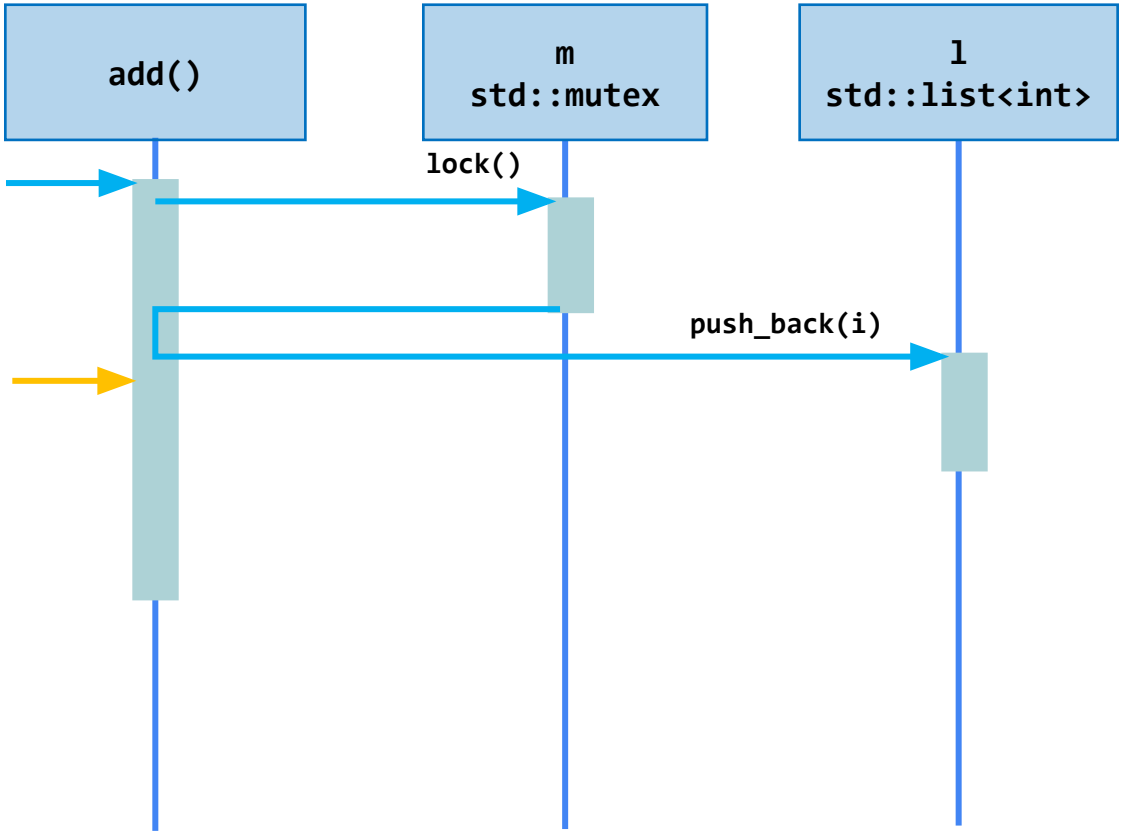
```
std::list<int> l;
std::mutex m;

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



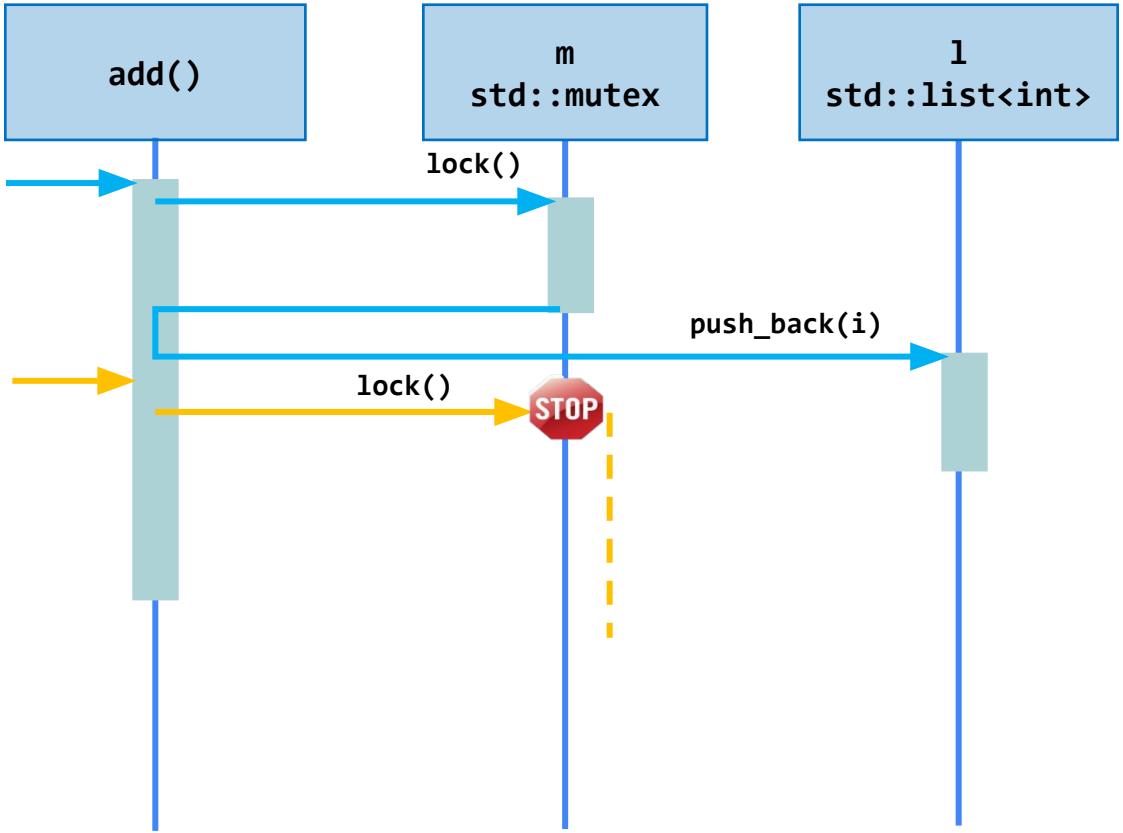
```
std::list<int> l;
std::mutex m;

