

MicroBash

Giovanni Pio Antonuccio, Filippo Baldini, Giacomo Cerlesi

Corso di laurea in Informatica L-31, Unige DIBRIS

MicroBash 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, seguita dalla stringa “ \$ ”.

Come le *shell* “vere”, *MicroBash* offre sia comandi *build-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*.

Ora procediamo con un'attenta analisi del nostro codice e i metodi di debug associati:

Funzioni `free_command`, `free_line`:

Come si intuisce dal loro nome sono utilizzate per liberare rispettivamente lo struct che definisce ciascun comando e ogni linea di comandi (diversi struct di comandi associati fra di loro dal contatore `n_commands`). Per queste due funzioni non abbiamo implementato metodi di debug specifici poiché il loro corretto funzionamento è controllato implicitamente eseguendo il debugging delle altre funzioni.

Funzione `parse_cmd`:

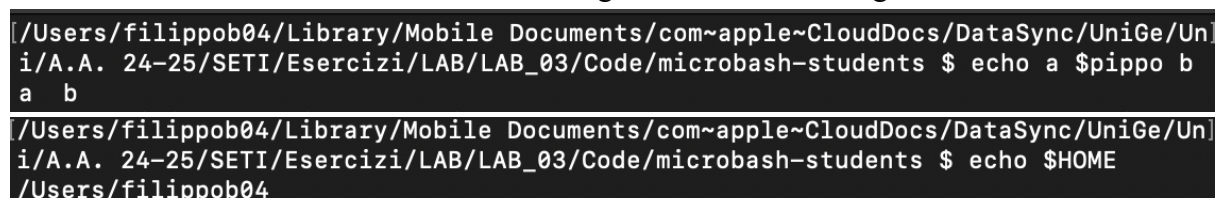
Questa funzione come si può facilmente dedurre dal nome interpreta ciascun comando, verifica la presenza di simboli come le ridirezioni dell'*output/input* (<, >) o la gestione di *variabili d'ambiente* (\$). In particolare, era necessario completare la sezione del *parsing* delle *variabili d'ambiente*.



```
if (*tmp=='$') {
    if (!tmp[1]) {
        fprintf(stderr, "Parsing error: no argument specified after $ symbol\n");
        goto fail;
    }
    char *env_value = getenv(tmp + 1);
    if (env_value) {
        tmp = env_value;
    } else {
        tmp = "";
    }
}
```

La nostra funzione prima esegue un controllo della sintassi del comando e, successivamente, utilizzando la `getenv()` restituisce, se presente, la variabile d'ambiente associata.

In questo caso per eseguire il debugging ci siamo affidati ai comandi descritti nel .pdf fornito; quindi, verificando il funzionamento sia nel caso che la variabile *pippo* fosse presente nell'ambiente che non. Scrivendo “\$” senza argomenti otteniamo il giusto errore.



```
/Users/filippob04/Library/Mobile Documents/com~apple~CloudDocs/DataSync/UniGe/Un
i/A.A. 24-25/SETI/Esercizi/LAB/LAB_03/Code/microbash-students $ echo a $pippo b
a b
/Users/filippob04/Library/Mobile Documents/com~apple~CloudDocs/DataSync/UniGe/Un
i/A.A. 24-25/SETI/Esercizi/LAB/LAB_03/Code/microbash-students $ echo $HOME
/Users/filippob04
```

Funzione check_rederictions:

Questa funzione ha il compito di verificare che le *I/O rederictions*, come da progetto, avvengano solo all'inizio (*input*) o alla fine (*output*) di ciascun comando.

```

for (int i = 0; i < l->n_commands; ++i) {
    command_t *cmd = l->commands[i];

    if (cmd->in_pathname != NULL && i != 0){
        fprintf(stderr, "Error: Input redirection is only allowed for the first
        command\n");
        return CHECK_FAILED;
    }
    if (cmd->out_pathname != NULL && i != l->n_commands - 1){
        fprintf(stderr, "Error: Output redirection is only allowed for the last
        command\n");
        return CHECK_FAILED;
    }
}
}

```

Anche in questo caso il debugging risulta essere molto semplice, i controlli funzionano correttamente e otteniamo il risultato desiderato sia in *input* che *output*.

```

/home/JackCeres/Desktop/UNI24/SETI/microbash-students-svolto/microbash-students
$ ls | grep foo <bar | wc -l
Error: Input redirection is only allowed for the first command

```

In questo caso il controllo è identico per il caso *output* (>)

