07-Concorrenza:

Un programma concorrente dispone di due o più flussi di esecuzione contemporanei che possono essere eseguiti in parallelo nel caso in cui ci sono più core, altrimenti si alternano.

Quando chiediamo di creare un thread, il sistema operativo riserva una zona di memoria per contenere lo stack del thread. Per poter creare un thread, bisogna specificare una funzione che ne rappresenta la **computazione**:

- Thread esegue questa funzione
- Quando la funzione ritorna, il thread viene chiuso e cessa di esistere
    - In base al sistema operativo, questa funzione può tornare o meno un valore
        - Rust torna il valore che viene conservato finchè qualcuno non va a leggere.

Il SO si occupa di allocare le risorse fisiche necessarie e lo scheduler mette a disposizione la CPU, ripartendo i core disponibili.
Alcuni linguaggi offrono supporto a **thread nativi**

- gestione dei thread demandata direttamente al S.O

In altri casi, oltre ai thread nativi vengono offerti **i green thread**

- gestiti da librerie a livello utente, con il supporto parziale del sistema operativo
    - In java Fattibile perché in runtime è presente un proprio scheduler che agisce on topo di quello del sistema operativo

Nota: C++ e Rust offrono thread nativi, volendo con delle librerie ulteriori si possono ottenere green thread.

THREAD NATIVI

Tipicamente ciascun sisema operativo, offre 3 funzioni prinicpali:

- **Creo nuovo thread**
    - indicando la funzione che rappresenta la computazione che deve essere svolta e la dimensione dello stack richiesto
- **Identifico** in modo **univoco** al livello di sistema TID
    - Oltre al nome, posso ottenere l'inventario dei thread
- **Attesa di altro thread**: Chiedere di bloccarci, finchè un certo thread è finito
    - Eventualmente, posso avere accesso al valore finale oppure no (del thread che stavo aspettando)

Nota: i s.o. non danno la possibilità di cancellare un thread in quanto la **cancellazione** è possibile solo in un processo **cooperativo**: chiedo al thread di fermarsi e quando lui ha voglia, si ferma.
(la cosa che si può fare e uccidere il processo ma questo uccide tutti i thread)

Benefici concorrenza thread:

- Fare cose contemporaneamente
    - Mentre un thread è bloccato, posso fare altro
- Riduzione del sovraccarico dovuto alla comunicazione tra processi
    - Preferibile thread rispetto a processi in parallelo per la difficoltà nella comunicazione tra processi.
- Nel caso di cPU multicore:
    - Posso fare operazioni in parallelo, i diversi core eseguono diverse funzioni

Svantaggi concorrenza thread:

- Aumenta complessità del programma
    - Saltano le relazioni temporali tra quello che avviene in thread diversi perché i diversi thread, vanno per fatti loro.
        - Nuove fonti di erorre
        - Nuove tipologie di errore
- Memoria non può più essere pensata come deposito statico:
    - Se ho scritto 32 in una variabile, non è detto che tutti quelli che vanno a legggere leggano 32 perché in realtà io non scrivo veramente nella variabile
    - Se mentre io scrivo, tu leggi, cosa succede?

Dunque, concorrenza:

Se in un processo, sono presenti 2+ thread, per conto loro fanno quello che devono fare, bisogna però far interagire → non posso sapere se il thread2 è al mio passo o meno. Per orientarsi**, bisogna parlarsi**:

- Programma deve aver qualcosa che mi permetta di sapere che esistono gli altri e dove sono, cosa stanno facendo.

Dunque, oltre a definire le attività che ciascun thread deve fare per sé, vanno aggiunte altre **attività utili a coordinarsi**.

Meccanismi di comunicazione, interagiscono con l'architettura interna e forniscono una serie di cose utili per il multi-thread:

- Il processore esegue semplicemente il fetch-decode-execute
- Questo meccanismo continua finchè non si scatena un **interrupt**
  - Interviene s.o.
  - **Congela** i valori (in modo tale che il thread potrà ripartire come era)
  - Sceglie un **altro thread** per farlo cominciare

Per far comunicare i thread, servirebbe un pezzo di memoria dove uno scrive e l'altro legge (fattibile perché i thread operano in uno stesso spazio di indirizzamento), complesso.

