/\*

\* Part of the solution for Assignment 2

\*/

//#define DEBUG

#include <stdio.h>

#include <libgen.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <string.h>

#include "tokenize.h"

#include "tcp-utils.h" // for `readline' only

/\*

\* Global configuration variables.

\*/

const char\* path[] = {"/bin","/usr/bin",0}; // path, null terminated (static)

const char\* prompt = "sshell> "; // prompt

const char\* config = "shconfig"; // configuration file

/\*

\* Typical reaper.

\*/

void zombie\_reaper (int signal) {

int ignore;

while (waitpid(-1, &ignore, WNOHANG) >= 0)

/\* NOP \*/;

}

/\*

\* Dummy reaper, set as signal handler in those cases when we want

\* really to wait for children. Used by run\_it().

\*

\* Note: we do need a function (that does nothing) for all of this, we

\* cannot just use SIG\_IGN since this is not guaranteed to work

\* according to the POSIX standard on the matter.

\*/

void block\_zombie\_reaper (int signal) {

/\* NOP \*/

}

/\*

\* run\_it(c, a, e, p) executes the command c with arguments a and

\* within environment p just like execve. In addition, run\_it also

\* awaits for the completion of c and returns a status integer (which

\* has the same structure as the integer returned by waitpid). If c

\* cannot be executed, then run\_it attempts to prefix c successively

\* with the values in p (the path, null terminated) and execute the

\* result. The function stops at the first match, and so it executes

\* at most one external command.

\*/

int run\_it (const char\* command, char\* const argv[], char\* const envp[], const char\*\* path) {

// we really want to wait for children so we inhibit the normal

// handling of SIGCHLD

signal(SIGCHLD, block\_zombie\_reaper);

int childp = fork();

int status = 0;

if (childp == 0) { // child does execve

#ifdef DEBUG

fprintf(stderr, "%s: attempting to run (%s)", \_\_FILE\_\_, command);

for (int i = 1; argv[i] != 0; i++)

fprintf(stderr, " (%s)", argv[i]);

fprintf(stderr, "\n");

#endif

execve(command, argv, envp); // attempt to execute with no path prefix...

for (size\_t i = 0; path[i] != 0; i++) { // ... then try the path prefixes

char\* cp = new char [strlen(path[i]) + strlen(command) + 2];

sprintf(cp, "%s/%s", path[i], command);

#ifdef DEBUG

fprintf(stderr, "%s: attempting to run (%s)", \_\_FILE\_\_, cp);

for (int i = 1; argv[i] != 0; i++)

fprintf(stderr, " (%s)", argv[i]);

fprintf(stderr, "\n");

#endif

execve(cp, argv, envp);

delete [] cp;

}

// If we get here then all execve calls failed and errno is set

char\* message = new char [strlen(command)+10];

sprintf(message, "exec %s", command);

perror(message);

delete [] message;

exit(errno); // crucial to exit so that the function does not

// return twice!

}

else { // parent just waits for child completion

waitpid(childp, &status, 0);

// we restore the signal handler now that our baby answered

signal(SIGCHLD, zombie\_reaper);

return status;

}

}

/\*

\* Implements the internal command `more'. In addition to the file

\* whose content is to be displayed, the function also receives the

\* terminal dimensions.

\*/

void do\_more(const char\* filename, const size\_t hsize, const size\_t vsize) {

const size\_t maxline = hsize + 256;

char\* line = new char [maxline + 1];

line[maxline] = '\0';

// Print some header (useful when we more more than one file)

printf("--- more: %s ---\n", filename);

int fd = open(filename, O\_RDONLY);

if (fd < 0) {

sprintf(line, "more: %s", filename);

perror(line);

delete [] line;

return;

}

// main loop

while(1) {

for (size\_t i = 0; i < vsize; i++) {

int n = readline(fd, line, maxline);

if (n < 0) {

if (n != recv\_nodata) { // error condition

sprintf(line, "more: %s", filename);

perror(line);

}

// End of file

close(fd);

delete [] line;

return;

}

line[hsize] = '\0'; // trim longer lines

printf("%s\n", line);

}

// next page...

printf(":");

fflush(stdout);

fgets(line, 10, stdin);

if (line[0] != ' ') {

close(fd);

delete [] line;

return;

}

}

delete [] line;

}

int main (int argc, char\*\* argv, char\*\* envp) {

size\_t hsize = 0, vsize = 0; // terminal dimensions, read from

char\* rhost = 0; //machine name

char\* rport = 0; //port num

// the config file

char command[129]; // buffer for commands

command[128] = '\0';

char\* com\_tok[129]; // buffer for the tokenized commands

size\_t num\_tok; // number of tokens

printf("Simple shell with client version v1.0.\n");

// Config:

int confd = open(config, O\_RDONLY);

if (confd < 0) {

perror("config");

fprintf(stderr, "config: cannot open the configuration file.\n");

fprintf(stderr, "config: will now attempt to create one.\n");

confd = open(config, O\_WRONLY|O\_CREAT|O\_APPEND, S\_IRUSR|S\_IWUSR);

if (confd < 0) {

// cannot open the file, giving up.