void add(int i){

  m.lock();

  l.push_back(i);

  m.unlock();

}
```



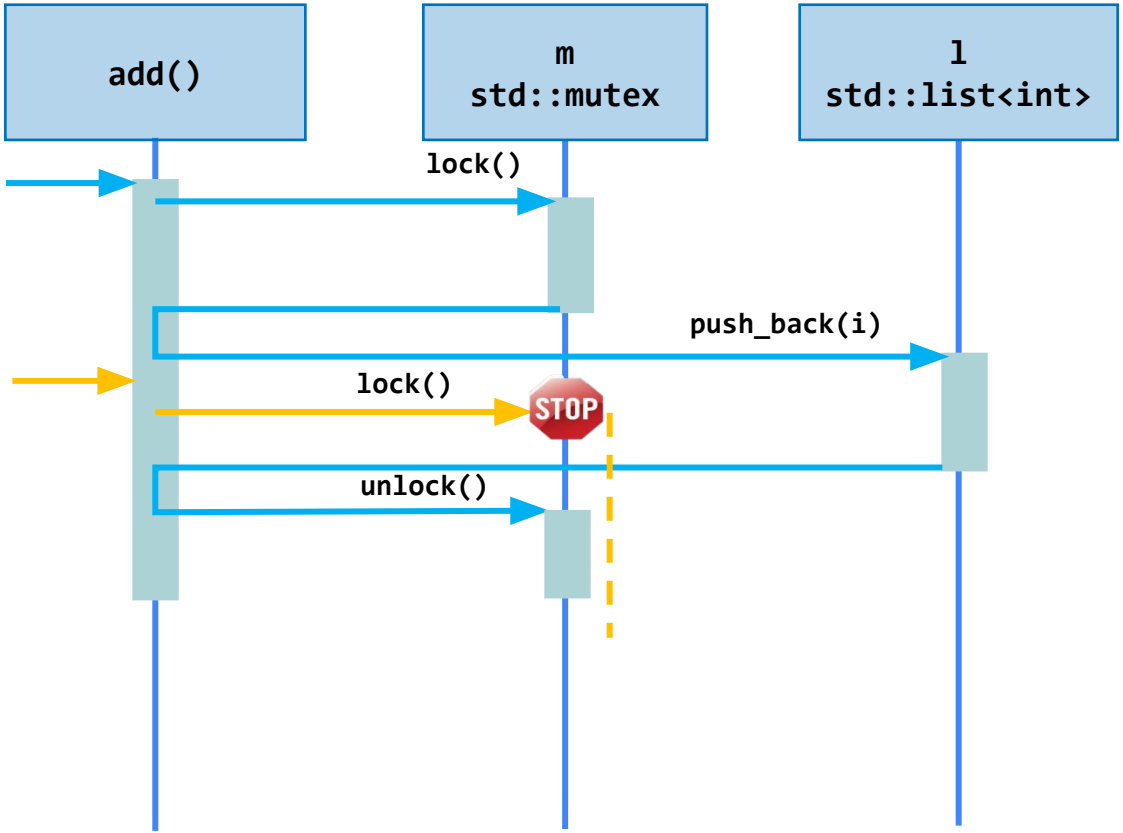
```
std::list<int> l;
std::mutex m;

