# Sistemas Operativos

## Mendez-Simó

## Lab Shell

## Entrega Parte 1, 2 y 3

## 04/05

**Hernán Tain**

## builtin.c

```
#include "builtin.h"

// returns true if the 'exit' call
// should be performed
int exit_shell(char* cmd) {

      if (strcmp(cmd,"exit") == 0)
            return true;

      return 0;
}

// returns true if "chdir" was performed
// this means that if 'cmd' contains:
//    $ cd directory (change to 'directory')
//    $ cd (change to HOME)
// it has to be executed and then return true
int cd(char* cmd) {

      if(strlen(cmd) == 2){
            if(strcmp(cmd,"cd") == 0){
                  int ret = chdir(getenv("HOME"));
                  if (ret == 0) return true;
                  else return 0;
            }
            else return 0;
      }


      if((cmd[0] == 'c') && (cmd[1] == 'd')){
            char* splited = split_line(cmd,' ');
            if (chdir(splited) == 0) return true;
            return 0;
      }

      return 0;
}

// returns true if 'pwd' was invoked
// in the command line
int pwd(char* cmd) {

      if (strcmp(cmd,"pwd") == 0){
            printf("%s\n",get_current_dir_name());
            return true;
      }
      return 0;
}
```

## createcmd.c

```
#include "createcmd.h"

// creates an execcmd struct to store
// the args and environ vars of the command
struct cmd* exec_cmd_create(char* buf_cmd) {

      struct execcmd* e;

      e = (struct execcmd*)calloc(sizeof(*e), sizeof(*e));

      e->type = EXEC;
      strcpy(e->scmd, buf_cmd);

      return (struct cmd*)e;
}

// creates a backcmd struct to store the
// background command to be executed
struct cmd* back_cmd_create(struct cmd* c) {

      struct backcmd* b;

      b = (struct backcmd*)calloc(sizeof(*b), sizeof(*b));

      b->type = BACK;
      strcpy(b->scmd, c->scmd);
      b->c = c;

      return (struct cmd*)b;
}

// encapsulates two commands into one pipe struct
struct cmd* pipe_cmd_create(struct cmd* left, struct cmd* right) {

      if (!right)
            return left;

      struct pipecmd* p;

      p = (struct pipecmd*)calloc(sizeof(*p), sizeof(*p));

      p->type = PIPE;
      p->leftcmd = left;
      p->rightcmd = right;

      return (struct cmd*)p;
}
```

## exec.c

```
#include "exec.h"

// sets the "key" argument with the key part of
// the "arg" argument and null-terminates it
static void get_environ_key(char* arg, char* key) {

      int i;
      for (i = 0; arg[i] != '='; i++)
            key[i] = arg[i];

      key[i] = END_STRING;
}

// sets the "value" argument with the value part of
// the "arg" argument and null-terminates it
static void get_environ_value(char* arg, char* value, int idx) {

      int i, j;
      for (i = (idx + 1), j = 0; i < strlen(arg); i++, j++)
            value[j] = arg[i];

      value[j] = END_STRING;
}

// sets the environment variables passed
// in the command line
//
// Hints:
// - use 'block_contains()' to
//    get the index where the '=' is
// - 'get_environ_*()' can be useful here
static void set_environ_vars(char** eargv, int eargc) {

      if (eargc == 0)
            return;

      int idx,status;
      for(int i = 0; i < eargc; i++){
            idx = block_contains( eargv[i],'=' );
            char* key = malloc( sizeof(char) * ( idx + 1 ));
            get_environ_key( eargv[i], key );
            char* value = split_line( eargv[i], '=' );

            status = setenv( key, value, 1 );

            free(key);
      }
}

// opens the file in which the stdin/stdout or
// stderr flow will be redirected, and returns
// the file descriptor
//
// Find out what permissions it needs.
// Does it have to be closed after the execve(2) call?
//
// Hints:
// - if O_CREAT is used, add S_IWUSR and S_IRUSR
```

```
//    to make it a readable normal file
static int open_redir_fd(char* file) {

      int fd = open(file, O_CREAT | O_RDWR, 0664 );
      if (fd < 0 ){
            return -1;
      }
      return fd;
}

// executes a command - does not return
//
// Hint:
// - check how the 'cmd' structs are defined
//    in types.h
void exec_cmd(struct cmd* cmd) {

      int fd_in,fd_out,fd_err;
      struct pipecmd* p;
      struct backcmd* b;
      struct execcmd* c,*r;
      switch (cmd->type) {

            case EXEC: {
                  c = (struct execcmd*) cmd;

                  set_environ_vars(c->eargv,c->eargc);
                  if(c->argc == 0)
                        break;

                  if((execvp(c->argv[0], c->argv)) < 0)
                        _exit(-1);

                  break;
            }

            case BACK: {
                  b = (struct backcmd* ) cmd;
                  exec_cmd(b->c);
                  _exit(-1);
                  break;
            }

            case REDIR: {

                  r = (struct execcmd*) cmd;

                  if((fd_out = open_redir_fd(r->out_file)) > 0){
                        dup2(fd_out, 1);
                        close(fd_out);
                  }

                  if((fd_in = open_redir_fd(r->in_file)) > 0) {
                        dup2(fd_in, 0);
                        close(fd_in);
                  }

                  if((fd_err = open_redir_fd(r->err_file)) > 0) {
                        dup2(fd_err, 2);
                        close(fd_err);
```

```
                }

                if ((fd_out < 0) && (fd_in < 0) && (fd_err < 0))
                        _exit(-1);

                execvp(r->argv[0],r->argv);
                break;
        }

        case PIPE: {

                int pipefd[2];

                p = (struct pipecmd*) cmd;

                if ( (pipe(pipefd)) < 0)
                        _exit(-1);

                c = (struct execcmd*) p->leftcmd;
                r = (struct execcmd*) p->rightcmd;

                if (fork() == 0) {
                        if((dup2(pipefd[1], 1)) < 0 )
                                _exit(-1);

                        close(pipefd[1]);

                        execvp(c->argv[0],c->argv);
        }

                if (fork() == 0) {
                        if ((dup2(pipefd[0], 0)) < 0)
                                _exit(-1);

                        close(pipefd[0]);
                        execvp(r->argv[0],r->argv);
        }

                // free the memory allocated
                // for the pipe tree structure
                free_command(parsed_pipe);

                break;
        }
    }
}
```

