Massimo Mantovani 5186259
Matteo Attolini 4677905
Dionis Nikolla 5196050

# Testing con GDB

Nota: con “risultato atteso” intendiamo l’esito che ci aspettiamo da un comando e il risultato effettivamente ottenuto dal lancio del comando. Se il risultato effettivamente ottenuto fosse diverso dal risultato atteso verrà aggiunta la nota: “risultato ottenuto diverso dal risultato atteso”.

Nota: il valore di ritorno di alcune funzioni è stato ottenuto con il comando di gdb “p/u $eax”.
In questi casi dopo il valore scriveremo (p/u $eax)

Linea inviata alla microbash in questa serie di test:
**“cat </proc/cpuinfo | grep processor | wc -l >output”**

## Funzione ”free_command” dalla riga 93 alla 106:

Consideriamo il caso i = 0

- test riga 98 **“free(c→args[i]);”**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale*: c→args[i] = “cat”
  - *risultato atteso:* l’area di memoria occupata dalla variabile c→args[i] viene liberata

- test riga 100 **“free(c→args);”**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* c→args “punta” ad un puntatore che “punta” ad un’area di memoria che è stata liberata con la precedente free
  - *risultato atteso:* l’area di memoria puntata direttamente da c→args (cioè quella che rappresenta il “secondo puntatore”) viene liberata

- test riga 101 **“free(c→out_pathname);”**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* c→out_pathname vale “null” perché nel caso i = 0 stiamo analizzando il primo comando di una serie di comandi collegati da pipe. Quindi questo comando non può redirezionare l’output.
  - *risultato atteso:* in questo caso questa free non fa niente perché è chiamata su un puntatore a null. La free chiamata su un puntatore a null rimane un operazione sicura che non ha bisogno di controlli

- test riga 102 **“free(c→in_pathname);”**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* c→in_pathname vale “/proc/cpuinfo”
  - *risultato atteso:* l’area di memoria “puntata” dal puntatore c→in_pathname viene liberata

- test riga 103 **“free(c);”**
  - *scopo:* verifica istruzione specificata

- *situazione iniziale:* c è un puntatore ad una struct che contiene dei puntatori, ai quali nelle istruzioni precedenti sono state chiamate le free, che hanno liberato le aree di memoria da essi “puntate”
- *risultato atteso:* viene liberata l’area di memoria puntata dal puntatore c

## Funzione ”free_line” dalla riga 108 alla 119:

Consideriamo il caso i = 0

- test riga 114 **“free_command(l→commands[i]);”**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* l è un puntatore ad una struct che contiene due elementi: il numero di comandi presenti nella linea e un array (doppio puntatore) alle struct che contengono le informazioni di ogni singolo comando. Verifichiamo il contenuto di l→commands stampando per esempio l→commands[0]→args[0] otteniamo "cat"
  - *risultato atteso:* viene liberata l’area di memoria puntata dal puntatore l→commands. Infatti se cerchiamo di stampare nuovamente il contenuto di l→commands[0]→args[0] GDB restituisce l’errore:
    <error: Cannot access memory at address 0x19800001>

- test riga 116 **“free(l→commands);”**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* l→commands è un array (doppio puntatore) di puntatori a struct che rappresentano i singoli comandi della linea. Nel precedente ciclo sono state liberate le aree di memoria “puntate” dai puntatori “interni”
  - *risultato atteso:* viene liberata l’area di memoria puntata dal puntatore l→commands

- test riga 117 **“free(l);”**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* nelle istruzioni precedenti sono state liberate le aree di memoria “puntate” dai puntatori interni di l
  - *risultato atteso:* viene liberata l’area di memoria puntata dal puntatore l

Linea inviata alla microbash in questa serie di test:
**“echo $SHELL”**

## Funzione ”parse_cmd” dalla riga 150 alla 214:

- test riga 195 **“char *aux = my_strdup(&tmp[1]);”**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* aux è una variabile ausiliaria
  - *risultato atteso:* aux “punta” ad una nuova stringa che ha il valore di tmp privato del primo carattere ($)

- test riga 196 **“tmp = getenv(aux);”**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* il risultato dell’istruzione precedente, tmp contiene il valore "$SHELL"
  - *risultato atteso:* tmp contiene il valore "/bin/bash"

- test riga 197 **“free(aux);”**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* aux contiene il valore “SHELL”
  - *risultato atteso:* viene liberata l’area di memoria “puntata” dal puntatore aux

Linea inviata alla microbash in questo test:
**“echo $XYX”**

- test righe 198 e 199
  - *scopo:* verificare le due istruzioni che sono eseguite solo nel caso di variabile di ambiente non esistente
  - *situazione iniziale:* dopo l’istruzione alla riga 196 tmp ha valore “null”
  - *risultato atteso:* dopo queste due istruzioni tmp ha valore “” (stringa vuota)

Linea inviata alla microbash in questa serie di test:
**“cat </proc/cpuinfo | grep processor | wc -l >output”**

## Funzione ”check_redirections” dalla riga 240 alla 279:

Consideriamo il caso i = 0

- test del check redirezione dell’input
  - *scopo:* verificare che il programma esegua correttamente il controllo della redirezione dell’input
  - *situazione iniziale:* le strutture dati che rappresentano la linea e i singoli comandi sono già state inizializzate
  - *risultato atteso:* il programma riconosce che la variabile c→in_pathname non è “vuota” e che il comando associato è il primo

Consideriamo il caso i = 2

- test del check redirezione dell’output
  - *scopo:* verificare che il programma esegua correttamente il controllo della redirezione dell’output
  - situazione iniziale: le strutture dati che rappresentano la linea e i singoli comandi sono già state inizializzate
  - *risultato atteso:* il programma riconosce che la variabile c→out_pathname non è “vuota” e che il comando associato è l’ultimo

