Esercitazione 2

5 / 14 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 più eseguibili all'interno di un crate
- condividere informazioni tra processi attraverso file ed eseguire Lock
- definire lifetime per accedere per riferimenti mutabili a elementi di strutture dati complesse
- leggere la struttura del file system

Esercizio 1

Un sistema legacy scritto in C deve essere esteso con nuove funzionalità scritte in Rust. Per minimizzare le modifiche al codice esistente si è deciso di salvare i dati gestiti dal sistema legacy su file e il programma Rust li deve leggere per processarli. L'obiettivo è realizzare l'interfaccia di comunicazione (salvataggio dati / lettura) dei due programmi.
I dati sono dei record definiti in C in questo modo

```
typedef struct {
    int type;
    float val;
    long timestamp;
} ValueStruct;

typedef struct {
    int type;
    float val[10];
    long timestamp;
} MValueStruct;

typedef struct {
    int type;
    char message[21]; // stringa null terminated lung max 20
}  MessageStruct;

typedef struct {
    int type;
    union {
        ValueStruct val;
        MValueStruct mvals;
```

```
        MessageStruct  messages;
    };
} ExportData;
```

I dati da esportare sono di tre tipi:

- misure di vario tipo memorizzate in `ValueStruct` / `MValueStruct`
- messaggi di vario tipo memorizzati in `MessageStruct`.

I dati vengono poi incapsulati in una `struct ExportData` che può contenere un qualsiasi dato da esportare, con il tipo indicato in type (1=Value 2=MValue 3=Message)

Scrivere quindi un programma C che crei un vettore con 100 valori da esportare e li salvi in modo binaro con la seguente funzione, dove `*fp` è un file aperto precedentemente:

```
void export(ExportData *data, int n, FILE *pf) {
    fwrite(data, sizeof(ExportData), )
}
```

Creato il file, scrivere in programma Rust che:

- prenda da command line il nome del file da leggere
- definisca una struttura dati idonea a contenere i dati lettti (`struct CData`)
- la struct abbia un metodo `from_file` in grado di leggere il dati da file aperto
- memorizzare i dati letti in un array di `struct CData`

Attenzione: vi sono sono due modi possibili per leggere il contenuto del file binario in Rust e creare degli oggetti:

- leggere i byte degli attributi uno per uno e poi convertirli nel tipo desiderato con il trait `from_le_bytes` / `from_be_bytes`
- leggere tutta la struct in un byte buffer una volta noto tipo e dimensione, e poi reinterpretarla come struct Rust

Nel secondo caso si deve usare del codice unsafe, perché? Quale dei due approcci è più efficiente e perché? È possibile con tutte le struct definite o vi sono problemi?

Esercizio 2

Questo esercizio sotto Windows deve essere svolto all'interno dell'ambiente WSL (Windows Linux Subsystem) in quanto si dovranno usare delle API Unix.

Due processi Rust devono eseguire le seguenti operazioni

- un processo chiamato producer legge a intervalli di 1s dei dati da dieci sensori, li salva su un file scrivendoli in formato binario (simulare i sensori con una funzione random)
- un processo 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 nell'esercizio 1).

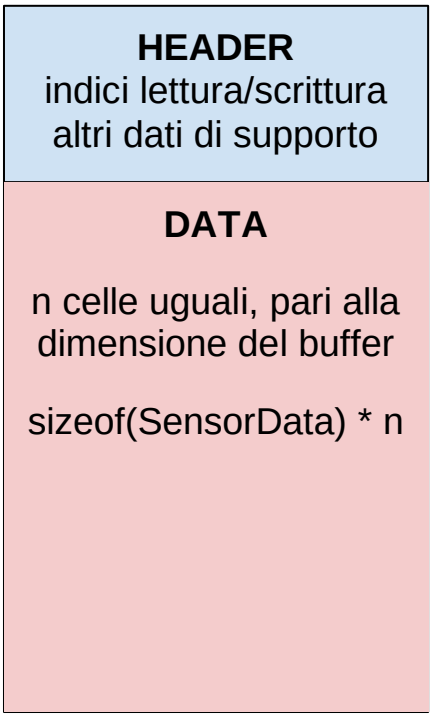
Poiché entrambi i processi leggono e scrivono sullo stesso file, occorre definire come sono organizzati i dati e come gestire l'accesso in simultanea.

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



Per quanto riguarda l'accesso simultaneo invece si usi 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 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`.

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:

- 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 3

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/dir 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/dir più recenti di val
- “older:val” -> tutti i file/dir 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 Node>
}
```

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!!!