



Programmazione asincrona

Gestire attivamente le attese

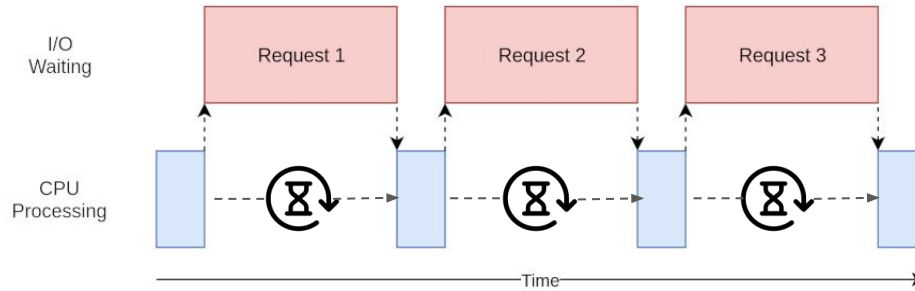
Programmi concorrenti

- Se la necessità di eseguire più operazioni, indipendenti tra loro, **in parallelo** porta alla creazione di più thread, altre esigenze di programmazione possono beneficiare di un approccio differente
 - L'uso dei thread trova piena giustificazione quando occorre eseguire algoritmi complessi, basati principalmente sull'uso intenso della CPU e si dispone di un hardware multicore
 - In queste situazioni, infatti, è possibile ridurre il tempo complessivo di elaborazione sfruttando il fatto che più computazioni procedono parallelamente
- I costi legati alla programmazione multithread sono costituiti, da un lato, dalla complessità legata ai meccanismi di sincronizzazione
 - E, dall'altro, dalla necessità di allocare preventivamente lo stack di esecuzione di ciascun thread
 - In caso di creazione di molti (1000+) thread, tale costo diventa significativo e, in certe situazioni, proibitivo

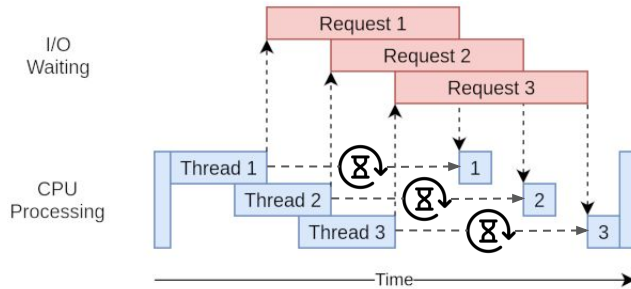
Operazioni bloccanti

- I casi in cui l'approccio multi-thread può non essere ottimale sono quelli in cui la computazione richieda di ricevere informazioni da un sottosistema separato
 - Come il file system, la rete, un timer, un altro programma...
- In queste situazioni l'esecuzione non può continuare e occorre attendere che il sottosistema in questione fornisca le informazioni attese
 - Normalmente, il sistema operativo rileva la situazione e sposta il thread corrente nello stato *"NotRunnable"*, sospendendone l'esecuzione fino a che non si verifica la condizione attesa
- Se il programma deve svolgere altri compiti, oltre a quello che ha generato l'attesa, si creano tre possibilità:
 - Gli altri compiti saranno eseguiti successivamente, dal thread corrente
 - Si creano più thread secondari per eseguirli, accollandosi la complessità legata alla loro sincronizzazione
 - Si organizza il codice in modo tale da separare la richiesta di eseguire l'operazione, dalle operazioni che dovranno essere fatte quando arriverà la risposta, così da non bloccare l'esecuzione del thread corrente, ammesso che sia altro da fare

