IMPOSTAZIONI PROGETTO

versione documento: 1.0

Descrizioni

Per le descrizioni informali fare riferimento al file già pubblicato

Indicazioni

- Ogni gruppo sarà identificato dalla sigla LABSO1-2017_2018--*matricola1_matricola2_matricola3* (analogamente per eventuali eccezioni di gruppi con meno o più componenti) con le matricole in ordine crescente (es. LABSO1-2017_2018--1234560_1234570_1234580) definita "***ID gruppo***"
- **Iscriversi entro il giorno venerdì 11/05/2018** come gruppo al form online https://goo.gl/forms/iX5U8xlXHpC7YpIu2 indicando una mail di un referente (o di gruppo, comunque unica) ed eventuali annotazioni, oltre ai componenti del gruppo (3, salvo eccezioni) indicando per ciascuno "MATRICOLA, COGNOME, NOME COMPLETO" (es. "1234567, ROSSI, MARIO LUIGI") e al progetto scelto tra "Shell custom", "Analizzatore (gestione statistiche comandi)" e "Gestore I/O 8+8 bit su raspberry"
- Per il progetto impostare una cartella denominata come l'id del gruppo dentro cui devono esserci esattamente:
    - Un makefile denominato esattamente "Makefile" con almeno una regola "build" che generi il/gli eseguibile/i dentro una sottocartella "bin" (da crearsi al volo) e una "clean" che elimini ogni eventuale file eseguibile e temporaneo.
    - Una sottocartella "src" contenente tutti gli eventuali sorgenti (files e/o ulteriori sottocartelle come necessario).
    - Un eventuale file di descrizione "pdf" principalmente per eventuali peculiarità e/o per sottolineare soluzioni implementative o elementi che si ritengono particolarmente validi per la valutazione
- Entrando nella cartella da shell si deve poter lanciare il tool make (comando "make" da shell) che sfrutterà il makefile sopra indicato: entrando nella sottocartella "bin" dopo la generazione del/degli eseguibile/i si deve poter utilizzare il programma. *L'esecuzione deve essere verificata sotto Linux Ubuntu 14.x o 16.x*.
- La valutazione tiene conto di diversi fattori tra cui l'ordine (nel senso della pulizia: buona indentazione, nomi di variabili e funzioni chiari, etc.) di stesura di tutti i codici, che devono essere diffusamente e chiaramente commentati, dell'impostazione del makefile (utilizzando diverse regole per la compilazione correlando con eventuali dipendenze dove opportuno), della compilazione senza avvisi/errori (nota: errori bloccanti comportano una valutazione negativa), dell'organizzazione del codice C (preferibile la suddivisione in più files per le diverse funzionalità piuttosto che un singolo file monolitico e l'uso di MACRO e funzioni), della versatilità (più il programma è generico meglio è, con meno vincoli e limitazioni possibili) e della possibilità di essere configurato (ad esempio se ci sono file di supporto esterni scelte possibili dalla meno premiante alla più valida potrebbero essere: uso di un nome fisso tipo "./calc.txt", uso di una macro tipo "#DEFINE EXTNAME "./calc.txt", disponibilità di un'opzione tipo "... --extfile=./altronome.txt …" che l'utente può impostare in fase di esecuzione più un'eventuale richiesta interattiva tramite input dell'utente se non è impostato alcun nome). *Si valuta anche la discussione (padronanza degli argomenti, conoscenza dei contenuti, eventuali risposte a*

domande sui contenuti del laboratorio e/o del progetto stesso, capacità di dimostrare che le richieste di progetto sono soddisfatte, ad esempio che vengono generati tutti i processi richiesti*) singolarmente*.