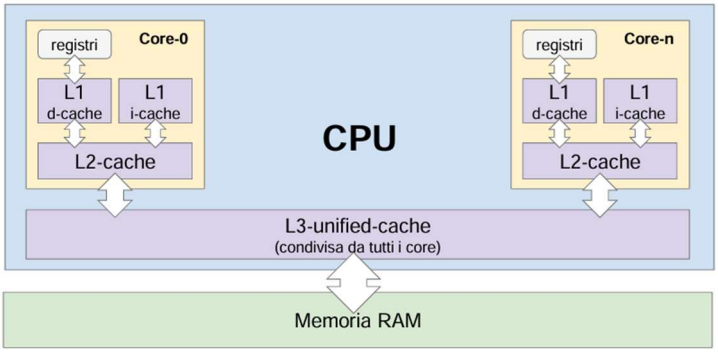
Modello di memoria

Quando un thread legge in memoria, può trovare:

- Valore iniziale del file eseguibile mappato su memoria
- Valore inserito dal thread stesso, l'ultima volta
- Valore inserito da un altro thread
  - Problematico a causa di cache e riordianamento istruzioni

***NOTA**: le cpu non vanno a leggere e scrivere direttamente dalla ram in quanto la RAM è troppo lenta per la CPU:*

- Per questo motivo la **cpu ha diversi livelli di cache:**
  - (Registri)
  - Cache l1 (divisa in dati e istruzioni)
    --in realtà anche tlb--
  - Cache l2
  - Cache l3 unificata
- *Io scrivo nella L1d del core0, non so quando arriva alla RAM o quando arriva al L1d del core n.*



Ci sono dei metodi in ARM per andare a prendere/scrivere direttamente su RAM

La grande differenza tra le varie cache è la differenza nei 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

Problemi aperti:

- **Atomicità**
- **Visibilità**
  - Se la mia modifica si è fermata nella mia cache, l'altro non vede che io ho scritto
- **Ordinamento**
  - Prima scrivo in un punto e poi in un altro
    - Mi aspetto che tu legga in quell'ordine e che trovi tutto; in realtà potresti trovare il secondo che ho scritto ma non il primo a causa di possibili tempi di propagazione diversi.

Per mettermi al riparo dai dati farlocchi, posso provare a pulire la cache o forzare la cache → presente in funzioni di bassissimo livello di libreria. → ho comportamenti affidabili che mi danno garanzie che il programma fa ciò che deve fare.

Nota: ARM mi permette di distinguere in modo più fine se:

- Sto facendo **lettura** → svuotare la mia cache
- Sto facedo **scrittura** → scrivere e forzare il flushing → trasferimento cache in memoria princiaple
  Sto facendo **scambio** → svuotare prima e forzare scrittura dopo

Se non si includono meccanismi di sincronizzazione, il programma va a caso:

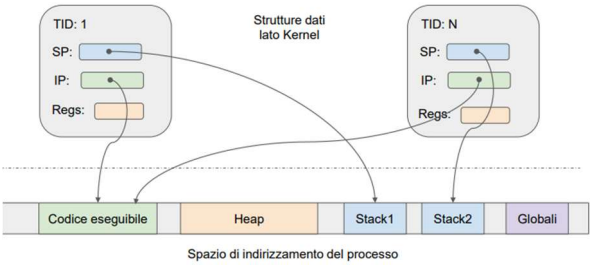
- Malfunzionamenti casuali e irriproducibili → non riesci neanche a fixarli

Internamente ogni thread è mappato nel kernel con un oggetto che ne ricorda uno stato:

- Utile allo scheduler del kernel per poter congelare e ripristinarlo
- Viene salvato lo stato del core su cui quel thread è in esecuzione
    - Registri contenenti indirizzi che puntano nello spazio di indirizzamento del processo che è in comune con altri thread
        - T2 può cambiare delle informazioni
        - Quando T1 si risveglia, deve capire cosa è successo

Thread di uno stesso processo:

- **Condividono** eseguibile → sola lettura
- Condividono heap → facile socntrarsi
- Condividono variabili globali → facile scontrarsi
- **Hanno proprio stack**
    - *Sembra safe ma in realtà se inserisco l'indirizzo heap, diventano non safe.*



Esecuzione e non determinismo thread:

- L'esecuzione del singolo thread è sequenziale
- Se più thread sono in esecuzione, non è possibile fare assuzioni sulle relative velocità di avanzamento
- In alcuni casi serve che un thread raggiunga un certo stato per abilitare altro thread a procedere
    - **condition variable**
- In alcuni casi serve che i thread eseguando azioni su aree condivise
    - c'è bisogno della certezza che, mentre cambio un dato condiviso, nessuni deve usare quel dato in lettura/scrittura.

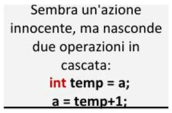
Mutex: oggetto che mi protegg l'area di memoria in modo tale che mentre scrivo/leggo , nessun'altro possa scrivere/leggere.

- Si accede alla zona protetta dal mutex con un'operazione di locking.
- Usato quando si deve procedere uno per volta.

Tutte le operazioni per concorrenza, sono riconducibili ad un uso misto di mutex o condition variable

Codice lez20_05_concorrenza

Nota su c++:
i++ → nella nostra testa è un'operazione atomica ma in realtà non lo è, è composta da più operazioni.



- per risolvere questo in c++, esistono dei tipi atomici es: std::atomic<int> e per sommare i.fecth_add(1)

INTERFERENZA

Si crea quando 2+ thread fanno accesso allo stesso dato modificandolo.

