Esercitazione 2

5 / 12 aprile 2023

Obiettivi

Eseguendo correttamente gli esercizi si avrà padronanza delle seguenti attività

- chiamare da Rust delle system call unix
- serializzazione / deserializzazione di oggetti complessi Rust in formato binario
- definire di più eseguibili all'interno di un crate
- condividere informazioni tra processi attraverso file ed eseguire Lock sui file
- definire lifetime per accedere per riferimenti mutabili ad elementi di strutture dati complesse
- leggere la struttura del file system

Esercizio 1

Creare un progetto rust in cui ci sono due eseguibili (ciascuno con un proprio main e lanciabili indipendentemente) che devono fare le seguenti operazioni:

- un eseguibile chiamato producer legge ad intervalli di 1s dei dati da dieci sensori, li salva su un file appena appena letti, scrivendoli in formato binario (simulare i sensori con una funzione random che restituisce 10 valori casuali)
- un eseguibile chiamato consumer ogni 10 secondi legge i dati dal file e stampa la media, il minimo e il massimo dei valori di ciascun sensore

I due processi una volta fatti partire entrano in un loop infinito e continuano a funzionare fino a quando non si preme ctrl-c .

La struttura su cui salvare i dati è definita nel modo seguente:

```
#[repr(C)]
struct SensorData {
	seq: u32, // sequenza letture
	values: [f32; 10],
	timestamp: u32
}
```

Sul file la struttura deve essere serializzata in binario (utilizzare le funzioni scritte nella esercitazione 0 o realizzarle se non sono ancora state fatte).

Siccome entrambi i processi leggono e scrivono sullo stesso file, occorre definire come sono organizzati i dati e come gestire l'accesso in simultanea.

Organizzazione file scambio dati

Per l'organizzazione dei dati si utilizzi un buffer circolare

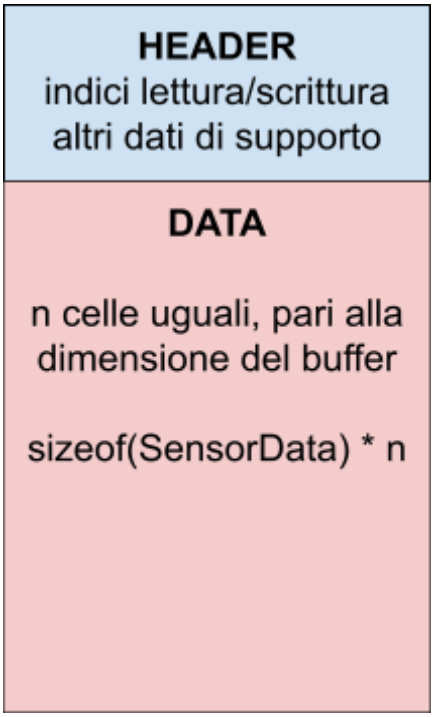
https://en.wikipedia.org/wiki/Circular_buffer

Ricordiamo che un buffer circolare ha:

- una dimensione fissa, arrivati alla fine si torna a scrivere dall'inizio
- un puntatore (indice) alla prima cella libera in cui scrivere
- un puntatore (indice) alla prima cella da cui leggere

Quando il buffer è pieno non è possibile scrivere, occorre attendere che i valori vengano letti.

Il nostro file quindi avrà la seguente organizzazione, con un header che contiene i metadati di supporto per gestire il buffer circolare e un blocco "data" di dimensione fissa, che contiene n struct SensorData



Accesso simultaneo e lock

Per quanto riguarda l'accesso simultaneo invece occorre usare delle primitive di sistema per accedere in modo esclusivo: quando un processo usa il file condiviso, l'altro processo deve sapere che il file è in uso e attendere.

- Per chi ha Mac, Linux, e Windows con il Linux Subsystem si può usare la system call posix fcntl, importando questo crate:

  https://docs.rs/fcntl/latest/fcntl/

  Attraverso fcntl è possibile ottenere un lock esclusivo su file, in modo che solo un processo per volta possa accedere al file. Se il file è libero chiamando "lock_file" si ottiene la possibilità di accedervi in modo esclusivo (lock=true), se il file è già in

possesso di un altro processo invece si ottiene lock=false.