Funzione check_cd:

Lo scopo di questa funzione è quello di verificare la corretta sintassi dell'unico comando *built-in* della *MicroBash*, il *cd*.

```

for (int i = 0; i < l->n_commands; ++i) {
    command_t *cmd = l->commands[i];

    if (strcmp(cmd->args[0], CD) == 0){
        if(l->n_commands > 1){
            fprintf(stderr, "cd must be the only command of the line\n");
            return CHECK_FAILED;
        }
        if(cmd->n_args != 2){
            fprintf(stderr, "cd must have only one argument\n");
            return CHECK_FAILED;
        }
        if(cmd->in_pathname || cmd->out_pathname){
            fprintf(stderr, "cd cannot have I/O rederictions\n");
            return CHECK_FAILED;
        }
    }
}
}

```

Anche in questo caso il debugging risulta essere molto semplice, abbiamo utilizzato i vari esempi proposti nel .pdf, i controlli funzionano correttamente e otteniamo il risultato desiderato.

```
/home/JackCeres/Desktop/UNI24/SETI/microbash-students-svolto/microbash-students
$ cd
cd must have only one argument
/home/JackCeres/Desktop/UNI24/SETI/microbash-students-svolto/microbash-students
$ cd echo
chdir to newdir failed, - cd error
/home/JackCeres/Desktop/UNI24/SETI/microbash-students-svolto/microbash-students
$ cd .. | touch test.txt
cd must be the only command of the line
/home/JackCeres/Desktop/UNI24/SETI/microbash-students-svolto/microbash-students
$ cd foo <bar
cd cannot have I/O redirections
```

Verifichiamo così le condizioni del comando *built-in* `cd` e la `change_current_directory`. Utilizzando `cd` senza alcun argomento (o con più di un argomento), con un argomento diverso da una *directory* e in *pipeline* e con le *I/O* redirections.

Associata alla funzione `check_cd` c'è la funzione **`change_current_directory`**, che ha lo scopo di sostituire la directory corrente con una nuova.

```
if(chdir(newdir) == -1){  
    fprintf(stderr, "chdir to newdir failed - cd error\n");  
}
```

Nel nostro caso se la `cd` fallisce (ad esempio per un errore di *typing* dell'utente), piuttosto che uscire dalla *MicroBash* con una `EXIT`, stampiamo un errore e riproponiamo una nuova *command line* vuota.

Funzione **redirect**:

Questa funzione è essenziale per il corretto funzionamento della *MicroBash* e la sua gestione dell' *I/O*. Infatti, la funzione Verifica se è presente un file da ridirezionare, a quel punto duplica il contenuto da `from_fd` a `to_fd` e chiude il *file descriptor* originale per liberare memoria.

```
if(from_fd != NO_REDIR){  
    if (dup2(from_fd, to_fd) == -1) {  
        fatal_errno("dup2 failed\n");  
    }  
    if (close(from_fd) == -1) {  
        fatal_errno("close failed\n");  
    }  
}
```

Funzione `wait_for_children`:

Questa funzione, assieme alla `run_child`, è essenziale per il corretto funzionamento della *MicroBash*. La `wait_for_children` gestisce ogni processo *child* creato e verifica la causa della sua terminazione, se da parte di un *segnale* e con che codice è stato terminato. Come suggerito nel .pdf utilizziamo la `WIFSIGNALED`, `WTERMSIG` e `WEXITSTATUS` per verificare ciò.

```
int status;
pid_t pid;


while ((pid = waitpid(-1, &status, 0)) > 0) {
    if (WIFEXITED(status)) {
        int exit_status = WEXITSTATUS(status);
        if (exit_status != 0) {
            fprintf(stderr, "Child process %d exited with status %d\n", pid,
                exit_status);
        }
    } else if (WIFSIGNALED(status)) {
        int sig_num = WTERMSIG(status);
        fprintf(stderr, "Child process %d was killed by signal %d (%s)\n", pid,
            sig_num, strsignal(sig_num));
    }
}

if (pid == -1 && errno != ECHILD) {
    fatal_errno("waitpid failed\n");
}
```

La funzione opera correttamente e svolge le sue operazioni come attese.