## freecmd.c

```
#include "freecmd.h"

// frees the memory allocated
// for the tree structure command
void free_command(struct cmd* cmd) {

      int i;
      struct pipecmd* p;
      struct execcmd* e;
      struct backcmd* b;

      if (cmd->type == PIPE) {

            p = (struct pipecmd*)cmd;

            free_command(p->leftcmd);
            free_command(p->rightcmd);

            free(p);
            return;
      }

      if (cmd->type == BACK) {

            b = (struct backcmd*)cmd;

            free_command(b->c);
            free(b);
            return;
      }

      e = (struct execcmd*)cmd;

      for (i = 0; i < e->argc; i++)
            free(e->argv[i]);

      for (i = 0; i < e->eargc; i++)
            free(e->eargv[i]);

      free(e);
}
```

## parsing.c

```
#include "parsing.h"

// parses an argument of the command stream input
static char* get_token(char* buf, int idx) {

	char* tok;
	int i;

	tok = (char*)calloc(ARGSIZE, sizeof(char));
	i = 0;

	while (buf[idx] != SPACE && buf[idx] != END_STRING) {
		tok[i] = buf[idx];
		i++; idx++;
	}

	return tok;
}

// parses and changes stdin/out/err if needed
static bool parse_redir_flow(struct execcmd* c, char* arg) {

	int inIdx, outIdx;

	// flow redirection for output
	if ((outIdx = block_contains(arg, '>')) >= 0) {
		switch (outIdx) {
			// stdout redir
			case 0: {
				strcpy(c->out_file, arg + 1);
				break;
			}
			// stderr redir
			case 1: {
				strcpy(c->err_file, &arg[outIdx + 1]);
				break;
			}
		}

		free(arg);
		c->type = REDIR;

		return true;
	}

	// flow redirection for input
	if ((inIdx = block_contains(arg, '<')) >= 0) {
		// stdin redir
		strcpy(c->in_file, arg + 1);

		c->type = REDIR;
		free(arg);

		return true;
	}

	return false;
}
```

```
// parses and sets a pair KEY=VALUE
// environment variable
static bool parse_environ_var(struct execcmd* c, char* arg) {

      // sets environment variables apart from the
      // ones defined in the global variable "environ"
      if (block_contains(arg, '=') > 0) {

            // checks if the KEY part of the pair
            // does not contain a '-' char which means
            // that it is not a environ var, but also
            // an argument of the program to be executed
            // (For example:
            //    ./prog -arg=value
            //    ./prog --arg=value
            // )
            if (block_contains(arg, '-') < 0) {
                  c->eargv[c->eargc++] = arg;
                  return true;
            }
      }

      return false;
}

// this function will be called for every token, and it should
// expand environment variables. In other words, if the token
// happens to start with '$', the correct substitution with the
// environment value should be performed. Otherwise the same
// token is returned.
//
// Hints:
// - check if the first byte of the argument
//    contains the '$'
// - expand it and copy the value
//     to 'arg'
static char* expand_environ_var(char* arg) {
      if (arg[0] == '$')
            return strcpy(arg,getenv(arg+1));

      return arg;
}

// parses one single command having into account:
// - the arguments passed to the program
// - stdin/stdout/stderr flow changes
// - environment variables (expand and set)
static struct cmd* parse_exec(char* buf_cmd) {

      struct execcmd* c;
      char* tok;
      int idx = 0, argc = 0, eargc = 0;

      c = (struct execcmd*)exec_cmd_create(buf_cmd);
      c->eargc = eargc;

      while (buf_cmd[idx] != END_STRING) {

            tok = get_token(buf_cmd, idx);
```

```
            idx = idx + strlen(tok);

            if (buf_cmd[idx] != END_STRING)
                  idx++;

            tok = expand_environ_var(tok);

            if (parse_redir_flow(c, tok))
                  continue;

            if (parse_environ_var(c, tok))
                  continue;

            c->argv[argc++] = tok;
      }

      //printf("eargc = %s\n", c->eargc);
      c->argv[argc] = (char*)NULL;
      c->argc = argc;

      return (struct cmd*)c;
}

// parses a command knowing that it contains
// the '&' char
static struct cmd* parse_back(char* buf_cmd) {

      int i = 0;
      struct cmd* e;

      while (buf_cmd[i] != '&')
            i++;

      buf_cmd[i] = END_STRING;

      e = parse_exec(buf_cmd);

      return back_cmd_create(e);
}

// parses a command and checks if it contains
// the '&' (background process) character
static struct cmd* parse_cmd(char* buf_cmd) {

      if (strlen(buf_cmd) == 0)
            return NULL;

      int idx;

      // checks if the background symbol is after
      // a redir symbol, in which case
      // it does not have to run in in the 'back'
      if ((idx = block_contains(buf_cmd, '&')) >= 0 &&
                  buf_cmd[idx - 1] != '>')
            return parse_back(buf_cmd);

      return parse_exec(buf_cmd);
}

// parses the command line
```

```
// looking for the pipe character '|'
struct cmd* parse_line(char* buf) {

      struct cmd *r, *l;

      char* right = split_line(buf, '|');

      l = parse_cmd(buf);
      r = parse_cmd(right);

      return pipe_cmd_create(l, r);
}
```

