Αναφορά Εργαστηριακής Άσκησης

Στην παρούσα εργαστηριακή άσκηση, υλοποιείται σε τελικό στόχο ένα μινιμαλιστικό-απλοποιημένο κέλυφος. Το τελικό αποτέλεσμα της εργασίας απαρτίζει ένα κέλυφος ικανό να δεχθεί βασικές εντολές του unix και να τυπώσει στην οθόνη τα αποτελέσματα αυτών. Οι βασικές εντολές χωρίζονται σε απλές εντολές, την εντολή cd, εντολές με ένα ή και περισσότερα pipes και τέλος την εδική εντολή exit.

Αναλυτικότερα, το πρόγραμμα αποτελεί μια μέθοδο η οποία διαβάζει συνεχώς το stream της εισόδου του χρήστη από το πληκτρολόγιο και διαχειρίζεται ανάλογα με το είδος των παραπάνω εντολών την εκάστοτε περίπτωση. Στην περίπτωση της εντολής exit γίνεται και τερματισμός του προγράμματος. Στην περίπτωση μιας απλής εντολής υπάρχει διαχωρισμός μονάχα για την εντολή cd η όποια και εκτελείται βάσει της συνάρτησης chdir ενώ σε κάθε άλλη περίπτωση κάνουμε χρήση της εντολής execvp.