Se due thread cercano di accedere in lettura e scrittura contemporanemanete alla stessa variabile e qualcuno la cambia mentre qualcuno la scrive → **corsa critica:**

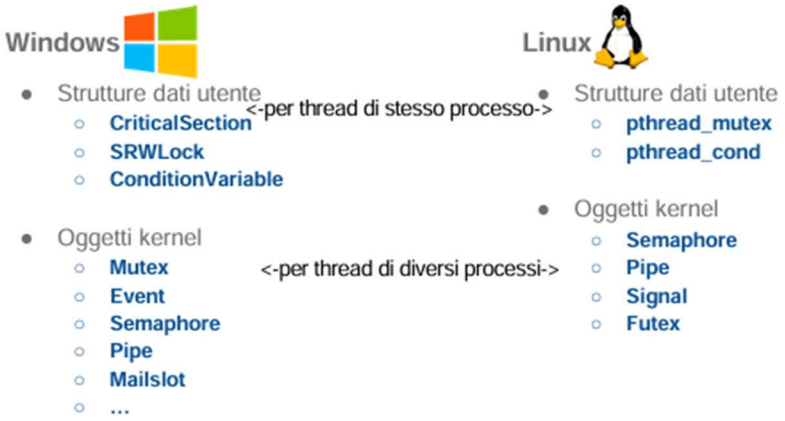
- una serie di condizioni che non sono controllabili dal programmatore interferiscono con l'esito del codice.

Se devo leggere qualcosa scritto da qualcun altro, devo prima svuotare la mia cache in modo tale da evitare di leggere il mio dato vecchio:

- devo leggere solo in momenti in cui sono sicuro che l'altro non sta scrivendo
- devo evitare di fare del polling → mando cpu al 100%, scarico batteria e non ottengo nulla

Processore + sistema operativo → gestiscono offrendono astrazioni per sincronizzazione.

Strutture a disposizione:



Borrow checker verifica solo che non si compino azioni illecite. Noi **dobbiamo lavorare in modo simile:**

- dobbiamo garantire che nessun thread possa modificare un valore mentre un altro lo sta leggendo/modificando
  - se faccio push su una lista, ci sono una serie di passi intermedi in cui la lista ha valore errato
- arricchire il nostro algoritmo che, nei punti in cui serve, costruisca uno scudo
  - es: metto il dato in una struct che rendo privata
    - aggiungo metodi che si occupono di prendere precauzioni necessarie per fare in modo che si prendano solo valori corretti e non si tocchino valori che non si possono toccare

Accesso condiviso, **problemi**:

- **atomicità**: se due thread fanno accesso alla stessa struttura, non sono in grado di capire chi fa cosa.
  - ++ non è atomico
  - Push non è atomico
- **Visibilità**: quando una variabile che ho modificato può essere osservata da un altro
  - Se non svuoto le cache, la modifica c'è certamente nella mia cache ma non è detto che il resto del sistema lo sappia
- **Ordinamento**: sotto quali condizioni, le operazioni saranno viste nello stesso ordine da altri.
  - Compilatore e cpu possono invertire l'ordine di alcune azioni

Esistono 3 Tipi di oggetti:

- Oggetti **Atomic**: oggetti che incapsulano dei dati **elementari**
  - Funzionano con
    - Interi
    - booleani
    - Puntatori → garantisco il puntatore e non il dato puntato
  - Offrono metodi non interrombili
- Oggetti **Mutex**: garantiscono accesso 1 per volta a strutture dati arbitrariamente complesse
  - Sottoforma di **smartpointer**
    - Il mutex racchiude il dato che processo
    - Si può accedere al dato solo con i metodi del mutex
- **Condition variable**: per quelle situazioni in cui abbiamo bisogno di attendere, senza consumare cicli di cpu, che accada qualcosa
  - Possono essere usate solo insieme ad un mutex

USO DEI THREAD ***35***

Si usa la libresria **std::thread** che offre la funzione **spawn** che permette di creare un nuovo thread:

- Spawn accetta una closure che indica cosa vogliamo che il thread faccia
  - Funzione che fa qualcosa
  - Attraverso **join** posso accedere al valore di ritorno della funzione
    - **Blocca il chiamante** finchè il thread non è terminato con un result
      - Result può essere Ok(T) o contenente un Errore→ thread ha generato panic

Nota: il thread che crea e il thread creato non hanno alcuna relazione di parentela.