Quindi ciascun processo, al momento di leggere / scrivere il file, dovrà assicurarsi di avere il “lock”, leggere/scrivere i propri dati, aggiornare i puntatori nell’header il più velocemente possibile e, infine, rilasciare il lock con “unlock_file”. Se non può accedervi continua a riprovare finché non ottiene il lock.
Notare che lock_file non garantisce un accesso esclusivo del file, un programma può ignorare il lock e andare avanti. La system call è solo di tipo “advisory”, ovvero un processo segnala che userà il file in modo esclusivo e gli altri processi devono controllare che l’accesso sia libero tramite lock_file.

- Per chi avesse problemi ad avere il Linux Subsystem installato su Windows utilizzare il seguente crate:
  https://crates.io/crates/fslock
  Questa libreria deve essere utilizzata con un file di lock diverso dal file contenente il buffer circolare, perché restituisce solo un un oggetto di tipo LockFile che serve per capire se è possibile andare avanti: il processo che ottiene il lock prosegue, l’altro aspetta finché il primo non ha finito.

Altre note

Nel caso in cui il buffer sia pieno il producer salta il valore da scrivere e passa alla prossima lettura. Se non ci sono abbastanza valori invece il consumer utilizza ugualmente i valori letti e fa andare avanti l’indice di lettura. Il consumer segnala anche quando ci sono buchi nella sequenza di lettura dei valori.

Utilizzare un circular buffer di dimensione 20 (non dovrebbe mai riempirsi) e 10 (ogni tanto il producer dovrebbe trovarlo pieno).

Suggerimento: testare due processi separati che lavorano in simultanea è molto complicato e spesso è anche difficile determinare se si comportano in modo corretto. Possibili strategie

- iniziare a lanciare i programmi alternativamente: prima il producer, fagli scrivere qualche valore, bloccarlo, e poi lanciare il consumer
- scrivere degli unit test per ciascun processo simulando delle situazioni limite (buffer pieno, in underflow ecc)
- testarlo con dei pattern di valori predicibili (es i valori dei sensori fissati da 1 a 10), in modo da capire immediatamente se si comporta in modo corretto

Per ottenere due eseguibili nello stesso crate:
https://doc.rust-lang.org/cargo/reference/cargo-targets.html

Bonus: analizzando il codice di lock_file provare a chiamare direttamente la funzione di fcntl per ottenere il lock.

Esercizio 2

Realizzare un programma che gestisca in memoria la copia di un directory del proprio file system e permetta semplici query e manipolazioni dei file e delle cartelle.
Per realizzarlo utilizzare le seguenti strutture dati:

```
enum FileType {
   Text, Binary
}

struct File {
   name: String,
   content: Vec<u8>, // max 1000 bytes, rest of the file truncated
   creation_time: u64,
   type_: FileType,
}

struct Dir {
   name: String,
   creation_time: u64,
   children: Vec<Node>,
}

enum Node {
   File(File),
   Dir(Dir),
}

struct FileSystem {
   root: Dir
}
```

Scrivere quindi i seguenti metodi di implementazione di FileSystem (aggiungere il paremtro self all'occorrenza):

- new(): crea un file system vuoto
- from_dir(path: &str): lo crea copiando in memoria tutti i contenuti di una cartella esistente a partire da path
- mk_dir(path: &str): crea una nuova cartella; ad esempio con mkdir("/a/b") crea una nuova cartella "b" dentro "a", se "a" non esiste fallisce
- rm_dir(path: &str): cancella una cartella, solo se vuota
- new_file(path: &str, file: File): crea un nuovo file nel file system
- rm_file(path: &str): elimina il file
- get_file(path: &str) -> Option<&mut File>: ottiene un riferimento mutabile a un file

- search(queries: &[&str]) -> Option<MatchResult> cerca dei file che matchano le query indicate e restituisce un oggetto MatchResult con un riferimento mutabile ai file trovati

Le query sono definite nel modo seguente:

- “name:stringa” -> tutti i file che contengono stringa nel nome
- “content:stringa” -> tutti i file di testo che contengono stringa
- “larger:val” -> tutti i file più grandi di val
- “smaller:val” -> tutti i file più piccoli di val
- “newer:val” -> tutti i file più recenti di val
- “older:val” -> tutti i file più vecchi di val

In una search è possibile specificare una o più query, che vengono applicate in OR.
Il risultato è un oggetto MatchResult

```
struct MatchResult {
        queries: Vec<&str>, // query matchate
        nodes: Vec<&mut File>
}
```

Attenzione: i risultati contengono riferimenti mutabili a File/Dir, in modo da poter direttamente modificare nome/contenuto/data creazione, quindi attenzione a una corretta definizione dei lifetime!!!