- Gli argomenti accettati dall'eseguibile devono essere gestiti tramite *argc* e *argv*: è valutato positivamente saper gestire gli argomenti in qualunque ordine e poter usare versioni corte e lunghe (ad esempio un argomento "--outfile=NOMEFILE" oppure "-o NOMEFILE") come comune per i tipici comandi basilari della shell.
- Per ciascun progetto poi si bada alla risoluzione di un insieme minimo di problematiche come valutazione di partenza "sufficiente" che aumenta in base alla qualità delle soluzioni adattate e al numero di caratteristiche implementate, come meglio indicato nella sezione seguente.
- Dove non diversamente indicato i nomi degli eseguibili e degli argomenti sono a piacere (di seguito si riportano solo degli esempi).
- Si possono usare solo le caratteristiche viste a lezione e disponibili nelle slides e tutte le funzioni di manipolazione delle stringhe presenti in "string.h". Non si può usare alcuna librerie esterna aggiuntiva e la compilazione deve potersi effettuare senza passare argomenti speciali al tool "*gcc*" oltre eventualmente alle opzioni "-lncurses -lreadline" se si vogliono usare le librerie "curses" e "readline" (per gestire input interattivi avanzati) utilizzando documentazioni pubbliche
- **Inivare entro il giorno domenica 27/05/2018** il lavoro ultimato creando un file ".zip" con nome uguale all'ID del gruppo e estensione ".zip" che contiene la cartella di lavoro (quindi l'archivio CONTIENE la cartella, non direttamente i files, senza ulteriori contenuti): prestare attenzione che non vi siano file temporanei o "nascosti" magari aggiunti dal sistema! Tale file deve essere inviato come "consegna" tramite il form all'indirizzo https://goo.gl/forms/QJ2YjBoUEaNq6uOJ3 dove si può fare l'upload (solo in caso di problemi molto particolari indicare un link di download diretto con scadenza non inferiore al 30/06/2018).
- Nel campo "annotazioni" si possono indicare MOLTO SINTETICAMENTE osservazioni particolari e/o peculiarità che si ritengono importanti ai fini della valutazione (soprattutto particolari caratteristiche "avanzate" che si volessero sottolineare rispetto a una soluzione "basilare").
- Si consiglia di NON aspettare l'ultimo momento per iscrizioni/consegne per ovviare a possibili difficoltà per tempo.

CARATTERISTICHE PROGETTI

Shell custom

Un esempio di esecuzione potrebbe essere:

```
bash>./shell --outfile="/tmp/shell.out" --errfile="/tmp/shell.err" --maxlen=100 --code=true
shell>ls | wc
       1       1       6
shell>
```

dove `--outfile` e `--errfile` impostano rispettivamente il file di "log" per lo stdout e lo stderr, mentre `--maxlen` è il numero massimo di caratteri che vengono salvati sui file di log ad ogni esecuzione e `--code` indica se si vuole salvare anche il codice di ritorno. I primi due argomenti DEVONO essere implementati (anche con nomi diversi volendo), mentre gli altri sono due ipotesi di argomenti aggiuntivi che aumentano flessibilità rendendo gestibili alcune caratteristiche del programma attraverso delle opzioni. Nell'esempio dopo l'esecuzione ci si ritrova "dentro" la shell custom in cui posso eseguire comandi come se fossi nella shell "bash": in questo caso il file di log dell'output potrebbe contenere qualcosa come (è un esempio supponendo di aprire un'altra shell):

```
bash>cat "/tmp/shell.out"
-------------------------------------------------
COMMAND:     ls | wc
DATE:        Lun 30 Apr 2018 10:57:04
OUTPUT:

       1       1       6

RETURN CODE: 0
-------------------------------------------------
bash>
```

La soluzione come minimo deve accettare due argomenti per definire i nomi dei file di "log" e poter eseguire almeno comandi basilari come *ls*, *wc* e *date* e accettare il carattere di piping. Oltre questo l'ideale è che sia possibile inserire qualsiasi argomento per i comandi validi e magari qualsiasi comando sia possibile gestire anche con caratteri speciali.
Nell'esempio sopra sarebbe ancora meglio gestire SEPARATAMENTE i comandi *ls* e *wc* dentro il programma per cui nel file di output "shell.out" si avrebbe qualcosa tipo:

```
bash>cat "/tmp/shell.out"
-------------------------------------------------
ID:          #1.1
COMMAND:     ls | wc
SUBCOMMAND:  ls
DATE:        Lun 30 Apr 2018 10:57:04
OUTPUT:

shell

RETURN CODE: 0
```

```
------------------------------------------------
ID:          #1.2
COMMAND:     ls | wc
SUBCOMMAND:  wc
DATE:        Lun 30 Apr 2018 10:57:04
OUTPUT:

       1       1       6

RETURN CODE: 0
------------------------------------------------
bash>
```

dove si immagina che lanciando “ls | wc” si riescano a gestire singolarmente i due sottocomandi “ls” e “wc” di cui si riesce a riportare nel “log” i rispettivi output specifici. La riga “ID” è un ipotetico identificativo che il programma potrebbe creare per ogni comando eseguito.
Se non si è in grado di gestire “tutti” i comandi, eventuali non supportati non devono creare situazioni incontrollate, ma eventualmente fornire un avviso del tipo appunto “Comando non supportato”.
La gestione dell’output di errore (canale stderr) avviene analogamente a quello informativo (canale stdout visto sopra) tranne per il fatto che i dati sono salvati su un file che può essere differente (nota: è premiante essere in grado di gestire entrambi gli output su uno stesso file. In questo caso potrebbe esserci una voce aggiuntiva per indicare di quale output si tratta).