perror(config);

fprintf(stderr, "config: giving up...\n");

return 1;

}

close(confd);

// re-open the file read-only

confd = open(config, O\_RDONLY);

if (confd < 0) {

// cannot open the file, we'll probably never reach this

// one but who knows...

perror(config);

fprintf(stderr, "config: giving up...\n");

return 1;

}

}

// read terminal size

while (hsize == 0 || vsize == 0 ) {

int n = readline(confd, command, 128);

if (n == recv\_nodata)

break;

if (n < 0) {

sprintf(command, "config: %s", config);

perror(command);

break;

}

num\_tok = str\_tokenize(command, com\_tok, strlen(command));

// note: VSIZE and HSIZE can be present in any order in the

// configuration file, so we keep reading it until the

// dimensions are set (or there are no more lines to read)

if (strcmp(com\_tok[0], "VSIZE") == 0 && atol(com\_tok[1]) > 0)

vsize = atol(com\_tok[1]);

else if (strcmp(com\_tok[0], "HSIZE") == 0 && atol(com\_tok[1]) > 0)

hsize = atol(com\_tok[1]);

// lines that do not make sense are hereby ignored

}

close(confd);

if (hsize <= 0) {

// invalid horizontal size, will use defaults (and write them

// in the configuration file)

hsize = 75;

confd = open(config, O\_WRONLY|O\_CREAT|O\_APPEND, S\_IRUSR|S\_IWUSR);

write(confd, "HSIZE 75\n", strlen( "HSIZE 75\n"));

close(confd);

fprintf(stderr, "%s: cannot obtain a valid horizontal terminal size, will use the default\n",

config);

}

if (vsize <= 0) {

// invalid vertical size, will use defaults (and write them in

// the configuration file)

vsize = 40;

confd = open(config, O\_WRONLY|O\_CREAT|O\_APPEND, S\_IRUSR|S\_IWUSR);

write(confd, "VSIZE 40\n", strlen( "VSIZE 40\n"));

close(confd);

fprintf(stderr, "%s: cannot obtain a valid vertical terminal size, will use the default\n",

config);

}

printf("Terminal set to %ux%u.\n", (unsigned int)hsize, (unsigned int)vsize);

// install the typical signal handler for zombie cleanup

// (we will inhibit it later when we need a different behaviour,

// see run\_it)

signal(SIGCHLD, zombie\_reaper);

// Command loop:

while(1) {

// Read input:

printf("%s",prompt);

fflush(stdin);

if (fgets(command, 128, stdin) == 0) {

// EOF, will quit

printf("\nBye\n");

return 0;

}

// fgets includes the newline in the buffer, get rid of it

if(strlen(command) > 0 && command[strlen(command) - 1] == '\n')

command[strlen(command) - 1] = '\0';

// Tokenize input:

char\*\* real\_com = com\_tok; // may need to skip the first

// token, so we use a pointer to

// access the tokenized command

num\_tok = str\_tokenize(command, real\_com, strlen(command));

real\_com[num\_tok] = 0; // null termination for execve

int bg = 0;

int local = 0;

if (strcmp(com\_tok[0], "!") == 0){ //local commands

#ifdef DEBUG

fprintf(stderr, "%s: local command\n", \_\_FILE\_\_);

#endif

local = 1;

//discard the first token which means local access

real\_com = com\_tok + 1;

}

if (strcmp(real\_com[0], "&") == 0) { // background command coming

#ifdef DEBUG

fprintf(stderr, "%s: background command\n", \_\_FILE\_\_);

#endif

bg = 1;

// discard the first token now that we know that it

// specifies a background process...

real\_com = com\_tok + 2;

}

// ASSERT: num\_tok > 0

// Process input:

if (strlen(real\_com[0]) == 0) // no command, luser just pressed return

continue;

if (local == 0 ){ // TCP client stuff -- do not forget local ==0 bg =1

printf("do something client\n");

continue;

}

else{ //local commands

if (strcmp(real\_com[0], "exit") == 0) {

printf("Bye\n");

return 0;

}

else if (strcmp(real\_com[0], "more") == 0) {

// note: more never goes into background (any prefixing

// `&' is ignored)

if (real\_com[1] == 0)

printf("more: too few arguments\n");

// list all the files given in the command line arguments

for (size\_t i = 1; real\_com[i] != 0; i++)

do\_more(real\_com[i], hsize, vsize);

}

else { // external command

if (bg) { // background command, we fork a process that

// awaits for its completion

int bgp = fork();

if (bgp == 0) { // child executing the command

int r = run\_it(real\_com[0], real\_com, envp, path);

printf("& %s done (%d)\n", real\_com[0], WEXITSTATUS(r));

if (r != 0) {

printf("& %s completed with a non-null exit code\n", real\_com[0]);

}

return 0;

}

else // parent goes ahead and accepts the next command

continue;

}

else { // foreground command, we execute it and wait for completion

int r = run\_it(real\_com[0], real\_com, envp, path);

if (r != 0) {

printf("%s completed with a non-null exit code (%d)\n", real\_com[0], WEXITSTATUS(r));

}

}

}

}

}

}