void add(int i){

  m.lock();

  l.push_back(i);

  m.unlock();

}
```



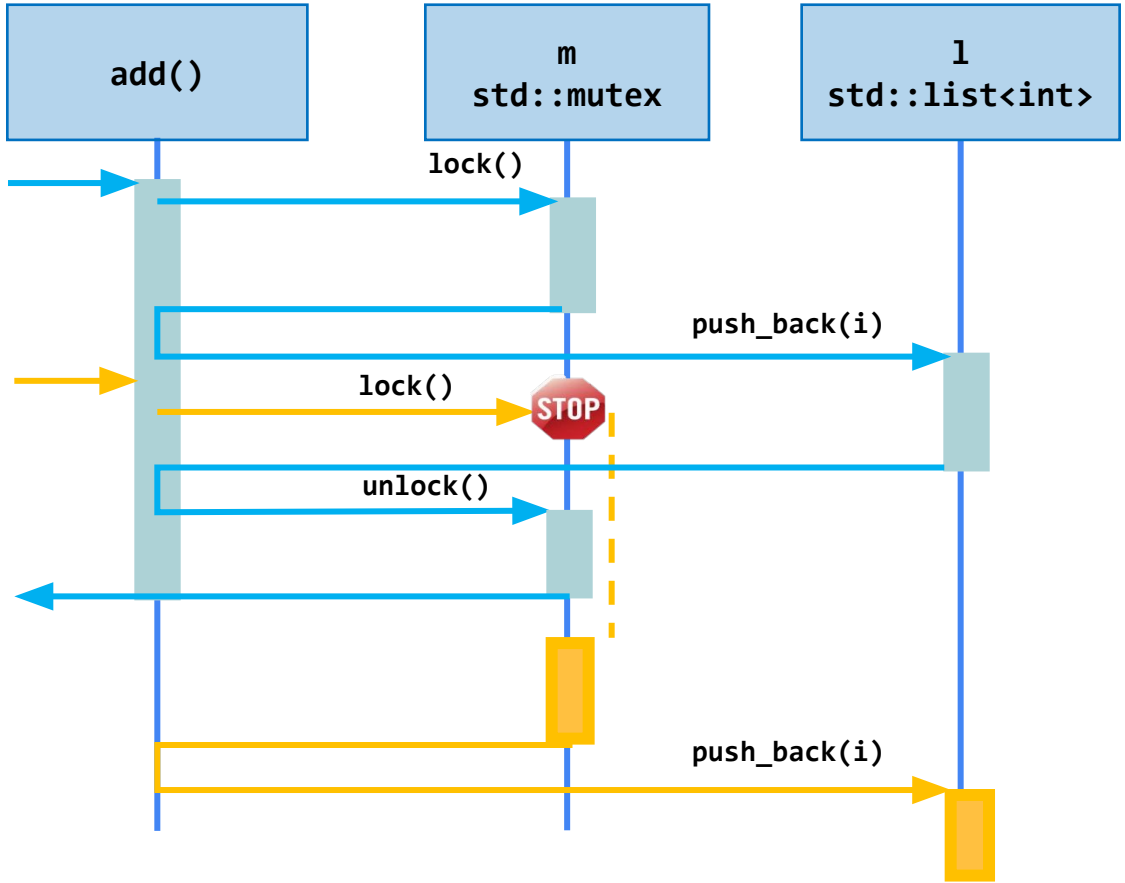
```
std::list<int> l;
std::mutex m;