SPAWN:

```
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) => { … },
}
```

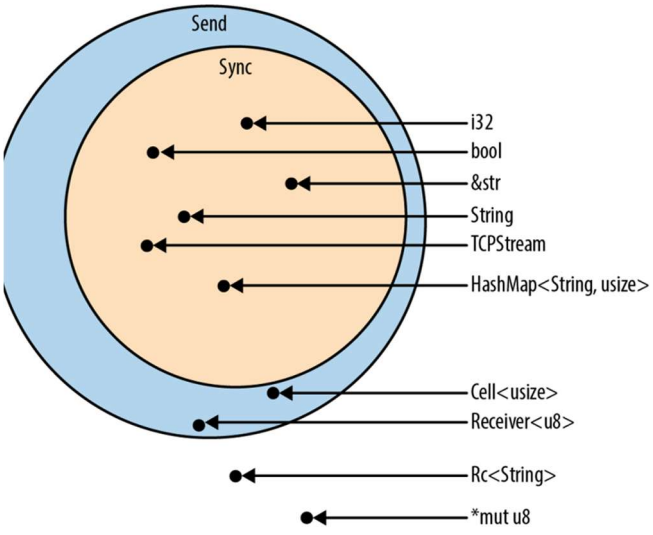
Move → se io prendo riferimento a qualcosa del mio scope, questo diventa proprietà completa.

Builder: permette di configurare il thread, specificando nome, dimensione stack

```
use std::thread;

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

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

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

Cosa posso passare ad un thread:

- Variabili devono rispettare alcuni vincoli
  - **Send**
    - È lecito che un certo dato sia passato ad un altro thread
    - Lo hanno tutti tranne i reference e i pointer nativi
      - Se io ti passo un reference, tu lo potresti usare da qui in avanti e non sono in grado di capire finchè dura. → non te lo passo
  - **Sync**
    - Condivisibili in sicurezza in thread differenti
      - Se qualcuno gode di sync → riferimento diventa passabile
    - Non implementato da:
      - Cell
      - RefCell



Nota: 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

```
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>`
```

Rc non può essere madnato da un thread all'altro in quanro non ha tratto send → usa Arc?

Modelli di concorrenza:

- A **stato condiviso:** struttura dati a cui accedono più thread
- A **scambio di messaggi:** prevede che di condiviso non ci sia nulla ma abbiamo dei canali in cui ci possiamo dire delle cose

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**
- Il secondo che consente **l'acquisizione in lettura/scrittura della struttura dati**, realizzato alternativamente mediante il costrutto std::sync::**Mutex**, con std::sync::**RwLock** oppure ricorrendo ai tipi atomici

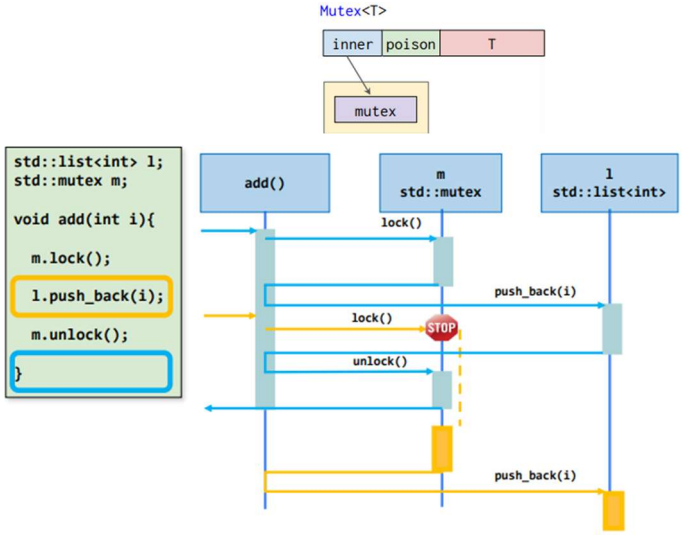
Questa combinazione che prende spunto dal pattern RAII, permette di rendere esplicito nella struttura del codice e nei pattern di accesso ai dati cosa sia condiviso e impedisce di fatto l'accesso senza il corretto possesso del lock relativo

- *Permettendo al compilatore di bloccare ogni tentativo di accesso non conforme*

Mutex<T>:

Mutex incapsula un dato di tipo T e rifetimento a mutex.
mutex **offre LOCK**:

- Prendo variabile m di tipo mutex
- Faccio m.**lock()**
  - Se c'è qualcuno che sta già facendo qualcosa → mi blocca
  - Altrimenti mi fa andare offrendomi un oggetto smartPointer attraverso il quale posso accedere al mio dato
- Mutex fornisce una mutabilità interna, condivisibile tra thread sotto certe condizioni



Lock restituisce uno smart pointer (LockResult<MutexGuard>)che, deferenziato, ci mostra il dato:

- La variabile mutex non è mutabile
- Quando chiamo lock, puntatore che prendo posso renderlo mutabile in quanto lock mi garantisce che il dato lo può avere max un utente per volta.
- Quando smartPointer richiesto, viene distrutto/esce da scope, sarà unlockato → permettendo ad un altro thread di prenderne il possesso

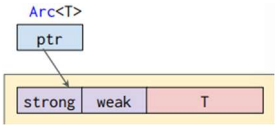
Note aggiuntive:

- Quando si invoca lock → interior mutability
  - Se lo chiedono in due, il primo entra e il secondo aspetta

Potrebbe succedere che mentre un thread possiede il lock, panica

- Panic comporta contrazione stack → rilascio smart pointer legato al mutex
  - Il rilascio con panic, avviene mentre ho il mutex ma non so cosa ho fatto su quel mutex fino a quel momento → rischio che ci sia in vita un valore parzialmente modificato
    - Per evitare ciò, mutex viene rilasciato MA viene settato il fleg POISON che mi indica che il dato è incerto in quanto ha panicato chi lo usava
    - Chi applica lock, ottiene un result di tipo error:
      - Io te lo do, ma potrebbe essere errato; a questo punto chi chiede lock può decidere:
        - Accettare comunque e togliere poison
        - Rinunciare al mutex

ARC<T>:



Blocco più esterno: Arc → per condividere in lettura il mutex.
Permette di condividere il possesso di un dato, allocnandolo 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