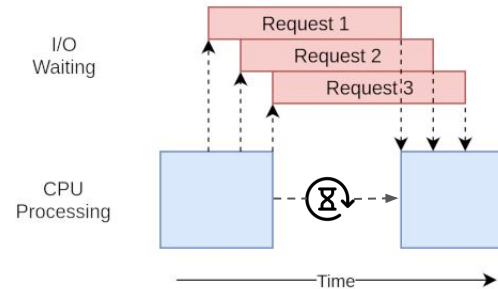
Strategie di esecuzione



(a) Single-threaded process



(b) Multi-threaded process
with GIL acquired by current thread



(c) Single-threaded process
with asyncio

Esecuzione asincrona

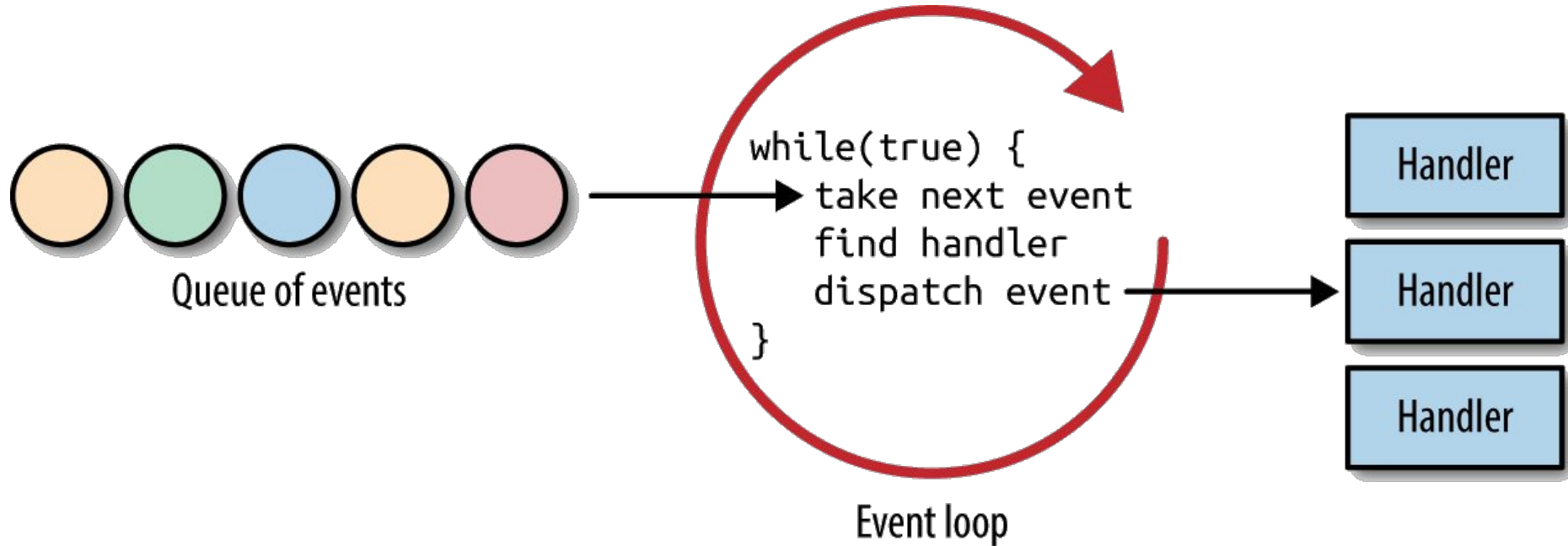
Si usano i **thread** quando occorre elaborare in parallelo, mentre si usa l'**esecuzione asincrona** quando occorre attendere in parallelo