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

C++

```
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!
};
```

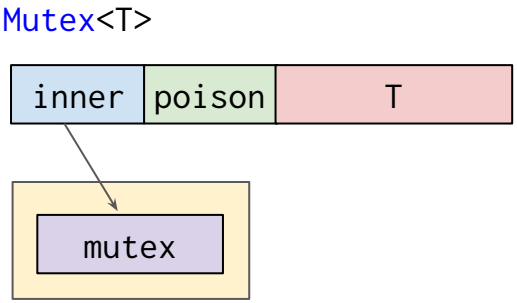
Valutazione

- L'uso del pattern RAII 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

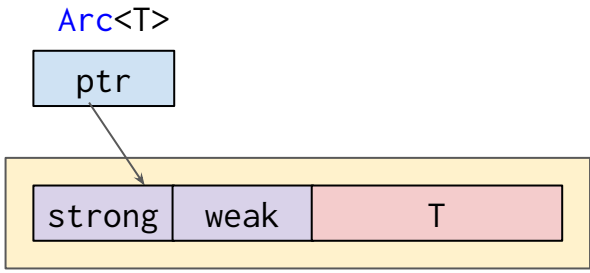
- L'accesso ad uno stato condiviso in Rust richiede l'utilizzo di **due blocchi** in cascata:
  - Il primo volto a permettere il possesso multiplo di una struttura dati in sola lettura da parte di più thread, realizzato mediante il costrutto `std::sync::Arc<T>`
  - Il secondo che consente l'acquisizione 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 RAII, ma - per come viene costruito - è necessariamente disponibile solo se si possiede il mutex

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

Condivisione dello stato

```
let shared_data = Arc::new(Mutex::new(Vec::new()));
let mut threads = vec![];
for (i in 1..10) {
  let mut data = shared_data.clone();    //duplicazione del possesso
  threads.push( thread::spawn( move || { //data è ceduto al thread
    let mut v = data.lock().unwrap();    //v è di tipo MutexGuard<T>
    v.push(i);                           //quando v esce dall scope, il lock
  }) );                                  //viene rilasciato
}
for t in threads { t.join().unwrap(); }  //v contiene i numeri da 1 a 9
```