```
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
```

→ duplicato di Arc

→ grazie al **move**, prende possesso di data; invoca **lock()** deferenziandolo (se c'è qualcuno che sta usando, apsetta); .**unwrap**: smartpointer che deferenziato mi da accesso al vettore che ospita il mutex.

→ aspetto che finiscano, colleziono ma l'ordine non è garantito perché dipende da come hanno preso possesso del mutex.

Nota: essendo .push, fa la deref da solo, altrimenti avrei dovuto deferenziare con asterisco

Se un thread viene creato mediante la **primitiva std::thread::spawn** il compilatore non può fare assunzioni sulla sua durata dunque impedisce l'utilizzo di riferimenti condivisi tra la funzione lambda del thread e la funzione all'interno della quale il thread viene creato

Differentemente, **thread_scope** accetta una lambda che viene invocata internamente, direttamente da scope passandogli un parametro scope, a partire dal quale crea thread:

- I thread creati con parametro scope, sono aspettati dalla funzione dello scope.

Nota: se il thread che prende a prestito, usa una variabile esterna allo scope, la variabile dura sicuramente almeno quanto lo scope.

```
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());
```

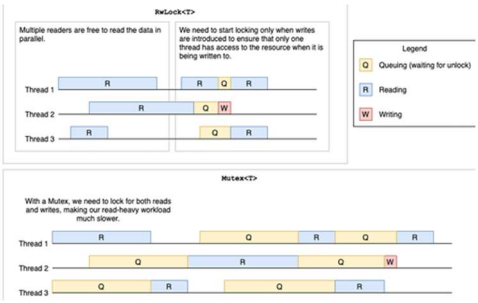
→

→preparo scope
→ oggetto s fornisce oggeto spawn → creo un thread (so che questa lambda dura meno dello scope)

→ altro thread che osserva v → posso farlo poerchè to facendo operazione di lettura → ref semplice
→ assegno ad x la somma di v[0] e v[2]; x viene preso come ref mut (non c'è move prima spawn)
→ se ci fosse un altro thread che voleva usare x → borrow checler stroncava

➔ Quando entrambi i thread sono finiti, thread termina

RWLock:



ci sono tanti che vogliono leggere e pochi che vogliono modificare:

- Oltre alla struttura mutex, posso usare RwLock: simile al mutex ma offre 2 metodi che restituiscono smartPointer con cui accedere al dato:
  - **Read** : tanti possono leggere insieme
  - **Write**: uno solo può scrivere

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

use std::thread;

let lock = Arc::new(RwLock::new(1));

let c_lock = Arc::clone(&lock);

let n = lock.read().unwrap();

assert_eq!(*n, 1);

thread::spawn(move || {

    let r = c_lock.read();

    assert!(r.is_ok());

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

TIPI ATOMICI: comodo per gestire dati semplici:

- Booleani, caratteri, interi
- Sostituisce attività semplici con attività più complesse che mi permettono di avere atomicità

Gli oggetti atomici hanno oggetti di tipo ref cell (anche se sono mut)