Statistiche comandi e analisi

Un esempio di esecuzione potrebbe essere:

```
bash>./run --logfile=”/tmp/run.log” --format=csv “ls | wc”
       1       1       6
bash>
```

da un punto di vista implementativo si dovrebbe comportare in maniera analoga a quanto indicato per il progetto “Shell custom” (→vedi) anche per quanto riguarda le implementazioni minime richieste, con alcune differenze principali:

- in questo caso non c’è un ambiente “interattivo”: il comando da eseguire è immesso direttamente passando anche gli eventuali argomenti a supporto
- il file di “log” è di default univoco e non contiene tipicamente l’output (la struttura può essere analoga a quella del progetto precedente, ma con campi diversi dove necessario), ma informazioni statistiche (oltre al comando eseguito ovviamente) come:
    - tempo di avvio, di conclusione e durata in secondi del comando
    - codice di ritorno
- come nell’altro caso è ben considerato essere in grado di gestire i comandi composti “spezzandoli” nel log anzichè computandoli come un comando unico

È premiante riuscire a gestire tutti i casi possibili, compresi i caratteri speciali (ad esempio di redirezionamento): anche qui - come nel progetto precedente - comandi non gestibili dovrebbero fornire un'opportuna indicazione all'utente.
Nell'esempio si immagina un argomento aggiuntivo "--format" che potrebbe consentire di decidere il layout dei dati nel file di log, ad esempio "txt" (analogamente a quanto ipotizzato nel progetto precedente, →vedi) oppure "csv" (salva i dati come record di un file csv). Similmente si potrebbero aggiungere argomenti per consentire personalizzazioni sulle informazioni "loggate".

Il programma DEVE generare un sotto-processo (o un albero) che abbia le funzioni proprie del "logger" e questo deve essere unico per ogni istante: se si lancia il programma da più shell contemporaneamente NON devono essere creati due "logger" separati, ma le istanze successive alla prima devono riconoscere la presenza del processo già attivo e inviare ad esso le informazioni richieste, così che si preoccupi di salvarle su file.

Gestore I/O 8+8 bit su raspberry

È premiante realizzare un modello hardware o implementare una soluzione di emulazione a livello software per la parte di interazione: bisogna tener conto che un minimo di supporto è comunque necessario per poter variare l'input e avere di conseguenza un feedback del comportamento del programma (una soluzione minimale in questo senso potrebbe essere un file "txt" esterno, magari da passare come argomento al programma, che elenchi in ogni riga una serie di azioni corrispondenti al prelievo delle uova dal cartone per l'input oltre a un analogo file di output).
La soluzione minimale deve essere in grado di "rispondere" alle variazioni dell'input integrando il cartone delle uova: l'ottimale è avere un buon sistema di feedback (quindi con input interattivo - tramite hardware o con una simulazione in cui ad esempio alcuni tasti della tastiera corrispondono alle posizioni del cartone delle uova e premerli indica che si stanno consumando le uova corrispondenti - e output interattivo, che mostri in tempo reale la situazione) magari rappresentando sia i "bit" che la situazione reale in forma più "grafica" (o semi-grafica: sempre su terminale comunque). Si valuta positivamente un buon algoritmo che riesce a tenere "pieno" il cartone con meno azioni possibili.

È richiesto che si generi un processo (o un albero) che faccia da "gestore" oltre a un processo (o albero) per ogni singolo bit, per cui devono esistere almeno 1 (principale) + 1 (gestore generale) + 1 (gestione input) + 8 (bit input) + 1 (gestione output) + 8 (bit output) processi, ossia 20 processi (come minimo). Quelli che fanno riferimento ai bit "leggono" e "scrivono" un singolo bit, comunicando lo stato poi al gestore di input o output e questi a loro volta al gestore generale che implementa la logica vera e propria.