Condivisione dello stato

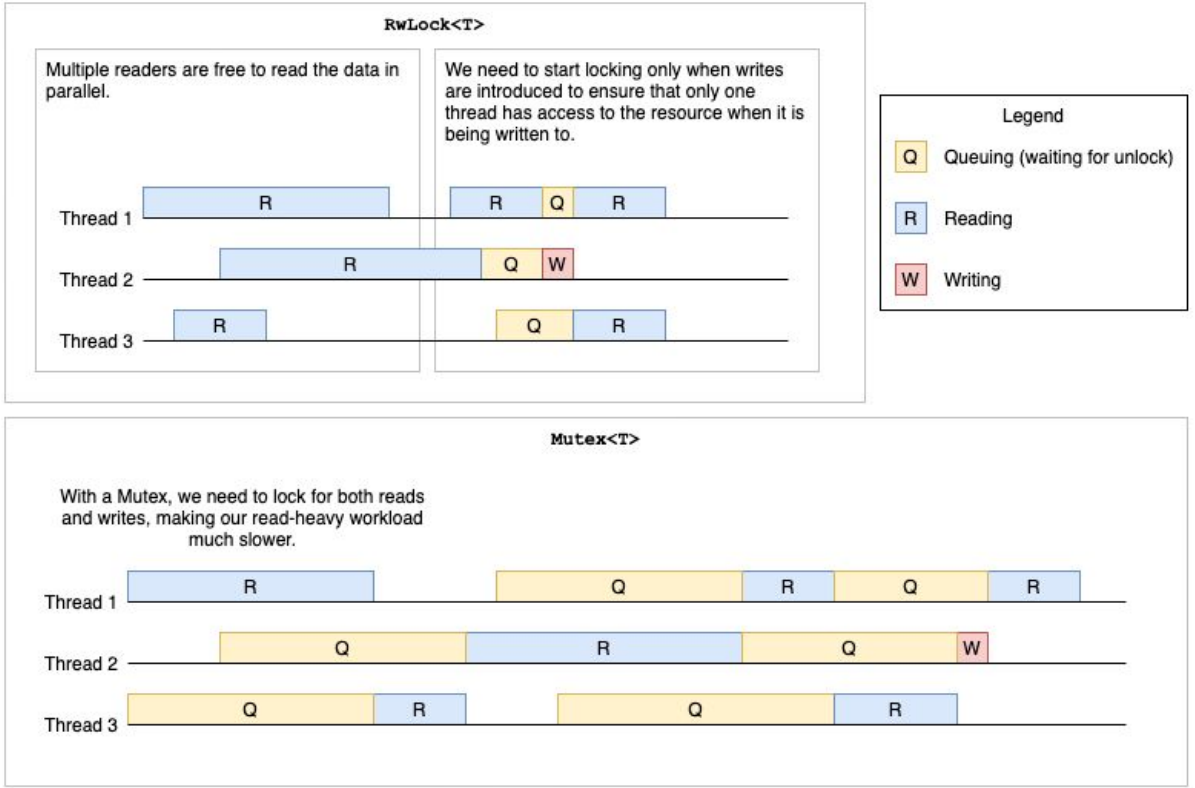
- Se un thread viene creato mediante la primitiva `std::thread::spawn(...)`, il compilatore non può fare assunzioni sulla sua durata
  - Di conseguenza, impedisce l'utilizzo di riferimenti condivisi tra la funzione lambda del thread e la funzione all'interno della quale il thread viene creato
- La libreria 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

Condivisione dello stato

```
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
assert_eq!(x, v.len());
```

std::sync::RwLock<T>

- Se gli accessi in lettura e in scrittura sono sbilanciati, può essere conveniente sostituire, alla **`struct Mutex<T>`**, la **`struct RwLock<T>`**
  - Questa 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>

```
use std::sync::{Arc, RwLock};
use std::thread;
let lock = Arc::new(RwLock::new(1));
let c_lock = Arc::clone(&lock);
let n = lock.read().unwrap();
assert_eq!(*n, 1);
thread::spawn(move || {
    let r = c_lock.read();
    assert!(r.is_ok());
}).join().unwrap();
```

- 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 in lettura è terminato senza averlo rilasciato o cercando di acquisirlo ulteriormente
- Entrambi gli oggetti di guardia implementano il pattern RAII
  - Rilasciando il lock nel momento in cui sono distrutti

Tipi atomici

- Il modulo std::sync::atomic mette a disposizione alcune strutture dati che costituiscono primitive di comunicazione tra thread basate sul principio della memoria condivisa
  - Esso offre versioni atomiche di valori booleani, numeri interi con e senza segno e puntatori nativi
  - Ciascun tipo è associato ad operazioni che, se usate correttamente, permettono di sincronizzare gli aggiornamenti di tali valori tra thread differenti
- Accanto alle operazioni di lettura e scrittura con associata barriera di memoria, questi tipi offrono funzionalità di tipo Read-Modify-Write
  - swap(...), compare_exchange(...), fetch_add(...), fetch_update(...)
  - Ciascuna di queste operazioni riceve come parametro esplicito il tipo di barriera di memoria da applicare per la fase di lettura e per quella di scrittura