Τέλος στην περίπτωση όπου υπάρχουν εντολές με ένα ή περισσότερα pipes, γίνεται χρήση των συναρτήσεων pipe και dup2, καθώς και ενδιαμέσων file descriptors. Το παραπάνω έχει ως στόχο την επικοινωνία των in/out των ενδιαμέσων εντολών με στόχο την ανακατεύθυνση του κάθε stream της εκάστοτε εντολής. Ο κώδικας αναφοράς του τελικού shell (mysh4.c) καθώς και των ενδιαμέσων είναι ο ακόλουθος.

```
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <string.h>
#include <termios.h>

#define FIXCHARS " \t\r\n\a"
#define NONCHARS "\r\n"
#define BUFSIZE 64

int count_pipe (char **args) {   int
arg = 0;   while (args[arg]!=NULL) {
if ( strcmp( args[arg],"|" ) == 0 )
return 1;     arg++;   }   return 0;
}
char **fix_arg_table( char *command ){
int bufsize = BUFSIZE,     position =
0;   char **buffer;   char *arg;
  buffer = malloc( BUFSIZE * sizeof(char*) ); /* Allocation */
if( buffer == NULL ) exit(0); /* Error @ malloc */    arg =
strtok( command, FIXCHARS );      while ( arg != NULL )
{     buffer[position] = arg;     position++;
    if ( position >= bufsize )
{       bufsize += BUFSIZE;
      buffer = realloc( buffer, bufsize * sizeof(char*) ); /*
ReAllocate the buffer */
      if( buffer == NULL ) exit(0); /* Error @ realloc */
    }
    arg = strtok(NULL, FIXCHARS);
  }
  buffer[position] = NULL;
return buffer;
```

```
}

void do_pipe_command( char ** args ) {
  // File descriptors
  int filedes[2]; // pos. 0 output, pos. 1 input of the pipe
int filedes2[2];

  int pipeCommands = 1;

  char *buffer; /* Stores the user input */   size_t
bufsize = BUFSIZE*BUFSIZE; /* Sizeof Buffer */   char
**args2;   pid_t pid;

  /* Allocate buffer */
  buffer = (char *)malloc(bufsize * sizeof(char));
if( buffer == NULL ) {
      fprintf(stderr, "malloc(): Unable to allocate buffer.!!\n");
exit(0);
  }    int err
= -1;   int end
= 0;

  /* Iterating commands */
int i = 0;   int j = 0;
int k = 0;   int l = 0;

  /* Count total commands (pipes+1) */
while ( args[i] != NULL ){     if
(strcmp(args[i],"|") == 0)
{       pipeCommands++;
    }
i++;
  }
  i = 0; /* Clear iterator */

  /* Parse and execute all commands
   * Determine the input or output file descriptor of each pipe */
while ( args[j] != NULL && end != 1 ) {     k = 0; /* Parse every
command */
    memset( buffer,0,strlen( buffer ) ); /* Clear command buffer */
while ( strcmp(args[j],"|" ) != 0 ) { /* Stop at the pipe */
strcat( buffer, args[j] );       j++;
      if (args[j] == NULL) { /* Parsed the whole command */
end = 1;         break;
      }
      k++;
      strcat( buffer, " " ); /* Add the missing whitespace back */
    }
    args2 = fix_arg_table( buffer ); /* Slpit command */
j++; /* Skip Pipe */
           if ( i % 2
!=         0         )
pipe(    filedes    );
else
      pipe( filedes2 );

    pid = fork();
```

```
    if ( pid == (pid_t)-1 ) { /* Error forking */
if ( i != pipeCommands - 1 ) {          /* Close
open file descriptors */          if (i % 2 != 0)
close(filedes[1]);          else
          close(filedes2[1]);
       }
       fprintf( stderr, "fork(): %s.\n", strerror(errno) );
exit(0);      }
    if( !pid ) { /* Child process */

       if ( i == 0 ) /* The first's command input fd is STDOUT */
dup2( filedes2[1] , STDOUT_FILENO );
       else if ( i == pipeCommands - 1 ) { /* The lasts's command fd
is STDIN */
         if (pipeCommands % 2 != 0) // for odd number of commands
dup2(filedes[0],STDIN_FILENO);          else // for even number
of commands           dup2(filedes2[0],STDIN_FILENO);
        }         else { //
for odd i           if (i %
2 != 0) {
          dup2(filedes2[0],STDIN_FILENO);
dup2(filedes[1],STDOUT_FILENO);
         }
         else { // for even i
dup2(filedes[0],STDIN_FILENO);
dup2(filedes2[1],STDOUT_FILENO);
         }
}
       if ( execvp( args2[0], args2 ) == -1 ) {
fprintf(stderr, "execvp(): %s.\n", strerror(errno));
kill( getpid(),SIGTERM );
       }         }      if ( i
== 0 )
{         close( filedes2[1]
);
    }
    else if ( i == pipeCommands - 1 ) {
if    (pipeCommands    %    2    !=    0)
close( filedes[0] );        else
         close( filedes2[0] );
    }
else {
       if (i % 2 != 0) {
         close(filedes2[0]);
close(filedes[1]);
       }        else{
close(filedes[0]);
close(filedes2[1]);
       }
}
    waitpid(pid,NULL,0);
i++;
  }
}
void exec_cd ( char **argv ) { chdir(argv[1]); }

void exec_command ( char **buffer ) {
pid_t child;    int status;    child =
```

```
fork();      if(child  ==  (pid_t)-1)
exit(0);    if  (!child)  /* Child */
execvp( buffer[0], buffer );   else
    waitpid (child, &status, 0);
}
char *command_reader(void) {
char *buffer;
  size_t bufsize = BUFSIZE, helper;
buffer = (char *)malloc(bufsize*sizeof(char));
if( buffer == NULL) exit(0);
  helper = getline(&buffer, &bufsize, stdin); /* Read stdin. */
if ( helper > (ssize_t)0 ) {
    if ( strcspn( buffer, NONCHARS ) > (size_t)0 ) /* Terminate input
command before the newline. */
      buffer[strcspn( buffer, NONCHARS )] = '\0';
return buffer;
  }    return
NULL;
}
int main(int argc,char **argv) {
   char  *buffer;  /*  Buffer  to  store  input  command  */      char
**args;   /* Stores the input command plus the arguments */    int
terminate  =  1;     do  {      printf("$  ");      fflush(stdout);
buffer=command_reader();     if (buffer!= NULL) {
      if ( !strcmp( buffer, "exit\0" ) )
break; /* Exit */       else {
            args=fix_arg_table(buffer);
if    (strcmp(buffer,"cd\0")    ==    0)
exec_cd( args );         else {
          if ( count_pipe( args ) )
do_pipe_command(args); /* Do pipe */           else
            exec_command( args ); /* Simple command */
        }
      }     }
free(buffer);   }
while (1);
}
```