LabSO1-AA_2019_2020–166233-201506-202224-204552, atmgso2020@gmail.com

Alberto Dal Lago, 166233, alberto.dallago@studenti.unitn.it

Mattia Salvador, 201506, mattia.salvador@studenti.unitn.it

Taras Rashkevych, 202224, taras.rashkevych@studenti.unitn.it

Gianluca Sassetti, 204552, gianluca.sassetti@studenti.unitn.it

Organizzazione del codice in file

Il progetto e' strutturato nel seguente modo:

- `src/`: contiene le sorgenti dei programmi del progetto, in particolare
  - `itoa.h/.c`: dichiarazione ed implementazione della funzione che produce la rappresentazione in formato stringa di un numero intero
  - `list.h/.c`: dichiarazione ed implementazioni di strutture dati e funzioni per la gestione di una linked-list ed di iteratori su di essi
  - `fs.h/.c`: dichiarazione ed implementazione di funzioni per l'accesso al file system (ottenere lista di file contenuti in una directory, verificare che un file e' una directory)
  - `file_analysis.h/.c`: dichiarazione ed implementazione di strutture dati e funzioni per la gestione di file, caratteri ed occorrenze (parsing di stringhe nel formato `file:carattere:occorrenze`)
  - `settings.h`: definizione di macro per impostare le dimensioni di alcune strutture dati utilizzate
  - `bool.h`: implementazione del tipo e dei valori booleani
  - `utilities.h`: dichiarazione e implementazione di funzioni utili
  - `history.h/.c`: dichiarazione ed implementazione di una struttura dati per memorizzare i risultati delle analisi all'interno della shell
  - `reader.c`: implementazione del Reader (aka Q)
  - `slicer.c`: impementazione dello Slicer (aka P)
  - `partitioner.c`: implementazione del Partitioner (aka C)
  - `analyzer.c`: implementazione dell'Analyzer (aka A)
  - `report.c`: implementazione del Report (aka R)
  - `shell.c`: implementazione della Shell (aka M)
- `assets/`: contiene una gerarchia di file di testo rigorosamente ASCII
- `Makefile`: contiene le ricette per la compilazione, l'esecuzione e la verifica di correttezza dei programmi realizzati, e la pulizia della directory

Architettura

Il progetto e' stato realizzato come un insieme di programmi indipendenti che comunicano tra loro per realizzare quanto richiesto dalla consegna. Si e'

optato per utilizzare principalmente le `pipe` come tecnica di IPC, e di utilizzare i `thread` ed i `mutex` per parallelizzare la computazione di ogni processo garantendo l'accesso mutuamente esclusivo dei thread alle risorse condivise.

Le `fifo` sono state utilizzate solo in un caso, per mettere in comunicazione 2 processi che non hanno una gerarchia diretta (nello scenario di metterli in comunicazione quando sono stati avviati in due terminali separati). Nello scenario in cui ci sia una gerarchia diretta, sono comunque utilizzate le `pipe`.

Sono quindi stati realizzati i seguenti programmi:

- `reader`, che legge una slice specifica di ogni file ricevuto
- `slicer`, che avvia processi `reader` per ogni slice in cui sono "tagliati" tutti i file ricevuti
- `partitioner`, che avvia processi `slicer` su ogni partizione in cui sono stati partizionati i file ricevuti
- `analyzer`, che si trasforma in un processo `partitioner` dopo aver "espanso" le directory ricevute
- `report`, che riceve informazioni computate dai processi precedentemente illustrati, ed effettua delle statistiche sulle occorrenze dei caratteri
- `shell`, una interfaccia testuale interattiva che consente all'utente di utilizzare piu' agevolmente il sistema, mettendo autonomamente in comunicazione i processi `report` ed `analyzer`, e che consente l'importazione e l'esportazione delle analisi

Per una descrizione piu' approfondita di ogni programma realizzato, incluso come comunicano tra loro, si veda il file `README_PRO.html`

Eventuali problematiche e soluzioni di situazioni anomale

Abbiamo cercato di gestire le principali problematiche che possono verificarsi durante l'esecuzione del sistema:

- abbiamo gestito la correttezza dei parametri passati ai 3 programmi principali, `shell`, `analyzer` e `report`: ad esempio, `-m` essere seguito da una stringa valida, ovvero un numero intero positivo. I programmi "di appoggio", `reader`, `slicer` e `partitioner`, non fanno tali controlli
- abbiamo gestito la situazione in cui vengono "sporcati" i dati, o comunque sono passate stringhe *non* nel formato `file:carattere:occorrenze`, che vengono ignorate e non fanno crashare i programmi coinvolti
- abbiamo cercato di limitare i problemi che possono insorgere con le chiamate a funzioni e a system call, come `fork`, `exec`, `mkfifo`, ...
- ci risulta che tutte le risorse allocate nella heap vengano liberate non appena non sono piu' necessarie
- il comando `export` e' limitato alla directory corrente

- il comando `import` non verifica la corretta formattazione del file che gli e' passato
- il passaggio a `report` di un file non formattato secondo lo standard `file:carattere:occorrenze` non causa errori o interruzioni del programma, saranno semplicemente riportate a video 0 occorrenze. Allo stesso modo vengono ignorati tutti gli input che non rispettano tale formato.

Come testare il progetto

Eseguendo `make run` viene effettuata la compilazione dei programmi e viene avviata la `shell`, che mostrera' un riepilogo dei comandi disponibili.

Se si vogliono utilizzare i programmi in modalita' stand-alone, segue una lista di esempi di chiamata ai programmi (per maggiori informazioni, riferirsi a `README_PRO.html`)

- `bin/reader -s 1 -m 4 assets/info1.txt assets/info2.txt`
- `bin/slicer -m 4 assets/info1.txt assets/info2.txt`
- `bin/partitioner -m 4 -n 3 assets/info1.txt assets/info2.txt`
- `bin/report npipe allchars -ls -1`
- `bin/analyzer -m 10 -n 10 assets/ src/ Makefile -r`
- `bin/shell -m 10 assets`