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

Tipi atomici

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

fn main() {
    let spinlock = Arc::new(AtomicUsize::new(1));

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

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

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()`**

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, riacqiusisce 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
  ```

Condvar

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

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

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

Meccanismo di funzionamento

- 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 si 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)

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
  ```

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)
```

Attesa temporizzata

- Altri metodi permettono di limitare il tempo massimo di attesa, permettendo al thread di risvegliarsi anche in assenza del verificarsi della condizione
- ```
  pub fn wait_timeout<'a, T>(
      &self,
      guard: MutexGuard<'a, T>,
      dur: Duration
  ) -> LockResult<(MutexGuard<'a, T>, WaitTimeoutResult)>
  ```
  - Attende per un tempo massimo pari a `dur`
- ```
  pub fn wait_timeout_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

Condivisione di messaggi

- In alternativa alla condivisione dello stato, Rust offre un meccanismo di comunicazione e sincronizzazione tra thread basato sulla condivisione di messaggi
  - La funzione `std::sync::mpsc::channel<T>()` restituisce una coppia ordinata formata da una `struct Sender<T>` ed una `struct Receiver<T>`
  - Tutti i dati inviati tramite il metodo `send(...)` 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 fornita 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`

Condivisione di messaggi

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

let (tx, rx) = channel();

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

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

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

Canali sincroni

- La funzione **`std::sync::mpsc::sync_channel<T>(bound: usize)`** restituisce 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;

