Sistemas Operativos

Mendez-Simó

Lab Shell

Entrega Parte 1 y 2

27/04

Hernán Tain

exec.c

```
void exec_cmd(struct cmd* cmd) {

      struct execcmd* c;
      switch (cmd->type) {

            case EXEC:
                  c = (struct execcmd*) cmd;
                  if((execvp(c->argv[0], c->argv)) < 0)
                        _exit(-1);
                  break;

            case BACK: {
                  // runs a command in background
                  //
                  // Your code here
                  printf("Background process are not yet implemented\n");
                  _exit(-1);
                  break;
            }

            case REDIR: {
                  // changes the input/output/stderr flow
                  //
                  // Your code here
                  printf("Redirections are not yet implemented\n");
                  _exit(-1);
                  break;
            }

            case PIPE: {
                  // pipes two commands
                  //
                  // Your code here
                  printf("Pipes are not yet implemented\n");

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

                  break;
            }
      }
}
```

parsing.c

```
static char* expand_environ_var(char* arg) {

      if (arg[0] == '$')
            return strcpy(arg,getenv(arg+1));;

      return arg;
}
```

FIN PRIMERA PARTE

Parte 2

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;
}
```

exec.c (solo set_environ_vars y exec_cmd fueron modificados)

```
#include "exec.h"

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);
      }
}

// 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 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);

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

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

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

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

      case PIPE: {
            // pipes two commands
            //
            // Your code here
            printf("Pipes are not yet implemented\n");

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

            break;
      }
   }
}
```

parsing.c (solo parse_exec fue modificado)

```
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;
}
```

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;
}
```

--- PREGUNTAS ---

1. ¿entre cd y pwd, alguno de los dos se podría implementar sin necesidad de ser *built-in*? ¿por qué? ¿cuál es el motivo, entonces, de hacerlo como *built-in*? (para esta última pregunta pensar en los *built-in* como true y false)

Creo que cd, se podría implementar sin la necesidad de ser built-in con el comando chdir. Por lo que pude investigar, antes cd no era un comando built-in y cuando se creó fork, dejo de funcionar como programa por lo que tuvieron que implementarlo como un comando built-in.

2. luego de llamar a fork(2) realizar, por cada una de las variables de entorno a agregar, una llamada a setenv(3)
¿por qué es necesario hacerlo luego de la llamada a fork(2) ?

Es necesario realizar esta llamada luego del fork para que las variables de entorno agregadas queden “vivas” solo mientras este vivo el proceso hijo.

3.Detallar cúal es el mecanismo utilizado. (Del proceso en segundo plano)

Primero para poder ejecutar un proceso en segundo plano fue necesario hacer un “if” luego del fork. Esto es para que luego de crear el proceso hijo, el padre no esperara hasta que este termine para tener control del shell.
Luego en el script ‘exec.c’ se llama recursivamente a la funcion exec_cmd cuando el comando es del tipo BACK.

(Eso es porque el struct del back es igual al generico con el agregado de tener tambien referencia a lo que seria el comando en sí sin el ‘&’ que indica el background process.)