L. Palmieri, Zero to production in Rust, 2022, ISBN 979-8847211437

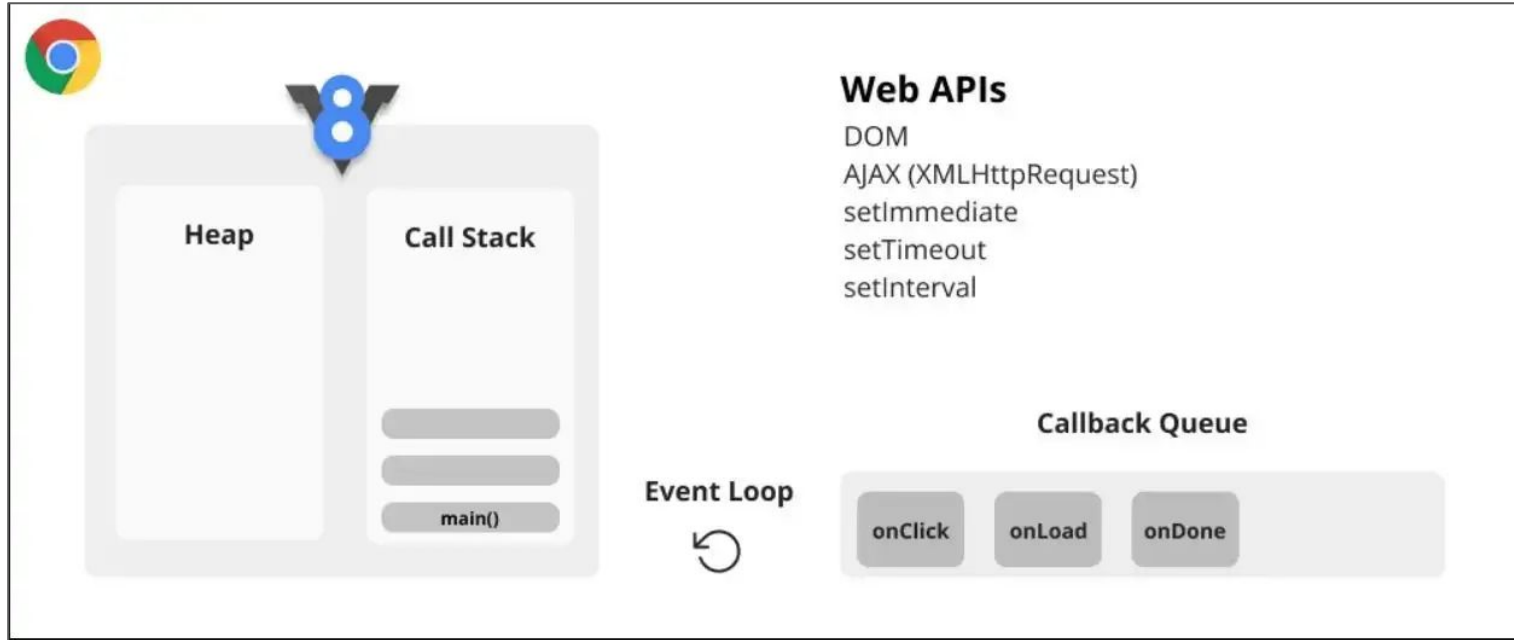
Esecuzione asincrona

- Il modo più diretto di implementare la terza strategia richiede che le azioni conseguenti ad una data operazione bloccante siano racchiuse in un'apposita funzione (callback)
 - Tale funzione riceve come parametro il risultato dell'operazione bloccante e sarà invocata quando il sottosistema che è stato interrogato avrà fornito la propria risposta
- Chi si occupa di fornire tale valore e in quale thread avverrà la chiamata?
 - Linguaggi diversi offrono soluzioni alternative
- In Javascript, ad esempio, il modello di esecuzione prevede la presenza di una coda dei messaggi, in cui il driver del sottosistema interrogato provvede ad inserire il risultato
 - Il driver è eseguito in un thread separato
 - Il thread principale esegue costantemente un ciclo, in cui attende la presenza di un messaggio, lo estrae dalla coda e lo elabora
 - Quando la risposta arriverà, questa verrà naturalmente elaborata dal ciclo di elaborazione dei messaggi

Esecuzione asincrona



Esecuzione asincrona in Javascript



<https://javascript.plainenglish.io/what-is-synchronous-async-single-threaded-execution-context-browser-apis-256d906b186d>

Continuazioni e callback

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

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

L'inferno delle callback

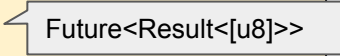

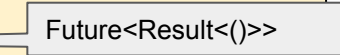
- Questo modo di scrivere il codice, tuttavia, crea grossi problemi quando l'azione che si sta compiendo richiede, **in cascata**, una seconda azione asincrona
 - Occorre infatti che la callback indicata provveda ad invocare un'ulteriore operazione passando la relativa callback
- Le cose si complicano se tale operazione è ciclica e se occorre gestire eventuali errori
 - Occorre trasformare il codice in una macchina a stati finiti
- Gli errori possono originarsi in momenti molto diversi
 - All'atto dell'invocazione della funzione asincrona
 - Come conseguenza dell'elaborazione asincrona