```
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();
```

→ atomic inzializzato ad 1
→ metto in un arc per garantire accesso condiviso
→ passo ad un thread con la clausola move
→ thread scrive all'interno qualcosa di diverso

LE operazioni sugli atomic in rust, richiedono che sis specifichi l'operazione che si sta facendo:

- **Scrivendo** → ordering release : fai un flush della cache, garantendomi che il dato chet u hai scritto nella cache, atterra in mmoeira pcentrale
- **Lettura** → order acquire → prma di fare questa lettura, invalida la cache così prendi il dato da memoria centrale
- **Lettura+scrittura (es ++)** → ordering sequencial → svuota la cache, effettura ooperazione rpendendo dalla ram il dato vero e successivamente fai il flush della cache
  - Permette di dosare il peggioramento delle operazioni del sistema, di **conseguenza** chiedo quando voglio avere penalizzazioni

Weak:

Struct std::sync::Weak<T> permette di realizzare dipendenze circolari con riferimenti che non partecipano al conteggio, garantendo così la possibilità di rilascio.

Per fare accesso al dato puntato, occorre invocare il metodo upgrade(), che restituisce un valore di tipo Option<Arc<T>> Si crea un oggetto di tipo Weak<T> a partire da un riferimento di tipo Arc<T> invocando su quest'ultimo il metodo downgrade()

Nota: ci sono delle operazioni che voglio essere l'unico a fare ma sono condizionate ad altro (voglio aspettare che un altro thread finisca di fare una cosa) → **Attese Condizionate**:

- Stato booleano (non ottimo)
    - Posso vederlo facendo polling
        - 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
- **Condition variable**, offre:
    - Metodo **wait**: aspetta
    - Metodo **notify**: se stavi aspettando, parti
        - Notify one→ notifica uno
        - Notify all → notifica tutti

CONDITION VARIABLE: da usare in coppia con un mutex

- Chi chiama wait, chiede al sistema operativo di essere rimosto dallo scheduler
    - Sistema operativo lo mette come not runnable con tag della condition variable
- Quando qualcuno chiama notify
    - Sistema operativo va nella coda nei not runnable e cerca se c'era qualcuno taggato come in attesa di questta condition variable
    - Inserisce in coda

Nota: se notifyOne → si ferma al primo (non so quale) ; se notifyAll → mette tutti nei runnable

- Se non c'è nessun dormiente, non succede nulla e operazione scartata
- Se inserisco più di uno nella runnable, quando si libera cpu, partono
- **Condition variable deve essere accompagnata da un mutex** → quando mi risveglio, la prima cosa che viene fatta dal sistema operativo e riacquistare il mutex
    - Mentre io dormo, il mutex lo possono usare gli altri
    - Quando mi risveglio, per prima cosa riprendo il mutex che avevo prima

Nota: se ci sono 10 thread, questi avranno in comune la condition variable e il mutex.

- `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, riacquisisce 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

→ necessita di un mutexGuard (smart pointer che otteggo quando ho un mutex), mi sospende finchè qualcuno non chiamerà una notify; A quel punto me lo restituisce dentro un result OK(lockguard)

→ prova a riacquisire il mutex → me lo restituisce dentro un result

```
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();
}
```

→ struttura con mutex(dato condiviso) e condvar

→ comunica che ha terminato, facendo diventare false in true e sveglia se c'è uno che aspetta .

Thread principale:

→ finchè non è ancora partito, mi addormento.

→ a wait, passo il lock eseguendo una move, il lock viene buttato via dalla wait, in modo tale che altri possano lavorarci (se mi addormentassi con il lock, nessuno potrebbe usarlo) ; Quando qualcuno mi sveglia, vengo messo in runnable; Quando vengo pescato dalla coda:

- Riacquisisco vecchi mutex, acquisito nel result (wait.started.unwrap)

notifiche spurie

Esistono le possibilità che si viene rischedulati anche se nessuno ha chiamato una notify.
Tutti i linguaggi, sono soggetti al problem delle notifiche spurie: potrebbe capitare che vieni svegliato anche se non è ora → per questo motivo meglio usare un while di un if.

- 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

- Thread svegliati ma non ce ne bisogno:
- se quelli che aspettano sono piu' di quelli effettivamente dati, chi non li ottiene, torna a dormire

Il sistema operativo contiene una lista di not Runnable:

- Not runnable hanno un tag che indica la ragione per cui quel thread è non runnable.
- Quando si sveglia, si cercano i not runnable con il tag della sveglia

NOTA: non possono uscire tutti insieme da wait in quanto devo prendere prima il mutex.

- Esco dal mutex, guardo lo stato (while), decido se ci sono le condizioni per andare avanti oppure tornare a dormire
- Wait_while chiama lambda torna true se bisogna tornare a dormire, False se devo uscire da waitwhile
  - Riceve come parametro il contenuto del mutex
    - Posso leggerlo e modificarlo

NOTA: La relazione tra l'evento che sì e notificato e la notifica che arriva, è nella testa del programmatoere:

- Notifico questa condition variable quando si verifica questa cosa

NOTA: è possibile che qualcuno faccia partire un thread che provoca una notify mentre non c'è ancora qualcuno pronto a ricevere

- S.o. scorre la lista dei not runnable, non vede nessuno con quel tag e scarta la notifica
- Potrebbe succedere che ci sarebbe dovuto essere qualcosa ma il mio programma è stato scritto male e lui non è ancora tra i not runnable
  - Perciò meglio che prima di addormentarmi verifico se per caso si sia già verificata
  - La cosa che sto aspettando, devo occuparmi di reificarla all'interno dello stato
    - Devo avere un modo semplice per capire se è già avventua o meno → scrivo in modo esplicito sullo stato
    - Uso lo stato (quello che sta dentro il mutex) per rendere esplicite le cose che stanno capitando
    - Uso le condizioni, i cicli while o waithwhile per guardare nello stato
      - C'è condizione → eseguo in base a condizione

Attesa temporizzata:

