Homework 7

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/wait.h>

#include <string.h>

#include <sys/stat.h>

#include <fcntl.h>

#define STDIN 0

#define STDOUT 1

#define MAXLINE 200 /\* This is how we declare constants in C \*/

#define MAXARGS 20

char \*file\_input;

char \*file\_output;

char \*child1 = NULL;

char \*child2 = NULL;

/\* In C, "static" means not visible outside of file. This is different

\* from the usage of "static" in Java.

\* Note that end\_ptr is an output parameter.

\*/

static char \* getword(char \* begin, char \*\*end\_ptr) {

char \* end = begin;

while ( \*begin == ' ' )

begin++; /\* Get rid of leading spaces. \*/

end = begin;

while ( \*end != '\0' && \*end != '\n' && \*end != ' ' && \*end != '#' )

end++; /\* Keep going. \*/

if ( end == begin )

return NULL; /\* if no more words, return NULL \*/

\*end = '\0'; /\* else put string terminator at end of this word. \*/

\*end\_ptr = end;

if (begin[0] == '$') { /\* if this is a variable to be expanded \*/

begin = getenv(begin+1); /\* begin+1, to skip past '$' \*/

if (begin == NULL) {

perror("getenv");

begin = "UNDEFINED";

}

}

return begin; /\* This word is now a null-terminated string. return it. \*/

}

/\* In C, "int" is used instead of "bool", and "0" means false, any

\* non-zero number (traditionally "1") means true.

\*/

/\* argc is \_count\_ of args (\*argcp == argc); argv is array of arg \_values\_\*/

static void getargs(char cmd[], int \*argcp, char \*argv[])

{

char \*cmdp = cmd;

char \*end;

int i = 0;

file\_input = NULL;

file\_output = NULL;

child1 = NULL;

child2 = NULL;

/\* fgets creates null-terminated string. stdin is pre-defined C constant

\* for standard intput. feof(stdin) tests for file:end-of-file.

\*/

if (fgets(cmd, MAXLINE, stdin) == NULL && feof(stdin)) {

printf("Couldn't read from standard input. End of file? Exiting ...\n");

exit(1); /\* any non-zero value for exit means failure. \*/

}

while ( (cmdp = getword(cmdp, &end)) != NULL ){

if(\*cmdp == '>'){

cmdp = end + 1;

file\_output = cmdp;

} else if (\*cmdp == '<'){

cmdp = end + 1;

file\_input = cmdp;

} else if(\*cmdp == '|'){

child1 = argv[i-1];

cmdp = end + 1;

child2 = cmdp;

} else {

/\* end is output param \*/

/\* getword converts word into null-terminated string \*/

argv[i++] = cmdp;

/\* "end" brings us only to the '\0' at end of string \*/

cmdp = end + 1;

file\_input = NULL;

file\_output = NULL;

}

}

argv[i] = NULL; /\* Create additional null word at end for safety. \*/

\*argcp = i;

}

void interrupt\_handler(int signum){

pid\_t childpid;

printf(" (^C recognized, killing child process)\n");

kill(childpid, SIGINT);

}

int mane(){

int pipe\_fd[2]; /\* 'man pipe' says its arg is this type \*/

int fd; /\* 'man dup' says its arg is this type \*/

pid\_t child1, child2; /\* 'man fork' says it returns type 'pid\_t' \*/

char \* argvChild[2];

printf("Executing \"ls | wc\";\n");

printf("Number of files in curr dir is (first # is answer): ");

fflush(stdout); /\* Force printing to complete, before continuing. \*/

if ( -1 == pipe(pipe\_fd) ) perror("pipe");

child1 = fork();

/\* child1 > 0 implies that we're still the parent. \*/

if (child1 > 0) child2 = fork();

if (child1 == 0) { /\* if we are child1, do: "ls | ..." \*/

if ( -1 == close(STDOUT) ) perror("close"); /\* close \*/

fd = dup(pipe\_fd[1]); /\* set up empty STDOUT to be pipe\_fd[1] \*/

if ( -1 == fd ) perror("dup");

if ( fd != STDOUT ) fprintf(stderr, "Pipe output not at STDOUT.\n");

close(pipe\_fd[0]); /\* never used by child1 \*/

close(pipe\_fd[1]); /\* not needed any more \*/

argvChild[0] = "ls"; argvChild[1] = NULL;

if ( -1 == execvp(argvChild[0], argvChild) ) perror("execvp");