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

Esecuzione parziale

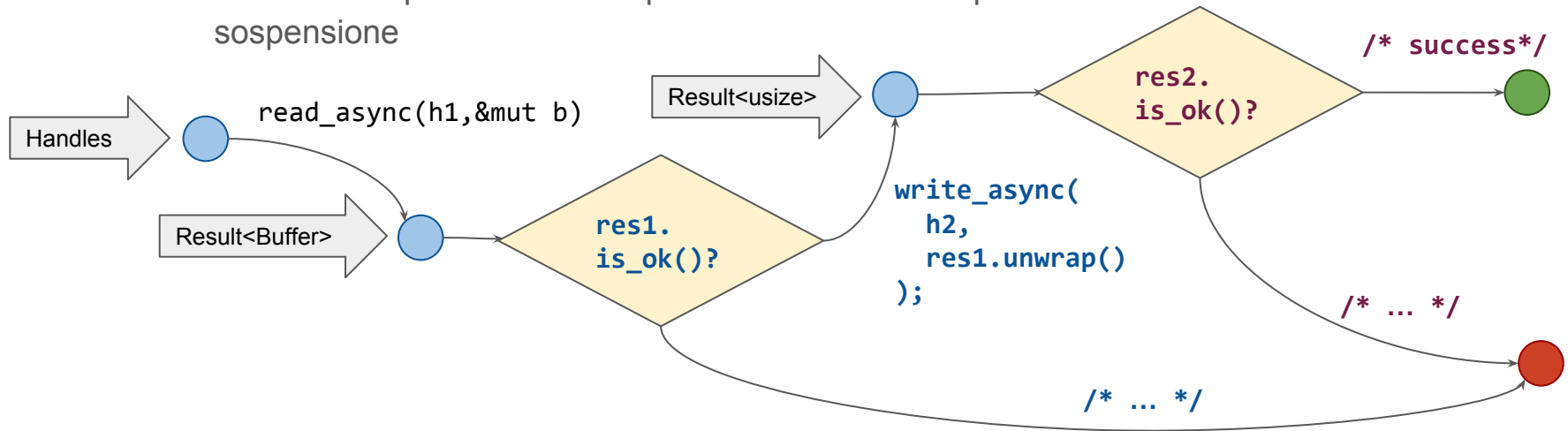
- Un primo passo che aiuta a semplificare il problema è provare a riscrivere la forma **annidata** delle callback in una forma **lineare**
 - Appoggiandosi ad una struttura dati che mantenga le informazioni di stato (**future**)

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

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

Esecuzione parziale

- Questa forma aiuta a vedere come le operazioni che si intendono eseguire possano essere viste come una macchina a stati finiti, che evolve "a strappi"
 - Quando raggiunge uno stato intermedio ritorna e, per continuare, sarà necessario riprenderne l'esecuzione passando come parametro l'esito dell'operazione asincrona che ha causato la sospensione



Esecuzione parziale

- La macchina a stati finiti può essere implementata da una chiusura
 - Essa racchiude il proprio **stato** e tutte le **variabili locali** di cui l'esecuzione ha bisogno
 - Quando viene eseguita ritorna un valore che indica se ha raggiunto uno degli stati finali o se si trova ancora in uno stato intermedio
 - Questo permette di non bloccare il thread corrente e procedere con l'esecuzione di altri task
- I punti di blocco sono tutti in corrispondenza degli stati intermedi
 - Questi sono immediatamente preceduti dall'invocazione di una funzione asincrona
 - Quando uno stato intermedio viene raggiunto, la chiusura ritorna
- Perché la macchina a stati possa progredire occorrono due condizioni:
 - Che la funzione asincrona invocata scateni qualche attività (in che thread avviene?) che possa portare all'evoluzione dello stato
 - Che ci sia qualcuno che indichi che vi sono le condizioni affinché lo stato possa evolvere

Async e await

- Il compilatore Rust supporta esplicitamente la programmazione asincrona grazie all'introduzione di due parole chiave nel linguaggio (**async** e **await**) ed alla presenza di un tipo specifico (**Future**) nella libreria core
 - Se una funzione o un blocco di codice sono preceduti dalla parola chiave **async**, il compilatore ne analizza il contenuto e lo trasforma in una macchina a stati
 - La funzione o il blocco vengono ridotti all'inizializzazione di tale macchina a stati
 - Il tipo restituito viene trasformato da **T** in un **tipo anonimo** che implementa il tratto **Future** al cui interno è implementata la macchina a stati precedentemente sintetizzata

```
async fn copy(file1: String, file2: String) -> Result<()> { ...codice }
```

```
fn copy(file1: String, file2: String) -> impl Future<Output = Result<()>> { ...altro }
```



Async e await

- Se all'interno del codice della funzione è presente una chiamata ad un'altra funzione asincrona, per poter accedere al suo risultato occorre esplicitamente attenderlo, attraverso l'operatore `.await`
 - Questo introduce un nuovo stato all'interno della macchina a stati associata alla funzione

