μbash v2.2: laboratorio di Sistemi di Elaborazione e Trasmissione dell'Informazione (SETI)

a.a. 2023/2024

5 novembre 2023

Introduzione

Lo scopo di questo laboratorio è implementare, testandola adeguatamente, una piccola shell che chiameremo μbash, per prendere familiarità con le *system call* POSIX di base per la gestione dei processi.
Come nel precedente laboratorio *ping pong*, avrete a disposizione uno scheletro di implementazione, dove le parti di codice da completare sono delimitate da commenti del tipo:

```
/*** TO BE DONE START ***/
/*** TO BE DONE END ***/
```

e, tipicamente, precedute da una breve spiegazione che *assume che abbiate letto questo documento.*

Sintassi e funzionalità La sintassi dei comandi è stata pensata per facilitarne il "*parsing*" e non la comodità d'uso da parte degli utenti. Per esempio, richiediamo che non ci sia nessun *blank* fra il carattere `>` e il nome del file dove redirigere lo standard output di un comando, quindi `ls >foo` è un comando valido, mentre `ls > foo` non lo è (nella vera *bash* entrambe le forme sono equivalenti).
Inoltre, a differenza delle shell comunemente usate, non gestiremo l'espansione dei nomi di file, le sequenze di escape, le stringhe, i gruppi di processi, processi in *foreground*/*background* e tante altre cose.

Gestione degli errori Per la gestione degli errori usate l'approccio semplificato già visto in *ping-pong*: quando una funzione di libreria o system call fallisce, per ragioni indipendenti dall'input dell'utente, non cercate di recuperare la situazione ma uscite segnalando il problema. Ovvero, uscire dalla shell perché fallisce, per esempio, una `chdir(2)`, a causa di un errore di digitazione da parte dell'utente, sembra un po' eccessivo, mentre è ok in caso di fallimento di `fork(2)` o `malloc(3)`.

Librerie e strumenti Non dovrete utilizzare librerie di terze parti, ad eccezione della GNU Readline (già utilizzata nel `Makefile`). Per installarne la versione da sviluppatore, su Ubuntu o analogo sistema Debian-based, potete usare: `sudo apt install libreadline-dev`
Come dialetto del C assumiamo ISO 2011, con estensioni GNU (ovvero, `-std=gnu11`).

Debugging e gestione della memoria Per facilitare la risoluzione di problemi il `Makefile` compila con i simboli per il debug (`-ggdb`) e abilita solo ottimizzazioni compatibili con il debugging (`-Og`).
Ricordatevi di controllare sempre il valore di ritorno di ogni funzione/syscall e rilasciare immediatamente le risorse (per esempio, i *file descriptor*) quando non più necessarie. Fate particolare attenzione alla gestione della memoria dinamica: l'uso di strumenti come *address sanitizer* (cioè usare l'opzione `-fsanitize=address`) o *valgrind* (se, per qualche ragione, *address sanitizer* non fosse disponibile) è obbligatorio.
Non saranno considerate valide le consegne con ovvi problemi legati all'uso della memoria, dove "ovvi" significa: "si trovano subito con address-sanitizer e/o valgrind".
Come già detto più volte a lezione, *sviluppate/testate su una macchina (eventualmente virtuale) Linux*, perché negli a.a. precedenti abbiamo avuto comportamenti strani su alcuni Mac.

Descrizione di μbash

μbash processa i comandi, leggendoli da *standard input*, linea per linea, finché non raggiunge la fine del file (*ctrl-D* da terminale). Prima di leggere una linea, stampa un *prompt* che visualizza la directory corrente, vedete `getcwd(2)`, seguita dalla stringa `" $ "`. Di `getcwd` potete usare la versione di glibc (la GNU libc è la libreria C standard sotto Linux), che estende POSIX.1-2001[1].

[1] Scoprite da soli perché dovreste volerlo fare ☺

Come le shell "vere", μbash offre sia comandi *built-in* (ma, nel nostro caso, uno solo: `cd`), sia la possibilità di eseguire comandi/programmi esterni, passando argomenti e redirigendo I/O in file o *pipe*.
L'unico comando built-in è `cd`, che prende un solo argomento: il *pathname* della directory di destinazione (quindi, a differenza di quello in *bash*, non ci sono argomenti opzionali e non dovete modificare le variabili d'ambiente). Per semplicità, il comando `cd` può essere usato solo come primo e unico comando di una linea, senza nessuna redirezione dell'I/O (nel caso l'utente cerchi di usare redirezioni o usi `cd` in pipe con altri comandi, dovete segnalare un errore). Per esempio, sono comandi legali:

- `cd foo`
- `cd /non/importa/se/non/esiste`

mentre non lo sono:

- `cd foo >bar` — errore: redirezione con comando `cd`
- `cd /etc | grep pippo` — errore: `cd` usato con altri comandi

Per la sua implementazione, vedete `chdir(2)`.