Funzione run_child:

Questa funzione è responsabile nel creare i *child* e gestirne *input* e *output*.



```
pid_t pid = fork();

if(pid == -1){
    fatal_errno("fork failed\n");
}
if (pid == 0){
    if (c_stdin != NO_REDIR){
        if (dup2(c_stdin, STDIN_FILENO) == -1){
            fatal_errno("dup2 failed for stdin\n");
        }
        close(c_stdin);
    }

    if (c_stdout != NO_REDIR){
        if (dup2(c_stdout, STDOUT_FILENO) == -1){
            fatal_errno("dup2 failed for stdout\n");
        }
        close(c_stdout);
    }

    execvp(c->args[0], c->args);
    fatal_errno("execvp failed\n");
}
```

Debuggare questa funzione risulta piuttosto difficile perché dovremmo causare il fallimento di una `fork()` o della creazione dello specifico *stdin* o *stdout*.

Funzione `execute_line` e gestione dell'I/O:

Nella funzione `execute_line` abbiamo due sezioni da completare, una per l'Input e una dedicata all'Output.

```
int fd = open(c->in_pathname, O_RDONLY);
if(fd == -1){
    fprintf(stderr, "input file open failed\n");
}
curr_stdin = fd;
```

```
int fd = open(c->out_pathname, O_WRONLY | O_CREAT | O_TRUNC, 0644);
if(fd == -1){
    fprintf(stderr, "output file open failed\n");
}
curr_stdout = fd;
```

In caso *input* apriamo il file in modalità *read-only*, l'*output* risulta essere leggermente più complesso in quanto dobbiamo considerare, oltre al caso di *write* semplice, anche una eventuale *sovrascrittura* o *creazione* stessa del file. Per questo motivo inseriamo anche **0644**, notazione in *ottale* che specifica i *permessi del file appena creato*.

Nello specifico $6 = 4 + 2$, permessi *rw* (*read*, *write*) dati all'*owner* del file, 4, permessi solo di lettura al *gruppo* e agli altri *utenti* esterni al gruppo, quindi sarà *rw-r-r*.

Nel debugging abbiamo notato che inserendo il comando:

~ touch nomefile.txt, e poi controllando con *stat* la *bitmask* del file viene alterata a 0664. con *echo* o *cat* la *bitmask* non viene modificata e resta 0644.

Ecco alcuni esempi:

Sostituiamo 0644 con 0000 (Assenza di permessi) e creiamo il file con *echo >filename*

```
/home/JackCeres/Desktop/UNI24/SETI/microbash-students-svolto/microbash-students
$ echo >file000.txt
/home/JackCeres/Desktop/UNI24/SETI/microbash-students-svolto/microbash-students
$ stat file000.txt
  File: file000.txt
  Size: 1          Blocks: 8          IO Block: 4096   regular file
Device: 8,2      Inode: 1048801    Links: 1
Access: (0000/-----)  Uid: ( 1000/JackCeres)   Gid: ( 1000/JackCeres)
Access: 2024-12-05 15:37:39.113256097 +0000
Modify: 2024-12-05 15:37:39.120256098 +0000
Change: 2024-12-05 15:37:39.120256098 +0000
Birth: 2024-12-05 15:37:39.113256097 +0000
```

Il comando *stat* restituisce correttamente la *bitmask* Access: (0000/-----)

Svolgendo la medesima operazione con il comando `touch` otteniamo invece:

```
/home/JackCeres/Desktop/UNI24/SETI/microbash-students-svolto/microbash-students
$ touch file001.txt
/home/JackCeres/Desktop/UNI24/SETI/microbash-students-svolto/microbash-students
$ stat file001.txt
  File: file001.txt
  Size: 0          Blocks: 0          IO Block: 4096   regular empty file
Device: 8,2      Inode: 1058559      Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/JackCeres)   Gid: ( 1000/JackCeres)
Access: 2024-12-05 15:39:19.047264265 +0000
Modify: 2024-12-05 15:39:19.047264265 +0000
Change: 2024-12-05 15:39:19.047264265 +0000
Birth: 2024-12-05 15:39:19.047264265 +0000
```

La *bitmask* viene alterata dal comando `touch`.

infatti cercando `touch` nel manuale è espressamente specificato che:

“If any file does not exist, it is created with default permissions”

Con i comandi `echo` e `cat` abbiamo verificato anche il corretto funzionamento della redirectione di *Input* e *Output*. Abbiamo verificato inoltre cosa succede in caso il file da aprire non abbia i permessi di *read* o *write*, in quel caso con `echo` otteniamo l'errore adeguato (Utilizziamo la `fprintf` su *standard error*), invece con `cat` riceviamo correttamente l'errore ma per terminare l'inserimento dobbiamo dare un `ctrl-D`.