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

Il tratto Future

```
use std::pin::Pin;
use std::task::{Context, Poll};

pub trait Future {
    type Output;

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

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

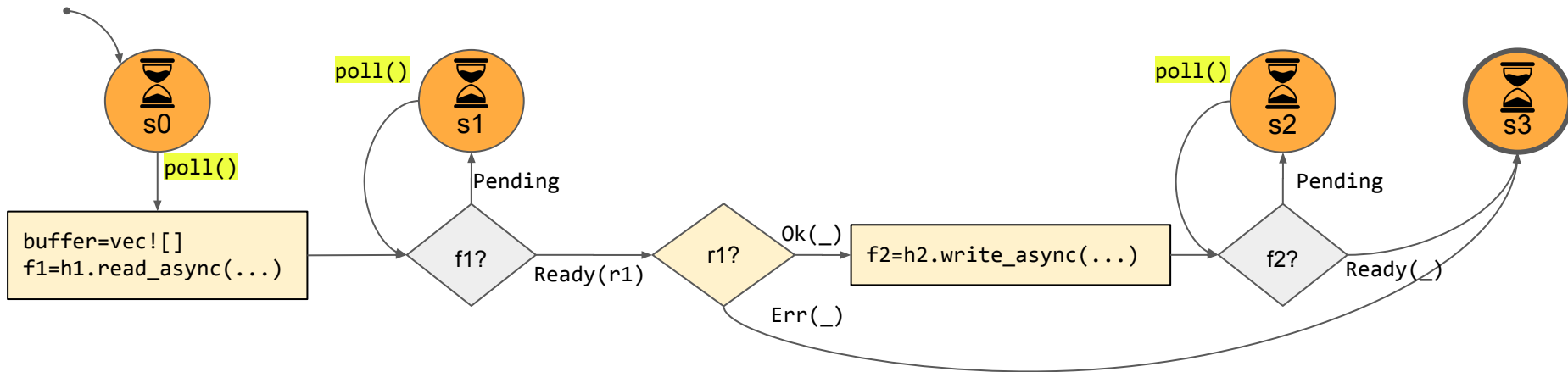
- **Pin<T>** è un particolare smart pointer che impedisce che la struttura venga mossa
 - Permettendo l'uso di riferimenti relativi
- **Context** incapsula un oggetto di tipo **Waker**, mediante il quale è possibile notificare all'esecutore che il metodo **poll(...)** può essere richiamato

Il tratto Future

- Tratto che viene implementato dagli oggetti che descrivono una computazione asincrona
 - Tale computazione può essere ancora in corso o essere terminata
- Il tratto offre un singolo metodo, **poll(...)**, che implementa la logica della macchina a stati associata alla funzione
 - L'oggetto che implementa il tratto mantiene al suo interno lo stato corrente
- Il metodo **poll(...)** restituisce enum che può assumere due soli valori:
 - **Poll:Pending** - per indicare che la computazione è ancora in corso
 - **Poll::Ready(val)** - per indicare che la computazione è terminata e ha come risultato **val**
- Un oggetto che implementa questo tratto è **inerte**
 - Affinché la computazione proceda, occorre che qualcuno ne invochi il metodo **poll(...)**

Generare la macchina a stati

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



Generare la macchina a stati

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

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

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

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