Comandi esterni

Tutte le linee, non vuote, vengono suddivise in una sequenza di comandi separati dal carattere pipe (`|`): $l = c_1 \mid c_2 \mid \ldots \mid c_n$. Ovviamente, nel caso $n = 1$ non ci sarà nessun separatore. Il risultato del parsing di una linea sarà un oggetto di tipo `line_t`, composto, a sua volta, da n comandi di tipo `command_t`.
Dopo aver separato l in una sequenza di comandi, ogni comando c è, a sua volta, suddiviso in una sequenza di argomenti separati da blank (spazi o tab): $c = a_1\ a_2\ \ldots\ a_k$. A questo punto, se un certo a_j inizia con il carattere...

dollaro (`$`) allora a_j va sostituito con il valore della variabile d'ambiente corrispondente. Per esempio, se un argomento fosse `$foo`, andrebbe sostituito con il valore della variabile d'ambiente `foo`, si veda `getenv(3)`.

minore (`<`) allora a_j va tolto dalla lista degli argomenti e considerato una redirezione dello *standard input*. Per esempio, se un argomento fosse `<foo` (notare l'assenza di spazi fra `<` e `foo`), per l'esecuzione del comando corrispondente lo standard input dovrebbe corrispondere al file `foo`. È un errore specificare più di una redirezione dell'input per ogni comando. Per la redirezione vedete `open(2)`, `dup/dup2(2)` e `close(2)`.

maggiore (`>`) allora a_j va tolto dalla lista degli argomenti e considerato una redirezione dello *standard output*. Per esempio, se un argomento fosse `>foo`, per l'esecuzione del comando corrispondente lo standard output dovrebbe corrispondere al file `foo`. È un errore specificare più di una redirezione dell'output per ogni comando.

In una sequenza di comandi, solo il primo comando può redirigere lo standard input e solo l'ultimo comando può redirigere lo standard output (e nessuno è costretto a farlo). Ovviamente, se $n = 1$ il singolo comando può redirigere entrambi.
Per tutti i comandi da c_2 a c_n, lo standard input di c_i deve corrispondere allo standard output di c_{i-1}, si veda `pipe(2)`. Per impostare `FD_CLOEXEC` vedere la descrizione di `F_GETFD` e `F_SETFD` in `fcntl(2)`.
Dopo aver rimosso le redirezioni, si considerano gli argomenti rimanenti: $a'_1\ a'_2\ \ldots\ a'_x$. Deve essere $0 < x \leq k$, altrimenti, se $x = 0$, vuol dire che in c non è stato specificato nessun vero comando, ma solo redirezioni.
A questo punto, a'_1 è il nome del file da eseguire e $a'_2 \ldots\ a'_x$ i suoi argomenti. Ricordate che, per convenzione, `argv[0]`=a'_1, `argv[1]`=a'_2, etc.
Dopo aver eseguito i comandi specificati in una linea, vedere `exec(3)`, aspettate la terminazione di tutti i processi figli, vedere `wait(2)`, segnalando se un processo termina con uno *status* diverso da 0 (usare `WIFIEXITED` e `WEXITSTATUS`), oppure è stato ucciso da un segnale (usare `WIFSIGNALED` e `WTERMSIG`).

Esempi

Alcuni esempi di linee che la μbash deve poter eseguire:

- `cd foo` — cambia la directory di lavoro
- `ls -l | grep foo >bar` — filtra l'elenco dei file tenendo solo le linee che contengono la stringa "foo" e scrive il risultato nel file "bar"
- `cat /proc/cpuinfo | grep processor | wc -l` — conta il numero di processori (core) presenti nel sistema
- `cat </proc/cpuinfo | grep processor | wc -l` — come il precedente, ma stavolta `cat` legge da standard input (che è stato rediretto)

E alcuni esempi di linee sbagliate (si deve segnalare un errore di "parsing"):

- `cd foo bar` — errore: il comando `cd` ha un solo argomento
- `cd foo <bar` — errore: il comando `cd` non supporta la redirezione
- `ls | cd foo` — errore: il comando `cd` deve essere usato da solo
- `ls -l | grep foo > bar` — errore: non è specificato il file per la redirezione dello standard output (c'è uno spazio fra `>` e `bar`)
- `ls | grep foo <bar | wc -l` — errore: solo il primo comando può avere la redirezione dell'input

Testing

Oltre a provare gli esempi elencati sopra, pensate ad altri casi di test, sia con esito positivo (cioè i comandi vengono eseguiti e producono l'output atteso), sia con esito negativo. Ovvero, verificate che l'implementazione si comporti bene anche nei casi di errore; cioè, non "esploda" ma segnali precisamente la condizione di errore all'utente. Alcune situazioni da verificare includono, per esempio:

- i file specificati non esistono
- i file esistono ma non sono leggibili/scrivibili
- le variabili di ambiente usate non sono presenti

Elencate, in un file PDF (o ASCII o markdown, ma evitate formati Office-like), i modi in cui avete testato la vostra implementazione. Per ogni test effettuato indicate:

1. *scopo*: cosa/quali parti di codice volete testare in questo caso
2. *situazione iniziale*: per esempio, se l'ambiente deve contenere/non-contenere particolari variabili; la directory corrente (se significativa), ...
3. *linea inviata alla microbash*
4. *risultato atteso*

Per esempio:

- test variabile di ambiente non esistente
  - *scopo*: verificare che la shell "espanda" le variabili di ambiente non esistenti in stringhe vuote
  - *situazione iniziale*: un ambiente in cui la variabile `XYZ` non esiste
  - *linea inviata alla microbash*: `echo a $XYZ b`
  - *risultato atteso*: "`a  b`"; si noti il doppio spazio fra `a` e `b`
- test variabile di ambiente esistente
  - *scopo*: verificare che la shell espanda correttamente le variabili di ambiente (definite)
  - *situazione iniziale*: ambiente in cui `XYZ` è uguale a `pippo`
  - *linea inviata alla microbash*: `echo a $XYZ b`
  - *risultato atteso*: `a pippo b`

L'idea é che per ogni riga di codice che scrivete, ci sia almeno un caso di test che va ad eseguirla.