Costa Davide, Forneris Luca, Ivaldi Emiliano

BUSH - -

Relazione

Lo scopo del secondo laboratorio assegnatoci consiste nell'implementazione di una versione più rudimentale e primitiva della shell presente nei sistemi Unix-like.

Per realizzare ciò abbiamo lavorato su alcune parti dei file, che fra poco elencheremo, mentre i rimanenti file, che contengono altre funzioni più complicate che ancora non possiamo svolgere con le nostre conoscenze, sono stati precedentemente completati dai professori.

I file, e le relative funzioni da implementare, che abbiamo dovuto modificare sono stati:

- ***bmm.c*** – contenente una funzione da scrivere ex novo, la **make_sure_PWD_is_set()**: questa si occupa della creazione e dell'inizializzazione, al valore restituito dalla system call *getcwd()*, della variabile PWD. *getcwd()* ritorna infatti un array con al suo interno una stringa rappresentate la cartella di lavoro corrente (current working directory).
- ***var_table.c*** – anch'esso file con un'unica funzione da modificare, ossia la **vt_to_envp()**. Tale funzione si occupa della creazione dell'array contenente le variabili di ambiente e dell'opportuna allocazione di memoria tramite la funzione *my_malloc()* (variante di *malloc*). Utilizzando il framework suggerito dai professori (*Valgrind*) abbiamo notato un errore di allocazione alla riga 62, dopo aver modificato e testato il codice più volte ci siamo limitati al metodo che ci sembrava migliore senza però riuscire a risolvere completamente il problema.
- ***shell.c*** – contenente due funzioni da svolgere:

  free_envp() e **wait_for_termination_of_children()**.

  La prima delle due funzioni è banale: richiama la funzione *free()* per deallocare memoria riservata all'ambiente envp: per fare ciò basta un ciclo *for*, per scorrere tutto l'array, e la chiamata di *free()*.

  La seconda, invece, si basa sulla chiamata della system call *wait()* per controllare il valore di terminazione di tutti i processi precedentemente lanciati. Finchè il pid ritornato dalla *wait()* è diverso da 0, ovvero finchè ci sono altri child in esecuzione, continuo ad aspettare. Controllo successivamente il valore di ritorno: se non è zero, oppure se il child è stato terminato senza richiamare exit né return, allora segnalo un errore.
- ***ast.c*** – questo file, a differenza dei file precedenti, contiene ben cinque funzioni da implementare:

  cd_execute(), **find_in_path()**, **redirect_fd()**, **ext_cmd_execute()** e **pipe execute().**

  La prima si occupa dell'inizalizzazione del comando cd, o *change directory*. Per fare ciò controllo il valore di *impl* (se uguale a zero restituisco HOME direttamente),

chiamo la system call *chdir()* e aggiorno i valori di PWD e OLDPWD rispettivamente tramite la *vt_set_value()* e *vt_lookup()*.

La seconda funzione, ovvero la *find_in_path()*, è forse la funzione più complicata dell'intero laboratorio: Se *name* contiene degli *'/'* (PATH_SEPARATOR) ne restituisco una copia con *strdup* (in quanto mi aspetto che il percorso sia nel nome), altrimenti continuo a scansionare path. Al fine di evitare allocazioni e deallocazioni continue, salvo nella variabile allocated la grandezza della sottostringa più grande allocata. Se la lunghezza della nuova stringa è maggiore di allocated la rialloco. Il nome allocated è leggermente fuorviante in quanto non sono le celle realmente allocate ma solamente le celle occupate dalla *sub_path* senza il nome dell'eseguibile. Copio quindi in currpath la stringa che va da start a start+len con una *memcpy()* e con una *sprintf()* concateno *'/'* con name.Se il percorso così creato punta ad un eseguibile (*access()* restituisce 0 se il file è eseguibile) allora alloco una variabile della lunghezza esatta della sub_path e con una *memcpy()* la copio. Dopo di che libero la memoria di currPath. Non restituisco subito currPath in quanto come detto prima potrebbe occupare più memoria di quella di cui ha realmente bisogno la stringa.

redirect_fd() chiama la funzione di sistema *dup2()* per duplicare il filedescriptor new_fd in old_fd (se from_fd non è NO_REDIR).

Nella parte da implementare di *ext_cmd_execute()* si richiede di creare un processo figlio chiamando la system call *fork*. Assegno poi a pid il suo valore di ritorno, e se la syscall fallisce invoco fail_errno. Successivamente, per lanciare l'eseguibile, uso *execve()* (sempre segnalando errori nel caso la syscall fallisca) passando tra gli argomenti anche una chiamata alla funzione *vt_to_envp()* creata precedentemente.

Infine la funzione *pipe_execute()* si occupa dell'inizializzazione di pipes[]. Per fare ciò chiamo *pipe()* e *fcntl()* per settare il File Descriptor in modo che venga chiuso automaticamente dopo l'esecuzione. Per far sì che l'output del comando *left_cmd* sia rediretto su pipes[1] e che l'input del comando *right_cmd* sia rediretto su pipes[0] eseguo il nodo sinistro passandogli *in_redir* come input e come output pipes[1], poi eseguo il destro, passando come input pipes[0] e come output *out_redir*.

Se pur rimandata, la data di scadenza ci ha messo fretta nell'ultima parte del laboratorio ma con qualche pomeriggio passato in università e discutendone con i nostri compagni siamo riusciti a completare tutte le parti richieste.