```
struct S3 {}
```

Generare la macchina a stati

- Inoltre genera una enum che racchiude i possibili stati e implementa il tratto Future

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

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

Generare la macchina a stati

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

Generare la macchina a stati

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

Generare la macchina a stati

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

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

Implementare la funzione

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

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


Gestire l'esecuzione

- Chi invoca una funzione asincrona, ottiene come risultato un **Future** nel proprio stato iniziale
 - Affinché possa capitare qualcosa, occorre che ne venga invocato il metodo **poll(...)**
- Se la funzione asincrona è chiamata all'interno di un'altra funzione asincrona, diventa automaticamente parte della macchina a stati del chiamante
 - Come succede per le funzioni **read_async(...)** e **write_async(...)** mostrate nell'esempio
 - Sarà responsabilità del chiamante della funzione esterna, gestire il **Future** risultante
- Se si invoca una funzione asincrona all'interno di una funzione "normale", occorre gestire il risultato di tipo **Future** in modo esplicito
 - Per farlo occorre disporre di un **Executor**

Gestire l'esecuzione

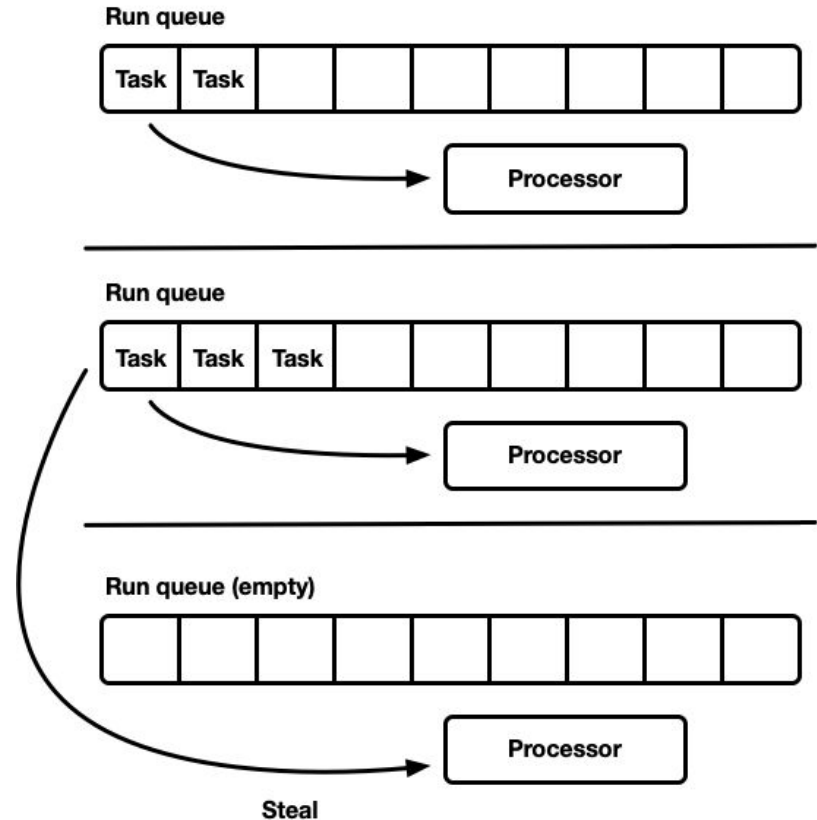
- Se il compilatore supporta la generazione automatica dei tipi che implementano la macchina a stati associata ad una funzione asincrona e la riscrittura delle funzioni in modo opportuno, nessun supporto è invece offerto dal linguaggio per la gestione dell'esecuzione
 - Il programmatore può scegliere quale libreria adottare nel proprio progetto, in base alle specifiche necessità
- Sono disponibili diverse librerie alternative per questo scopo
 - **Tokio** - l'ambiente più diffuso, con supporto per connessioni di rete, database, ...
 - **smol** - un ambiente semplificato, a basso impatto sulle risorse adatto a sistemi embedded
 - **async-std** - ambiente che offre la controparte asincrona delle librerie standard bloccanti

Gestire l'esecuzione

- I diversi ambienti di esecuzione non sono equivalenti
 - **Tokio** implementa un ciclo reattivo proprio, basato sul *crate* ad alte prestazioni **mio** (Metal I/O), che non è compatibile con i tratti usati dagli altri due
- Un runtime può basarsi su un singolo ciclo reattivo e/o utilizzare un thread-pool cui delegare l'esecuzione di più **Future** in parallelo
 - In questo caso, è possibile che l'elaborazione di una funzione asincrona inizi in un thread ma sia continuata in un thread differente
 - Questo implica che tutti i valori utilizzati nella funzione asincrona il cui uso si estende in più stati devono implementare il tratto **Send**, mentre i riferimenti devono implementare il tratto **Sync**

Architettura di elaborazione

- Tokio opera utilizzando un certo numero di code di esecuzione (default: n° di core)
 - Ogni coda è gestita da un ciclo reattivo (Processor) che estrae ed esegue i task presenti
 - Quando un Processor esaurisce i task della propria coda, prova a rubarne alcuni ad altre code, su base euristica
- L'intero algoritmo è ottimizzato per ridurre al minimo la sincronizzazione
 - Prevalentemente usando oggetti atomici
- <https://tokio.rs/blog/2019-10-scheduler>



Impostare un progetto con Tokio

- Oltre ad aggiungere l'opportuna dipendenza nel file Cargo.toml, indicando anche quali funzionalità si intendono attivare per il crate, occorre strutturare il punto di ingresso del programma in modo opportuno:

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

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

Task in Tokio

- `tokio::task::spawn(f: T) -> JoinHandle<T::Output>`
 where `T: Future + Send + 'static,`
 `T::Output: Send + 'static`
 - Funzione che definisce un *task*, eseguito in modo concorrente con altri *task*, la cui esecuzione inizia subito (a differenza di un *Future* di cui occorre invocare l'operatore `.await`)
 - L'oggetto passato come parametro può essere il risultato dell'invocazione di una funzione `async`, o essere un blocco `async` passato per valore

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

Attendere la terminazione di più Future

- La macro `join!(f1: impl Future, ..., fn: impl Future)` può essere invocata all'interno di una funzione/blocco `async` e forza l'attesa fino a che tutti i suoi parametri non sono completati
 - Restituisce una tupla contenente i risultati associati ai suoi parametri