## Funzione ”check_cd” dalla riga 281 alla 316:

- questa funzione è già stata testata in un altra serie di test in un altro pdf

Linea inviata alla microbash in questa serie di test:
**“cat </proc/cpuinfo | grep processor | wc -l >output”**

## Funzione ”wait_for_children” dalla riga 322 alla 344:

- test riga 333 **“if (wait(&status) == -1)”**

- *scopo:* verifica istruzione specificata
- *situazione iniziale:* nella funzione che chiama la wait_for_children è stata eseguita una fork per redirigere l'input e l'output
- *risultato atteso:* la syscall wait ritorna come valore l'ID del processo figlio terminato

- test riga 337 **"if (WIFEXITED(status) && WEXITSTATUS(status) != 0)"**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* è stata eseguita la wait nell'istruzione precedente
  - *risultato atteso:* il flusso del programma non entra nell'if. Questo significa che il processo figlio è terminato correttamente. In queste due istruzioni che compongono l'if il controllo viene effettuato controllando il valore di ritorno del processo figlio

- test riga 340 **"if (WIFSIGNALED(status) && WTERMSIG(status) != 0)"**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* uguale alla precedente istruzione
  - *risultato atteso:* il flusso del programma non entra nell'if. Questo significa che il processo figlio è terminato correttamente. In queste due istruzioni che compongono l'if il controllo viene effettuato controllando se il processo figlio è stato terminato da un segnale oppure da un errore di sistema

Linea inviata alla microbash in questa serie di test:
**"cat </proc/cpuinfo | grep processor | wc -l >output"**

## Funzione "redirect" dalla riga 346 alla 364:

Consideriamo la redirezione del comando "cat"

- test riga 353 **"if (from_fd != NO_REDIR)"**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* from_fd = 3 e fa riferimento al file "/proc/cpuinfo"
    "NO_REDIR" = -1 è una costante ausiliaria che indica che non c'è redirezione nel comando
  - *risultato atteso:* il flusso del programma entra nell'if

- test riga 355 **"if (dup2(from_fd, to_fd) < 0)"**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* from_fd come nell'istruzione precedente. to_fd = 0 e fa riferimento allo standard input
  - *risultato atteso:* la dup2 crea una copia del file descriptor from_fd in to_fd e ritorna il valore 0 (p/u $eax) che è il valore del nuovo file descriptor

- test riga 357 **"if (close(from_fd) < 0)"**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* niente di particolare
  - *risultato atteso:* l'istruzione close chiude il file descriptor from_fd e restituisce 0 (p/u $eax) senza errori

Linea inviata alla microbash in questa serie di test:
**"cat </proc/cpuinfo | grep processor | wc -l >output"**

Funzione ”run_child” dalla riga 366 alla 388:

- test riga 377 **“pid_t pid = fork();”**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* niente di particolare
  - *risultato atteso:* la fork crea un nuovo processo, duplicando il processo chiamante

- test riga 379 **“if (pid == 0)”**
  - *scopo:* verifica valore ritornato dalla fork
  - *situazione iniziale:* è stata eseguita la fork nell’istruzione precedente
  - *risultato atteso:* la fork ritorna 0 nel processo figlio

- test riga 381 **“redirect(c_stdin, 0);”**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* c_stdin è il filedescriptor su cui deve essere rediretto lo standard input
  - *risultato atteso:* lo standard input viene rediretto sul file descriptor c_stdin

- test riga 382 **“redirect(c_stdout, 1);”**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* c_stdout è il filedescriptor su cui deve essere rediretto lo standard output
  - *risultato atteso:* lo standard output viene rediretto sul file descriptor c_stdout

- test riga 383 **“if (execvp(c→args[0], c→args) < 0)”**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* c→args[0] è il comando da eseguire, c→args i suoi argomenti
  - *risultato atteso:* il flusso del programma non entra nell’if e il processo figlio termina, questo significa che la exec è stata eseguita correttamente

Linea inviata alla microbash in questo test:
**“cd ..”**

Funzione ”change_current_directory” dalla riga 390 alla 407:

- test riga 397 **“if (chdir(newdir) < 0)”**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* newdir è la stringa il cui valore rappresenta la directory in cui dobbiamo cambiare la current working directory
  - *risultato atteso:* l’istruzione chdir cambia la current working directory in newdir e restituisce 0 (p/u $eax) senza errori

Linea inviata alla microbash in questa serie di test:
**“cat </proc/cpuinfo | grep processor | wc -l >output”**

Funzione “execute_line” dalla riga 417 alla 474:

- test riga 437 **“curr_stdin = open(c→in_pathname, O_RDONLY);”**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* c→in_pathname contiene il valore “/proc/cpuinfo” e dobbiamo aprire un file descriptor associato a questo file

  - *risultato atteso:* la open ritorna il valore 3 (p/u $eax) che è il nuovo file descriptor appena aperto

- test riga 450 **“curr_stdout = open(c→out_pathname, O_CREAT | O_RDWR | O_TRUNC, 00666);”**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* c→out_pathname contiene il valore “output” e dobbiamo aprire un file descriptor associato a questo file
  - *risultato atteso:* la open ritorna il valore 4 (p/u $eax) che è il nuovo file descriptor appena aperto

- test riga 462 **“if (pipe2(fds, O_CLOEXEC) < 0)”**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* dobbiamo aprire una pipe
  - *risultato atteso:* viene aperta la pipe senza errori

## Funzione “main” dalla riga 490 alla 514

- test riga 502 **“pwd = getcwd(NULL, 0);”**
  - *scopo:* verifica istruzione specificata
  - *situazione iniziale:* niente di particolare
  - *risultato atteso:* pwd contiene la stringa che rappresenta la current working directory