Successivamente modifichiamo le proprietà dei fd interessati nella pipeline con:

```
if (pipe(fds) == -1) {
    fatal_errno("pipe creation failed\n");
}
if (fcntl(fds[0], F_SETFD, FD_CLOEXEC) == -1 || fcntl(fds[1], F_SETFD, FD_CLOEXEC)
== -1) {
    fatal_errno("fcntl failed to set FD_CLOEXEC\n");
}
```

Utilizzando sempre dei comandi suggeriti nel .pdf

Nella funzione **main** invece abbiamo solo una sezione marcata da completare

```
const int max_size = 512;

pwd = my_malloc(max_size + 1);
if(getcwd(pwd, max_size + 1) == NULL){
    fatal_errno("getcwd error\n");
}
```

allochiamo dello spazio ad un *puntatore* `pwd` per il salvataggio della current *work directory*.
Se la `malloc` fallisce la funzione termina con una `EXIT` e il messaggio appropriato.

Test con Valgrind:

```
JackCeres@VirtualeMachine:~/Desktop/Aux$ cc -Wall -fsanitize=address -o test mic
robash.c -lreadline
JackCeres@VirtualeMachine:~/Desktop/Aux$ valgrind test
==6537== Memcheck, a memory error detector
==6537== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
==6537== Using Valgrind-3.23.0 and LibVEX; rerun with -h for copyright info
==6537== Command: test
==6537==
==6537==
==6537== HEAP SUMMARY:
==6537==   in use at exit: 0 bytes in 0 blocks
==6537== total heap usage: 30 allocs, 30 frees, 3,993 bytes allocated
==6537==
==6537== All heap blocks were freed -- no leaks are possible
==6537==
==6537== For lists of detected and suppressed errors, rerun with: -s
==6537== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
JackCeres@VirtualeMachine:~/Desktop/Aux$ test
JackCeres@VirtualeMachine:~/Desktop/Aux$
```

Considerando un debugging ad ampio spettro, abbiamo provato ad eseguire una *MicroBash* lanciandola all'interno di *MicroBash* stessa, verificando i PID possiamo confermare il successo della sua creazione.

```
/home/JackCeres/Desktop/UNI24/SETI/microbash-students-svolto/microbash-students
$ ps
  PID TTY          TIME CMD
  4120 pts/0    00:00:00 bash
  6275 pts/0    00:00:00 microbash
  6281 pts/0    00:00:00 ps
/home/JackCeres/Desktop/UNI24/SETI/microbash-students-svolto/microbash-students
$ ./microbash
/home/JackCeres/Desktop/UNI24/SETI/microbash-students-svolto/microbash-students
$ ps
  PID TTY          TIME CMD
  4120 pts/0    00:00:00 bash
  6275 pts/0    00:00:00 microbash
  6283 pts/0    00:00:00 microbash
  6284 pts/0    00:00:00 ps
```

Allegiamo anche alcuni dei comandi descritti nel .pdf:

```
/home/JackCeres/Desktop/UNI24/SETI/microbash-students-svolto/microbash-students
$ cd foo
chdir to newdir failed, - cd error
```

Cerchiamo di accedere a una directory inesistente

```
/home/JackCeres/Desktop/UNI24/SETI/microbash-students-svolto/microbash-students
$ ls -l | grep foo >bar
Child process 3952 exited with status 1
```

Cerchiamo un file che contenga la parola foo (Non presente)

```
/home/JackCeres/Desktop/UNI24/SETI/microbash-students-svolto/microbash-students
$ cat /proc/cpuinfo | grep processor | wc -l
3
```

Questo dimostra il funzionamento corretto dei comandi in *pipeline*.

```
/home/JackCeres/Desktop/UNI24/SETI/microbash-students-svolto/microbash-students
$ cd foo <bar
cd cannot have I/O redirections
```

```
/home/JackCeres/Desktop/UNI24/SETI/microbash-students-svolto/microbash-students
$ ls | cd foo
cd must be the only command of the line
```

```
/home/JackCeres/Desktop/UNI24/SETI/microbash-students-svolto/microbash-students
$ ls -l | grep foo > bar
Parsing error: no path specified for output redirection
```

Così verifichiamo che inserendo uno spazio fra > e bar la *MicroBash* non trova il *path*.

```
/home/JackCeres/Desktop/UNI24/SETI/microbash-students-svolto/microbash-students
$ ls | grep foo <bar | wc -l
Error: Input redirection is only allowed for the first command
```