- Una versione specializzata, `try_join!(...)`, può essere usata quando le espressioni passate come parametro hanno come valore di ritorno `Result<T,E>`
 - In questo caso viene restituito un oggetto `Result` che contiene, se tutti i `Future` hanno avuto successo, una tupla con i relativi risultati, oppure, in corrispondenza del primo fallimento, l'errore corrispondente

Attendere la terminazione di più Future

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

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


Selezionare il primo Future che si completa

- La macro **select!(...)** permette di attendere su più rami asincroni, eseguiti nell'ambito dello stesso thread, quello che termina per primo
 - Cancellando l'esecuzione dei restanti
 - Può essere usata solo all'interno di funzioni/blocchi asincroni
- Al suo interno è possibile inserire condizioni della forma
 - **<pattern> = <async expression> (, if <precondition>)? => <handle>**
 - **else => <expression>**
 - Il ramo **else**, se presente, viene valutato solo se nessuno dei rami precedenti ha avuto successo

Selezionare il primo Future che si completa

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

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

Gestione del tempo

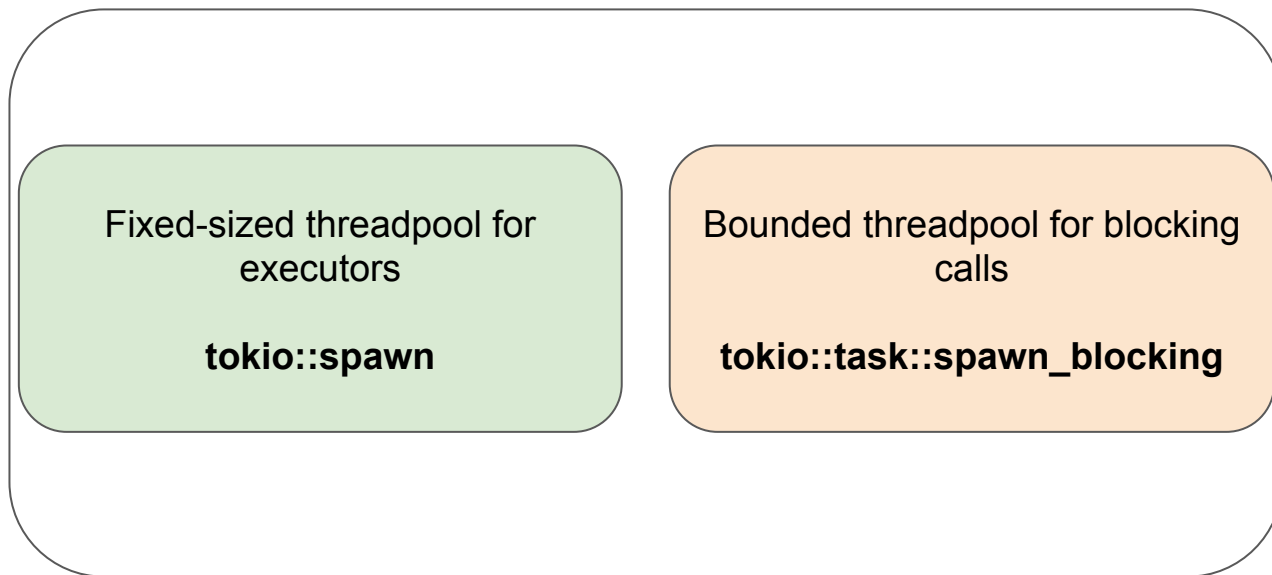
- La funzione **`tokio::time::sleep(d: Duration).await`** sospende l'esecuzione del task corrente per un tempo pari alla durata indicata
 - Durante l'attesa non viene consumata nessuna risorsa, se non la memoria necessaria a descrivere l'oggetto Future corrispondente
 - Allo scadere del tempo, l'esecuzione procede normalmente
- La funzione **`tokio::time::timeout(d: Duration, f: F).await`** attende per un tempo massimo pari alla durata indicata che il Future passato come secondo parametro si completi e restituisce un valore di tipo **`Result<T, Elapsed>`**
 - Se l'esecuzione si completa in tempo, il risultato è positivo e contiene il valore restituito dal Future, altrimenti riporta un errore di tipo `Elapsed`

Eseguire compiti computazionalmente intensi

- Se, in una funzione asincrona, occorre eseguire un compito computazionalmente intenso, la cui durata possa superare il centinaio di microsecondi, è bene richiedere che venga eseguito in un thread apposito
 - Così da evitare di introdurre latenza nell'elaborazione degli altri task
 - Si utilizza la funzione `tokio::task::spawn_blocking(f: FnOnce()->R)`
- Occorre evitare di lanciare troppi task di questo tipo
 - Durante la loro esistenza, infatti, tenderebbero a richiedere l'uso di CPU introducendo contesa con i thread deputati a elaborare le code dei messaggi, aumentando la latenza complessiva del sistema
 - Può essere opportuno condizionarne l'esecuzione alla disponibilità di risorse globali (usando, ad esempio, un semaforo) o usare esecutori ad hoc, come quelli offerti dalla libreria **Rayon**

Eseguire compiti computazionalmente intensi

Tokio's Runtime



Condividere dati tra task

- Se due task hanno bisogno di condividere una struttura dati, questa deve essere opportunamente protetta
 - A seguito del meccanismo di *work stealing* adottato dallo schedulatore, è infatti possibile che l'esecuzione avvenga in thread differenti
 - Occorre pertanto adottare le stesse precauzioni e strategie già viste con la programmazione *multithread*
- Tokio mette a disposizione una ricca serie di primitive asincrone nel modulo **tokio::sync**
 - Alcune basate sulla condivisione dello stato (**Barrier**, **Mutex**, **Notify**, **RwLock**, **Semaphore**)
 - Altre basate sulla comunicazione di messaggi (canali **oneshot**, **mpsc**, **broadcast**, **watch**)

Arc/Mutex - stato condiviso

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

Canali oneshot - Invio di un solo messaggio

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

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

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

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

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


Canali mpsc - Multiple Producer Single Consumer

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

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

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

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

Canali broadcast - comunicazione multi-molti

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

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

Canali watch - pattern Observer

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

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

Implementare un semplice server Http

