PROGETTO STATISTICHE SHELL

FEATURES

BASE

- Esecuzione comandi
- Poter specificare file di log
- Parametri extra
- Logger indipendente
- Possibile concorrenza tra più shell
- Supporto di qualsiasi comando e carattere speciale
- Statistiche comandi
- Exit Code
- Makefile con regole essenziali

AVANZATE

- Suddivisione di comandi '`;`' e '`|`'
- Consistenza id e sub-id comandi
- Ogni sottocomando viene eseguito una sola volta
- Conservazione ambiente shell
- Esecuzione da qualsiasi posizione della shell
- Possibilità di modificare impostazioni
- Formattazione TXT e CSV

UTILIZZO

MAKE

- `make h`: mostra una lista di comandi eseguibili e macro personalizzabili
- `make b`: compila solo il necessario
- `make c`: cancella le cartelle `bin` e `temp`

ESECUZIONE

- eseguire un comando: è supportato qualsiasi comando, con qualsiasi carattere speciale

```
bash>./bin/run "ls | wc"
16 16 157
bash>
```

In assenza di `&&` e `||` il comando verrà spezzato in sottocomandi all'occorrenza di `|` e `;`.

- interrompere il logger

```
bash>./bin/run -k true
Logger killed
bash>
```

- cambiare uno o più settings (nessun comando viene eseguito se si cambiano impostazioni)

```
bash>./bin/run -maxOutput 50 --code=false
Settings will be applied on next logger loading
To restart logger use -k=true
bash>
```

- visualizzare gli argomenti possibili

```
bash>./bin/run -h
Settings updated, logger restarted
bash>
```

IMPLEMENTAZIONE

BASE

Esecuzione comandi

Dopo aver suddiviso i comandi in eventuali sotto-comandi, vengono create due pipe: una per inviare comandi alla shell (`toShell`), l'altra (`fromShell`) per riceverne `stdout` e `stderr`. Si effettua un `fork()`. Il child sostituisce `stdin` con `toShell[0]` e `stdout` e `stderr` con `fromShell[1]` (utilizzando `dup2(2)`). Esso eseguirà i comandi attraverso `execvp(2)`. La funzione `executeCommand(5)` può ora inviare comandi a tale child e ricevere i rispettivi output, scrivendo ogni informazione in un `Pk`. Tutti i `Pk` vengono concatenati in un unica stringa, preceduta da un intero che ne rappresenta la lunghezza. Il programma può ora inviare tale stringa al logger attraverso la fifo `logger.fifo`.

Poter specificare file di log

La variabile `logfile` all'interno dei settings viene letta all'avvio del logger. Può essere passata come parametro, vedi modifica impostazioni. Il file di log rimane lo stesso a prescindere dalla formattazione scelta, vedi formattazione logfile.

Parametri extra

Possono essere sfruttati per cambiare le impostazioni/settings. Vengono letti dalla funzione `readArgumets(4)`, poi analizzati da `evaluateCommand(3)`, infine valorizzati da `checkCommandBool(3)` e `checkCommandInt(3)`. Vengono di conseguenza cambiate alcune variabili. Se necessario, il logger viene riavviato.

Logger indipendente

All'avvio del programma, viene controllata l'esistenza del processo di log. Il PID del logger è salvato nel file `logPID.txt`. Attraverso `getpgid(1)` si ottiene lo stato del logger. Se necessario se ne crea uno nuovo effettuando un `fork()` ed eseguendo la funzione `logger(1)`.

Possibile concorrenza tra più shell

Ogni shell scrive in modo indipendente all'interno della fifo `logger.fifo`. Il logger legge un "pacchetto" di informazioni per volta grazie all `int` presente all'inizio di ogni pacchetto.

Supporto di qualsiasi comando e carattere speciale

I comandi vengono eseguiti attraverso `execvp("sh", "sh", "-c", " *comando* ")`, quindi vengono eseguiti correttamente a prescindere dalla loro complessità.

Statistiche comandi

La funzione `executeCommand(5)` ottiene statistiche riguardanti i tempi, oltre ad output ed exit code. Vedi executeCommand(5).

Exit Code

L'exit code si ottiene con l'esecuzione di un `echo $?` subito dopo il comando. Una volta ottenuto tale `int`, viene rimosso dall'output del comando.

Makefile con regole essenziali

Eseguire il makefile senza argomenti genererà una schermata che indica all'utente di usare come argomento h oppure help al fine di ottenere una lista di informazioni d'uso

I comandi utilizzabili dall'utente sono i seguenti:

- `make h`: stampa a schermo una lista di informazioni riguardanti i comandi utilizzabili e le macro modificabili a tempo di compilazione. Attraverso le macro disponibili è possibile modificare nomi di cartelle, eseguibili e file di progetto, inoltre si può scegliere il compilatore da usare.
- `make b`: quando invocato, l'eseguibile viene aggiornato attraverso il linking dei file oggetto se il suo timestamp mostra una data che succede ad almeno un di questi ultimi file.