## readline.c

```
#include "defs.h"
#include "readline.h"

static char buffer[BUFLEN];

// read a line from the standar input
// and prints the prompt
char* read_line(const char* promt) {

      int i = 0,
          c = 0;

      fprintf(stdout, "%s %s %s\n", COLOR_RED, promt, COLOR_RESET);
      fprintf(stdout, "%s", "$ ");

      memset(buffer, 0, BUFLEN);

      c = getchar();

      while (c != END_LINE && c != EOF) {
            buffer[i++] = c;
            c = getchar();
      }

      // if the user press ctrl+D
      // just exit normally
      if (c == EOF)
            return NULL;

      buffer[i] = END_STRING;

      return buffer;
}
```

## runcmd.c

```
#include "runcmd.h"

int status = 0;
struct cmd* parsed_pipe;

// runs the command in 'cmd'
int run_cmd(char* cmd) {

	pid_t p;
	struct cmd *parsed;

	// if the "enter" key is pressed
	// just print the promt again
	if (cmd[0] == END_STRING)
		return 0;

	// cd built-in call
	if (cd(cmd))
		return 0;

	// exit built-in call
	if (exit_shell(cmd))
		return EXIT_SHELL;

	// pwd buil-in call
	if (pwd(cmd))
		return 0;

	// parses the command line
	parsed = parse_line(cmd);

	// forks and run the command
	if ((p = fork()) == 0) {

		// keep a reference
		// to the parsed pipe cmd
		// so it can be freed later
		if (parsed->type == PIPE)
			parsed_pipe = parsed;

		exec_cmd(parsed);
	}

	// store the pid of the process
	parsed->pid = p;

	// background process special treatment
	// Hint:
	// - check if the process is
	//    going to be run in the 'back'
	// - print info about it with
	//    'print_back_info()'
	//
	if(parsed->type == BACK)
		print_back_info(parsed);
	else{
		waitpid(p, &status, 0); // waits for the process to finish
	}
```

```
        print_status_info(parsed);

        free_command(parsed);

        return 0;
}
```

## sh.c

```
#include "defs.h"
#include "types.h"
#include "readline.h"
#include "runcmd.h"

char promt[PRMTLEN] = {0};

// runs a shell command
static void run_shell() {

      char* cmd;

      while ((cmd = read_line(promt)) != NULL)
            if (run_cmd(cmd) == EXIT_SHELL)
                  return;
}

// initialize the shell
// with the "HOME" directory
static void init_shell() {

      char buf[BUFLEN] = {0};
      char* home = getenv("HOME");

      if (chdir(home) < 0) {
            snprintf(buf, sizeof buf, "cannot cd to %s ", home);
            perror(buf);
      } else {
            snprintf(promt, sizeof promt, "(%s)", home);
      }
}

int main(void) {

      init_shell();

      run_shell();

      return 0;
}
```

## utils.c

```
#include "defs.h"
#include "types.h"
#include "readline.h"
#include "runcmd.h"

char promt[PRMTLEN] = {0};

// runs a shell command
static void run_shell() {

      char* cmd;

      while ((cmd = read_line(promt)) != NULL)
            if (run_cmd(cmd) == EXIT_SHELL)
                  return;
}

// initialize the shell
// with the "HOME" directory
static void init_shell() {

      char buf[BUFLEN] = {0};
      char* home = getenv("HOME");

      if (chdir(home) < 0) {
            snprintf(buf, sizeof buf, "cannot cd to %s ", home);
            perror(buf);
      } else {
            snprintf(promt, sizeof promt, "(%s)", home);
      }
}

int main(void) {

      init_shell();

      run_shell();

      return 0;
}
```