```
use tokio::io::AsyncWriteExt;
use tokio::net::{TcpListener, TcpStream};
use tokio::task;

#[tokio::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1:8181").await.unwrap();

    loop {
        let (stream, _) = listener.accept().await.unwrap();
        tokio::spawn(handle_connection(stream));
    }

    //continua...
```

Implementare un semplice server Http

```
//...continua
```

```
async fn handle_connection(mut stream: TcpStream) {  
    let contents = "{\"message\": \"Hello, Tokio!\"}";  
  
    let response = format!(  
        "HTTP/1.1 200 OK\r\nContent-Type: application/json\r\nContent-Length: {}\r\n\r\n{}",  
        contents.len(),  
        contents  
    );  
  
    stream.write(response.as_bytes()).await.unwrap();  
    stream.flush().await.unwrap();  
}
```

Prestazioni a confronto

- I costi di creazione e cambiamento di contesto tra attività asincrone e thread sono analizzati nel sito
 - <https://github.com/jimblandy/context-switch>
- Il sito riporta la metodologia, il codice e i risultati ottenuti su un elaboratore specifico, con il relativo sistema operativo
 - Gli ordini di grandezza sono comunque interessanti

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

Link

- Asynchronous Programming in Rust
 - <https://rust-lang.github.io/async-book/>
 - Trattazione dettagliata dei principi relativi alla programmazione asincrona in Rust
- Fearless Concurrency with Rust
 - <https://medium.com/pragmatic-programmers/fearless-concurrency-with-rust-part-3-asynchronous-concurrency-e23bad856087>
 - Trattazione esemplificativa dell'elaborazione asincrona applicata ad un server web
- What's a "Thread Boundary" in Rust's Async-Await ?
 - <https://cotigao.medium.com/whats-a-thread-boundary-in-rust-s-async-await-f783cff55c99>
 - Approfondimento sul meccanismo di *work stealing* usato in Tokio
- Zero to Production in Rust
 - L. Palmieri, 2022, ISBN 979-8847211437
 - <https://www.zero2prod.com/>
 - Introduzione pratica e dettagliata allo sviluppo di componenti backend in Rust