  I file oggetto , a loro volta, vengono aggiornati attraverso la ricompilazione dei file sorgente con il file header solo se quest'ultimi risultano più recenti.

  il makefile non fa distinzione per nome, ma considera i file per estensione e, di conseguenza, compila e linka insieme tutti i file presenti nella cartella dei sorgenti.

  Alla creazione dei file, le cartelle di destinazione vengono create sul momento se necessario.

- `make c`: viene eseguita una pulizia completa della cartella contenente i file temporanei e di quella contenente il file eseguibile, nonchè la loro rimozione.

AVANZATE

Suddivisione di comandi

Viene controllata la presenza di `&&` oppure `||`. In tali casi, nessun comando viene spezzato. Pezzi di comando avvolti da parentesi sono sempre eseguiti in blocco. La divisione avviene in presenza di `|` e di `;`. Il programma spezza il comando in varie parti. Ne esegue una, cattura l'output, in caso di `|` lo inoltra per il sotto-comando successivo, prosegue al prossimo sotto-comando. Una volta completati tutti i sottocomandi, viene inviata al logger una stringa contenente tutte le informazioni ottenute.

Consistenza id e sub-id comandi

Ogni sotto-programma ottiene il suo sub-id all'interno del proprio Pk. Solo il logger ha accesso all'ID del comando completo. Tale ID è persistente tra le varie sessioni del logger perchè viene memorizzato nel file `idCount`.

Ogni sottocomando viene eseguito una sola volta

Per esempio, alla chiamata di `ls | wc` non vengono chiamati `ls` e poi `ls | wc`. Infatti, potendo mantenere attivo il processo di shell, viene eseguito `ls`. L'output di tale comando viene passato al `wc` attraverso un `echo " *risultato di ls* " | wc`, nel caso in cui il risultato di `ls` sia sullo `stdout`. Se l'output è su `stderr`, il pipe non passa nulla a `wc`, quindi chiamiamo `false | wc`.

Conservazione ambiente shell

Il child che esegue i comandi rimane attivo tra un sottocomando e l'altro, grazie alla possibilità di scrivergli comandi tramite pipe. In questo modo l'esecuzione di `cd /; pwd` avviene in due step: il child esegue `cd /`, rimane in attesa, riceve `pwd` in `stdin` e lo esegue.

Esecuzione da qualsiasi posizione della shell

Al momento della compilazione viene creata una macro (quale?) con la path assoluta del programma. In questo modo anche chiamandolo da fuori della cartella del progetto, i file temporanei si troveranno sempre nella cartella del progetto.

Modifica impostazioni

Le impostazioni modificabili dall'utente sono le seguenti:

- nome del file di log
- lunghezza massima dell'output da salvare su log
- salvataggio del codice di ritorno su log
- stile di formattazione del log

Formattazione file di log

La variabile `print style` all'interno dei settings definisce la formattazione, tra CSV e TXT. Il file di log rimane lo stesso, non cambia cioè l'estensione. Spetta all'utente definire una nuova destinazione con l'estensione desiderata.

UTILITY

Pack

Lo `struct Pack` oppure `Pk` contiene tutte le informazioni riguardo al comando eseguito:

- tempo di avvio
- tempo di fine
- tempo impiegato
- comando completo
- sotto-comando eseguito
- id del sotto-comando
- pid del mittente
- tipo di output (stdout o stderr)
- output
- return code

ExecuteCommand(5)

Questa funzione prende 5 parametri:

- un booleano per specificare se occorre effettuare un `|` o meno
- un Pk passato per riferimento. Qui finiscono le informazioni
- fd per scrivere al child-shell
- fd per ricevere output dal child-shell
- output del comando precedente (da usare in caso di `|`)

Dato un sotto-comando=*comando* e pid del padre=*pid*

- in caso di `|`:
  - in caso di output precedente non vuoto, output=*output*:
    `echo "*output*" | *comando*; echo $?; kill -10 *pid*`
  - in caso di output precedente vuoto:
    `echo false | *comando*; echo $?; kill -10 *pid*`
- in caso di `;`: `*comando*; echo $?; kill -10 *pid*`

Mentre il child-shell esegue tale comando, la funzione attende che il booleano `proceed` cambi valore a causa dell' arrivo del segnale 10, che simbolizza la terminazione del comando. Questo per evitare di leggere in modo parziale l'output della shell. Le varie informazioni vengono poi scritte all'interno del `Pk`.