- ```
  pub fn wait_timeout<'a, T>(
      &self,
      guard: MutexGuard<'a, T>,
      dur: Duration
  ) -> LockResult<(MutexGuard<'a, T>, WaitTimeoutResult)>
  ```

→attendi massimo dur

Attende per un tempo massimo pari a dur; eventuali modifiche ricevute, portano a ri-addormentarsi se ← la funzione condition restituisce false

```
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
```

Condivisione di messaggi

78 – 3 giugno

Meccanismo di comunicazione e sincronizzazione tra thread basato sulla condivisione di messaggi.

Ci sono lunghi intervalli di tempo (tempo CPU) in cui non stiamo facendo nulla.

Nota: *la chiamata read è bloccante quindi se voglio fare due letture "in contemporanea" devo fare due thread… In realtà il thread1 è occupato per poco tempo (spiega al s.o. cosa vuole) e poi va in attesa. Stessa cosa per il thread2 ma in un istante di tempo differente.*
Dunque, in realtà, questi due thread non stanno lavorando in parallelo.
Volendo, si potrebbero portare avanti più letture in parallelo su uno stesso thread se avessi un modo per fare richiesta e passare a fare un'altra richiesta sullo stesso threead → mi servirebbe un'interfaccia non bloccante.
→ Prima, l'unico modo per farlo era eseguire più thread.

Per passare a programmazione asincrona, devo gestire bene lo stato della mia applicazione.

--esempio dei cuochi per un pranzo (primo, secondo, etc)

I diversi attori (thread) devono comunicare, comunicazione avviene in due modi:

- Variabile struct che mantiene lo stato dei vari thread (dato che i thread lavorano tutti nello stesso spazio di indirizzamento)
    - Vari thread aggiornano questa struttura condivisa (uno per volta)
        - Per garantire uno alla volta:
            - **Mutex**: monade che nasconde il dato a chi vuole usarlo (se uno ha un accesso, non lo ha nessun'altro)
                - Per accedere al dato devo invocare sull'oggetto mutex, il metodo lock
                    - Se libero → torno fornendoti uno smart pointer per accedere al dato e puoi leggere&scrivere
                    - Se dato in uso → aspetti
                - Quando ho finito, butto via lo smart-pointer che potrà essere dato a qualcun altro
        - Ci sono delle attività che richiedono di iniziare a fare qualcosa solo quando qualcun altro ha fatto qualcos'altro:
            - **Polling**: continuo a chiedere in maniera in efficiente
                - Uso elevato cpu per fare nulla
                    - Batteria scarica
            - **Condvar**: fornisce un modo efficiente per aspettare
                - **Wait**: mi fa aspettare finchè qualcuno non chiama notify

Usando mutex+condvar posso gestire il lavoro dei più thread:

- Posso modificare solo il mio pezzo
- Posso leggere gli altri
- Mi appoggio alla condvar se devo aspettare
- Condvar necessita di un mutex

Tupla **mutex+convar → se ho bisogno di condividere questa tupla, la metto dentro ad un Arc** così i vari thread possono avere un clone (quando l'ultimo thread avrà finito, la tupla verrà completamente rilasciata)

➔ MODO DI BASSO LIVELLO che da visibilità a tutti di tutto

In questo modo c'è il rischio che un thread modifichi le cose di altri perché, quando possiede il mutex, può modificare quel che vuole. Inoltre, mentre un thread possiede questa struttura, gli altri non la posseggono. Rust ci offre anche Rwlock che offre due metodi (accesso in scrittura e accesso in lettura che restituiscono diverso smart-pointer differente in base alla richiesta di accesso – si possono avere tanti lock in lettura ma solo uno in scrittura nel caso in cui non ci sia nessuno che legge).

Ogni **comunicazione implica una sincronizzazione** → rust mette a disposizione il concetto di **canale**: struttura dati condivisa tra mittenti e destinatari che permette di madnare dei messaggi:

- Canale è **struttura generica <T>** dove T deve godere del tratto send.
    - Qualcuno deposita nel canale e qualcun altro lo recupera
    - Canale è strettamente **sequenziale**, chi legge, legge nell'ordine in cui sono stati scritti
    - Canale disaccoppia tempo del mittente da quello del destinaario
        - Il messaggio resta nel canale finchè qualcuno non lo legge, quando viene letto, viene tirato via nel canale
            - Se nel frattempo chi scrive, aggiunge altre cose, il canale le aggiunge
            - Un canale può contenere tanti dati tanti quanti ne può mantenere la ram
    - Pensati in modo tale che un thread scriva e un altro legga
- Quando creo un oggetto di tipo canale ne vengono creati 2:
    - **Sender**: mandare
        - Offre metodo **send**
            - Ritorna sempre un result:
                - error → è possibile che il thread responsabile di leggere sia già andato via
                    - Se so che il destinario non c'è più, è inutile che scriva
        - Oggetto sender è clonabile
            - Usa politica **mpsc** → multiple procucer-single consumer
                - Ho più pezzi che possono scrivere e che possono essere assegnati a thread distinti
                - Se due thread provano a scrivere contemporaneamente, uno entra e l'altro aspetta e passa per secondo
    - **Receiver**: ricevere
        - Offre metodo **recv**
            - Ritorna un result:
                - Se c'è qualcosa nel canale → prende il messaggio più vecchio e me lo restituisce → ok
                - Se non c'è niente del canale, aspetto finchè qualcuno non mette qualcosa o finchè il sender non viene distrutto
                - Se sender distrutto →rec error
                    - Nel caso in cui ho più sender, torna error quando il conteggio dei sender è a 0
        - Receiver non è clonabile
- *Quando avviene il drop del receiver, tentativi di send non saranno accettati*
- *Canale usato come strumento di sincronizzazione tra 2 thread (senza dar fastidio ad altri thread)*

Meccanismo che viene adottato è quello di **cessione della proprietà:**

- Sender prende possesso del dato T che volevo mandare e lo cede al canale
- Possesso del canale finchè non arriva un ricevitore a leggere
- Quando ricevitore legge, thread che riceve prende possesso del dato

Questo metodo mi garantisce che **ricevitore legga solo dopo che mittente ha scritto**; in questo modo posso garantire che il thread che riceve, faccia delle azioni solo dopo che riceve quel dato.

Codice canaleThread

