Si considerino le seguenti strutture dati e rispettive porzioni di codice. Per ciascuna di esse si indichi la dimensione di memoria allocata nello stack e nello heap.

```
// Struttura 1
use std::rc::Rc;
let rc: Rc<u64> = Rc::new(42);
let rc2 = rc.clone();
let wk = Rc::downgrade(&rc);

// Struttura 2 e codice di supporto
Let mut vector = Vec::<u64>::with:capacity(8);
for i in 0..5 {
vector.push(i);
}
let vslice = &vector[1..3];
```

Si considerino le seguenti strutture dati e rispettive porzioni di codice. Per ciascuna di esse si indichi la dimensione di memoria allocata nello stack e nello heap.

```
// Struttura 1
use std::rc::Rc;
let rc: Rc<u64> = Rc::new(42);
let rc2 = rc.clone();
let wk = Rc::downgrade(&rc);

// Struttura 2 e codice di supporto
Let mut vector = Vec::<u64>::with:capacity(8);
for i in 0..5 {
vector.push(i);
}
let vslice = &vector[1..3];
```

Struttura 1
-Stack: rc (8byte),rc2(8byte) e wk(8byte) (puntatore al dato nello heap)
-Heap: un solo blocco di dati con strong counter = 2, weak counter=1 e u64 contenente il numero 42

Struttura 2
-Stack: vector: fat pointer allo heap (8 byte per il campo size, 8 byte per il campo capacity, 8 byte per il puntatore effettivo)
vslice: un puntatore allo heap nella stessa zona di memoria di vector contenente 8byte per la dimensione dello slice e 8byte per il puntatore.
-Heap: vector: Vec allocato di 8*sizeof(u64)

Si consideri il programma seguente che riporta la numerazione delle linee di codice.

```
1 use std::sync::{Arc, Mutex, Condvar};
2 use std::thread;
3
4 fn main() {
5     let data = Arc::new((Mutex::new(Vec::new()), Condvar::new()));
6
7     let data_clone1 = Arc::clone(&data);
8     let data_clone2 = Arc::clone(&data);
9
10    let writer = thread::spawn(move || {
11        let (lock, cvar) = &*data_clone1;
12        for i in 1..=5 {
13            thread::sleep(std::time::Duration::from_secs(1));
14
15            let mut data = lock.lock().unwrap();
16            data.push(1);
17            cvar.notify_all();
18        }
19    });
20
21    let reader1 = thread::spawn(move || {
22        let (lock, cvar) = &*data_clone2;
23        loop {
24            let mut data = lock.lock().unwrap();
25
26            data = cvar.wait(data).unwrap();
27
28            if let Some(value) = data.pop() {
29                println!("Reader 1: read {}", value);
30            }
31            else {
32                println!("(1)");
33            }
34        }
35    });
36
37    let reader2 = thread::spawn(move || {
38        let (lock, cvar) = &*data;
39        loop {
40            let mut data = lock.lock().unwrap();
41
42            data = cvar.wait(data).unwrap();
43
44            if let Some(value) = data.pop() {
45               println!("Reader 2: read {}", value);
46            }
47            else {
48              println!("(2)");
49            }
50        }
51    });
52    writer.join().unwrap();
53    reader1.join().unwrap();
54    reader2.join().unwrap();
55 }
```

-Il problema si pone in quanto il thread writer esegue solo 5 operazioni di scrittura, mentre i reader hanno un loop infinito.
Di conseguenza, una volta consumati i 5 elementi del writer, entrambi i reader entreranno in una wait perenne, portando ad un classico deadlock.
-Si potrebbe risolvere questo problema utilizzando un canale di sincronizzazione SPMC che permetta al writer di inserire i dati nel canale e ai reader di accedere ad essi o mettersi in attesa finché il writer rimane in vita. Una volta quindi terminato il ciclo del writer, il thread terminando dovrebbe eseguire una drop del Sender in modo tale da far ritornare un errore alla Receiver sul metodo recv() e permettendo di uscire dal loop con un match che consideri anche l'opzione di errore. In questo modo il thread riuscirebbe a terminare le join dei thread e a terminare correttamente il suo processo.
-Una alternativa ad usare channel potrebbe essere quella di aggiungere al Mutex, oltre al vettore, un booleano che indichi che il writer ha finito. I reader possono in questo modo ad ogni iterazione, appena acquisito il lock, controllare questo bool ed eseguire una return in caso il writer abbia finito.

```
//nel main inizializzo il mutex con una tuple
let data = Arc::new( ( Mutex::new( ( Vec::new() , false) ), Condvar::new()));


//in entrambi i reader
//subito dopo lock.lock().unwrap();
if(*data.1)==true{
  return;
}
```

Con la attuale configurazione le istruzioni alle righe 32 e 48 verranno eseguite finché il thread writer non si sveglierà e non immetterà il suo elemento nel Vec.

Si implementi in RUST la struct **CountDownLock** che permette a uno o più thread di attendere che un gruppo di operazioni eseguite da altri thread siano eseguite.

Essa incapsula un contatore ed offre i seguenti tre metodi thread-safe (oltre alla propria funzione costruttrice) che devono essere implementati:

fn new(n: usize) -> Self

// inizializza la struttura, impostando ad n il contatore

fn count_down(&self)

// decrementa il contatore, se maggiore di 0

fn wait(&self)

// blocca l'esecuzione del chiamante senza consumare cicli di cpu, finché il contatore non diventa zero

fn wait_timeout(&self, d: Duration) -> std::sync::WaitTimeoutResult

// blocca l'esecuzione del chiamante senza consumare cicli di cpu, in attesa cha il contatore raggiunga 0 per una durata massima pari a d, restituendo il risultato dell'attesa.

Si descriva nel dettaglio il comportamento di questo programma.

Si spieghi l'errore di compilazione che viene generato.

Che cosa deve essere modificato per permetterne la compilazione in modo da ottenere la visualizzazione di "Numero: 4"?

```
fn main() {
   let mut data = vec![1, 2, 3, 4, 5];
   data.push(60);
   let mut process_data = move || {
        data.push(50);
        let count = data.iter().filter(|&x| x % 2 == 0).count();
        println!("Numero: {:?}", count);
    };
```

```
    data.push(40);
```

```
    process_data();
```

```
    data.push(30);
```

```
}
```

Si descriva nel dettaglio il comportamento di questo programma.

Si spieghi l'errore di compilazione che viene generato.

Che cosa deve essere modificato per permetterne la compilazione in modo da ottenere la visualizzazione di "Numero: 4"?

```
fn main() {
   let mut data = vec![1, 2, 3, 4, 5];
   data.push(60);
   let mut process_data = move || {
        data.push(50);
        let count = data.iter().filter(|&x| x % 2 == 0).count();
        println!("Numero: {:?}", count);
    };
```

```
    data.push(40);
```

```
    process_data();
```

```
    data.push(30);
```

```
}
```

Il programma in questione inizializza un vettore di i32, inserisce un elemento, poi definisce una closure process_data che prende possesso delle variabili libere usate all'interno. Il programma poi inserisce un nuovo elemento (40) , chiama process_data() che inserisce un elemento (50) e conta quanti elementi pari contiene il vettore. in seguito proverà ad inserire un nuovo elemento (30) nel vettore ma questo non sarà possibile in quanto process_data avrà preso possesso del vec.
Per risolvere il problema di compilazione bisogna passare alla closure un clone di data, il che è fattibile in quanto Vec implementa Clone.
quindi ad esempio si potrebbe scrivere prima di let mut process_data...:
let mut data_clone = data.clone();
e usare data_clone all'interno della closure.