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

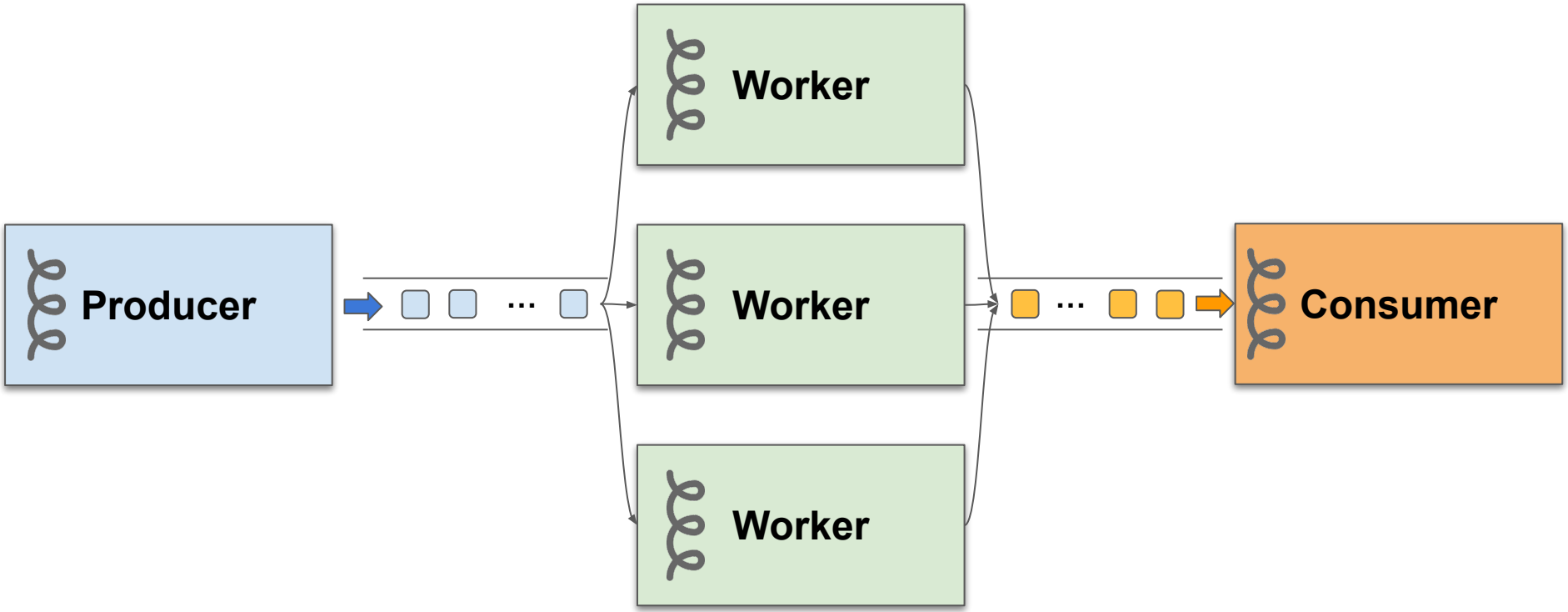
La libreria Crossbeam

- 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 mutabiltà 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 `rossbeam::channel::{after(...), tick(...)}` creano il solo estremo di ricezione che consegnerà un messaggio dopo il tempo indicato o periodicamente