```
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);
}
```

Volendo **si potrebbe settare una dimensione massima al canale** → se il receiver è lento, chi manda per un po' ha buffer ma dopo un po' si ferma:

- Usando **sync_channel** con parametro usize che indica quanti messaggi possono stare nel channel al massimo
    - Se sender manda più messaggi di quando disponibile→ aspetta

```
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);
```

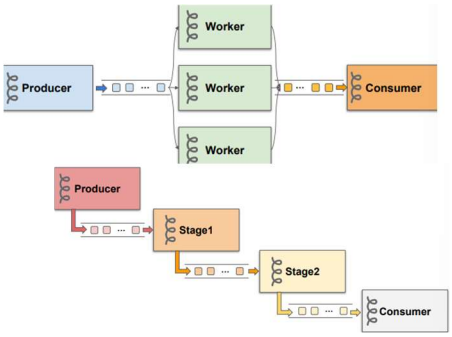
- `//se bound==0 -> posso depositare solo se il sender è già pronto a sentire`

I canali forniti da windows sono **mpsc**

CROSSBEAM

Ci sono altre librerie che forniscono funzionamenti diversi come CrossBeam
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
- Offre dei canali **mpmc**:
  - Ricevitore può essere clonato
  - Messaggio è sempre consumato una volta sola
    - Se ci sono due a leggerlo, il messaggio verrà letto a caso o dal lettore1 o dal lettore2 in base a chi arriva prima
  - Si usano in quelle situazioni in cui mi sta bene nons apere a priori chi consuma questo messaggio
- Mi abilita alcuni pattern di programmazione concorrente:
  - **Fan in/ fan out**:
    - Si usa in quelle situazionI in cui si ha una sorgente di dati che produce informazioini di qualche tipo
      - Thread genera sul canale una sequenza di valori (quadratini azzurri)
      - Valori hanno bisogno di subire una trasformazione (es da posizione gps, volgio passare a via e numero civico)
        - *Arricchire informazioni è un'operazione lenta*
          - *Faccio in modo che ci siano più worker in modo tale da provare a stare nel tempo voluto*
    - Producer: scrive in ordine i valori
    - Thread worker: lottano per strappare dal canale di ingresso un messaggio
      - Lo lavorano
      - Lo buttano in uscita
        - Non è detto che i messaggi escano dai diversi worker nello stesso ordine in cui sono stati inseriti dal producer
        - Consumer finale ha il compito di ordinarli.
    - Consumer: riceve i valori e deve ordinarli



  - **Pipeline**:
    - Ho una serie di fasi che devono essere applicate in sequenza
    - Fare attenzione perché se si pianta il 2, il producer non si accorge
      - Usare canali sync bounded
  - **Producer/Consumer**:
    - Aspettano di ricevere l'indirizzo della funzione da invocare sul canale.
    - Di base i thread Consumer sono fermi, quando l'algoritmo principale deve fare qualcosa, affida al thread pool la funzione da chiamare che sarà così mandata ai consumer