} else if (child2 == 0) { /\* if we are child2, do: "... | wc" \*/

if ( -1 == close(STDIN) ) perror("close"); /\* close \*/

fd = dup(pipe\_fd[0]); /\* set up empty STDIN to be pipe\_fd[0] \*/

if ( -1 == fd ) perror("dup");

if ( fd != STDIN ) fprintf(stderr, "Pipe input not at STDIN.\n");

close(pipe\_fd[0]); /\* not needed any more \*/

close(pipe\_fd[1]); /\* never used by child2 \*/

argvChild[0] = "wc"; argvChild[1] = NULL;

if ( -1 == execvp(argvChild[0], argvChild) ) perror("execvp");

} else { /\* else we're parent \*/

int status;

/\* Close parent copy of pipes;

\* In particular, if pipe\_fd[1] not closed, child2 will hang

\* forever waiting since parent could also write to pipe\_fd[1]

\*/

close(pipe\_fd[0]);

close(pipe\_fd[1]);

if ( -1 == waitpid(child1, &status, 0) ) perror("waitpid");

/\* Optionally, check return status. This is what main() returns. \*/

if (WIFEXITED(status) == 0)

printf("child1 returned w/ error code %d\n", WEXITSTATUS(status));

if ( -1 == waitpid(child2, &status, 0) ) perror("waitpid");

/\* Optionally, check return status. This is what main() returns. \*/

if (WIFEXITED(status) == 0)

printf("child2 returned w/ error code %d\n", WEXITSTATUS(status));

}

return 0; /\* returning 0 from main() means success. \*/

}

static void execute(int argc, char \*argv[]){

pid\_t childpid; /\* child process ID \*/

signal(SIGINT, interrupt\_handler);

if(child1 !=NULL){

mane();

return;

}

childpid = fork();

if (childpid == -1) { /\* in parent (returned error) \*/

perror("fork"); /\* perror => print error string of last system call \*/

printf(" (failed to execute command)\n");

}

if (childpid == 0) { /\* child: in child, childpid was set to 0 \*/

/\* Executes command in argv[0]; It searches for that file in

\* the directories specified by the environment variable PATH.

\*/

if (file\_input !=NULL){

close(STDIN\_FILENO);

int fd = open(file\_input, O\_RDONLY);

if (fd == -1) perror("open for reading");

} else

if (file\_output !=NULL){

close(STDOUT\_FILENO);

int fd = open(file\_output, O\_WRONLY | O\_CREAT, S\_IRUSR | S\_IWUSR);

if (fd == -1) perror("open for writing");

}

if (-1 == execvp(argv[0], argv)) {

perror("execvp");

printf(" (couldn't find command)\n");

}

/\* NOT REACHED unless error occurred \*/

exit(1);

} else /\* parent: in parent, childpid was set to pid of child process \*/

waitpid(childpid, NULL, 0); /\* wait until child process finishes \*/

return;

}

int main(int argc, char \*argv[])

{

char cmd[MAXLINE];

char \*childargv[MAXARGS];

int childargc;

if(argc > 1){

freopen(argv[1], "r", stdin);

}

while (1) {

printf("%% "); /\* printf uses %d, %s, %x, etc. See 'man 3 printf' \*/

fflush(stdout); /\* flush from output buffer to terminal itself \*/

getargs(cmd, &childargc, childargv); /\* childargc and childargv are

output args; on input they have garbage, but getargs sets them. \*/

/\* Check first for built-in commands. \*/

if ( childargc > 0 && strcmp(childargv[0], "exit") == 0 )

exit(0);

else if ( childargc > 0 && strcmp(childargv[0], "logout") == 0 )

exit(0);

else

execute(childargc, childargv);

}

}