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

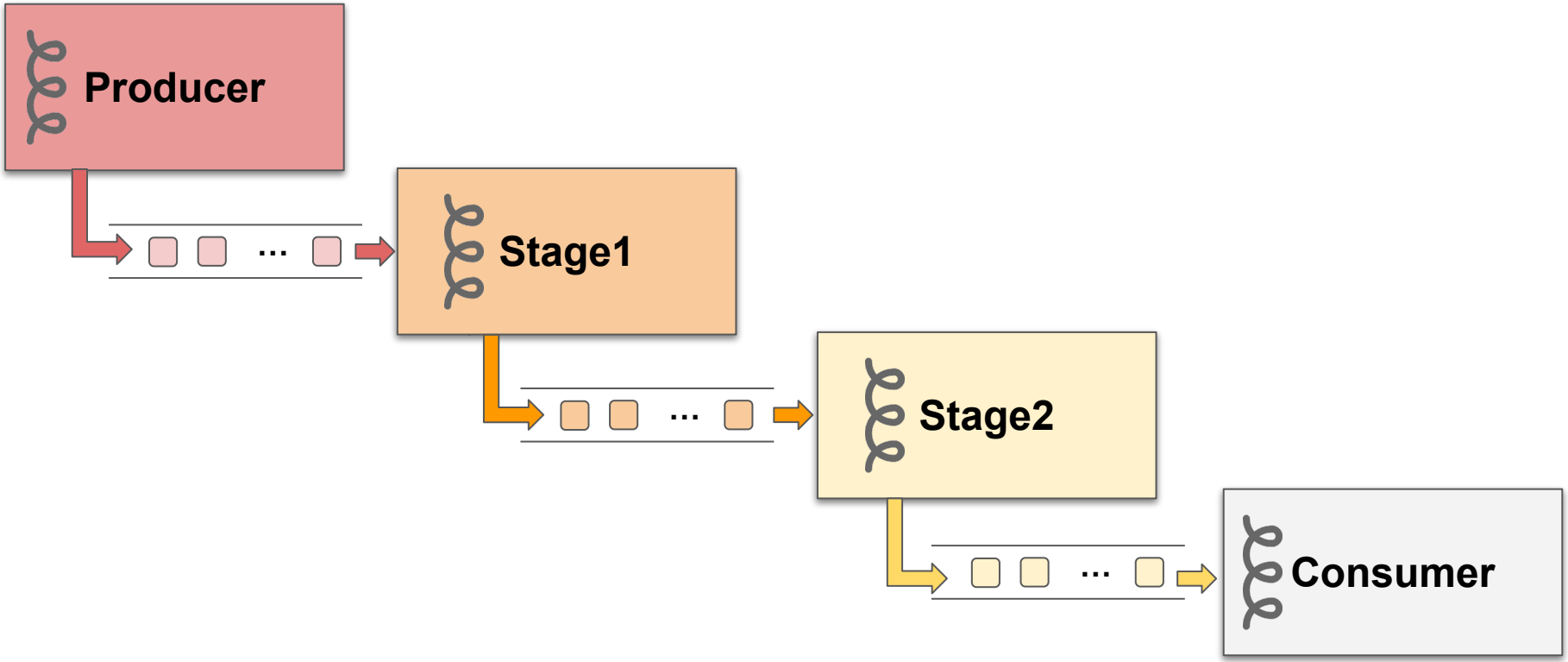
Fan-Out / Fan-In



Fan-Out / Fan-In

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

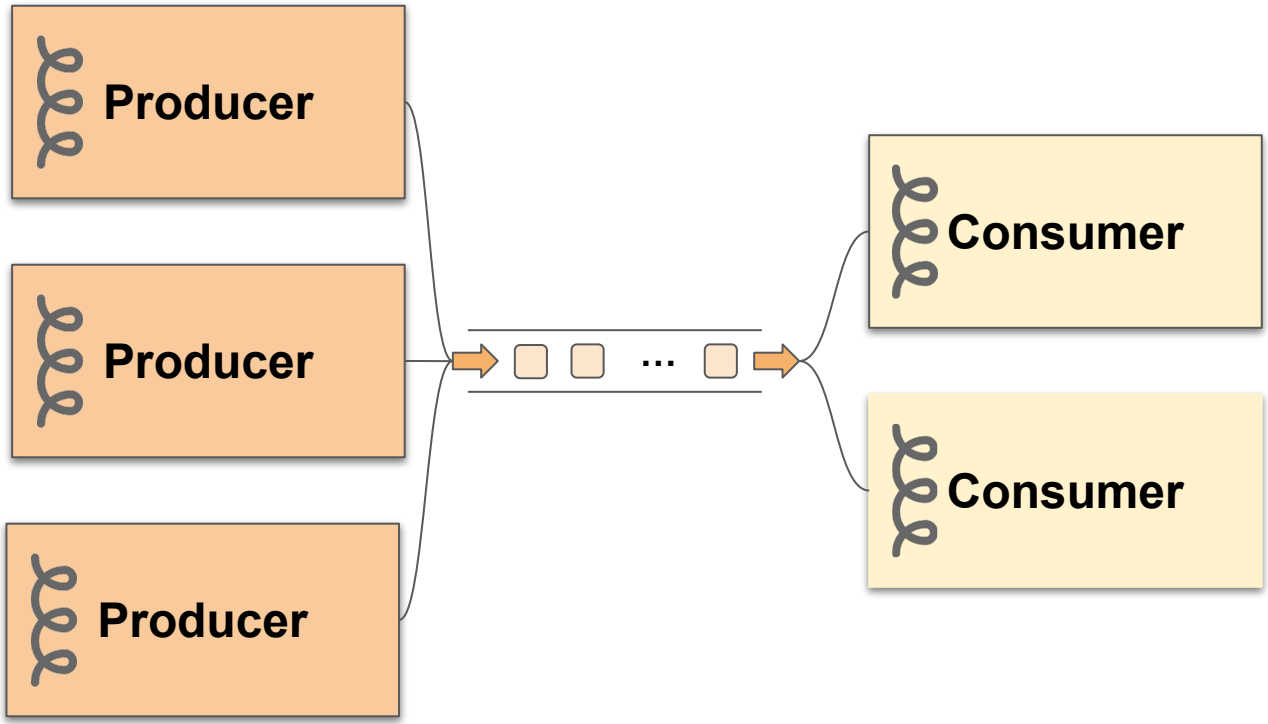
Pipeline



Pipeline

```
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



Producer/consumer

```
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(); }
}
```

Il modello degli attori

- Introdotto inizialmente in Erlang, è un modello concettuale che implementa la concorrenza a livello di tipo usando entità dette attori
  - Ideato da Carl Eddie Hewitt nel 1973
  - Toglie il bisogno di lock e sincronizzazione fornendo un modo più pulito e lineare di introdurre il concetto di concorrenza in un sistema
- L'attore è la primitiva principale
  - Contiene una mailbox alla quale possono essere inviati in modo asincrono messaggi
- Un messaggio incapsula una richiesta che può essere inviata ad un attore
  - I messaggi vengono depositati nella mailbox dell'attore destinatario e sono normalmente elaborati in modalità FIFO
- La libreria actix offre un'implementazione di questo modello basata sul framework asincrono Tokio
  - https://github.com/actix/actix

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/semaphores/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