Lab Shell

Dylan Alvarez, 98225
1º cuatrimestre, 27/04/2018

Parte 2: Invocación avanzada

Comandos *built-in* ☆

Implementación:

`runcmd.c` fue modificado para poder utilizar la expansión de variables de entorno en comandos built-in (por ejemplo, `cd $OLDPWD`). Esto se hizo mediante un parseo temprano de lo ingresado a la terminal, dado que de otro modo lograr esto habría resultado en repetición de código:

```
#include "runcmd.h"

int status = 0;
struct cmd *parsed_pipe;

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

    pid_t p;
    // parses the command line
    struct cmd *parsed = parse_line(cmd);

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

    if (parsed->type == EXEC) {
        // cd built-in call
        if (cd((struct execcmd *) parsed))
            return 0;

        // exit built-in call
        if (exit_shell((struct execcmd *) parsed)) {
            free_command(parsed);
            return EXIT_SHELL;
        }

        // pwd built-in call
        if (pwd((struct execcmd *) parsed))
            return 0;
    }

    // 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
    if (parsed->type != BACK) {
        // waits for the process to finish
        waitpid(p, &status, 0);
        print_status_info(parsed);
    }

    free_command(parsed);

    return 0;
}
```

builtin.c

```
#include "builtin.h"
#include "freecmd.h"

// returns true if the 'exit' call
// should be performed
int exit_shell(struct execcmd *parsed) {
    return strncmp(parsed->scmd, "exit", 4) == 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(struct execcmd *parsed) {
    bool invoked = false;
    if (parsed->argc > 0 && strcmp(parsed->argv[0], "cd") == 0) {
        if (chdir(parsed->argc == 1 ? getenv("HOME") : parsed->argv[1]) == -1) {
            perror(SHELL_NAME);
        } else {
            char *current_dir = get_current_dir_name();
            snprintf(prompt, sizeof prompt, "(%s)", current_dir);
            free(current_dir);
        }
        invoked = true;
    }
    return invoked;
}

// returns true if 'pwd' was invoked
// in the command line
```

```
int pwd(struct execcmd *parsed) {
    bool invoked = false;
    if (parsed->argc > 0 && strcmp(parsed->argv[0], "pwd") == 0) {
        char *current_dir = get_current_dir_name();
        printf("%s\n", current_dir);
        free(current_dir);
        invoked = true;
    }
    return invoked;
}
```

Pregunta: ¿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`)

Respuesta: `pwd` podría ser implementado como un comando, ya que su *working directory* es el mismo que el del proceso padre. Se lo suele hacer como *built-in* para evitar la generación de un nuevo proceso, con todo lo que eso implica (`fork`, `exec`) en términos de uso de recursos. Es el mismo caso con los *built-in* `true` y `false`: el resultado obtenido (un *return code*) no amerita el uso de recursos necesario para instanciar un nuevo proceso.

Variables de entorno adicionales ☆☆

Implementación:

`exec.c`

```
void exec_cmd(struct cmd *cmd) {
    struct execcmd *execcmd = (struct execcmd *) cmd;
    struct backcmd *backcmd = (struct backcmd *) cmd;
    switch (cmd->type) {
        case EXEC:
            // spawns a command
            for (int i = 0; execcmd->eargv[i] != NULL; i++) {
                char *name = execcmd->eargv[i];
                char *value = NULL;
                int position = 0;
                while (value == NULL) {
                    if (name[position] == '=') {
                        name[position] = 0;
                        value = name + position + 1;
                    } else {
                        position++;
                    }
                }
                if (setenv(name, value, true) == -1) {
                    char buf[BUFLEN] = {0};
                    snprintf(buf, sizeof buf,
                             "cannot define environment variable %s", name);
                    perror(buf);
                }
                name[position] = '=';
            }
            if (execvp(
                    execcmd->argv[0],
                    execcmd->argv
            ) == -1) {
```

```
            perror(SHELL_NAME);
        }
        break;
```

Pregunta: ¿por qué es necesario hacerlo luego de la llamada a `fork(2)` ?

Respuesta: Porque luego de la llamada a `fork` las variables de entorno están definiéndose para el proceso hijo solamente, y antes se están definiendo para el padre (shell), ergo el comportamiento no es el deseado: la definición/el reemplazo de los valores de las variables de entorno no afecta solamente al proceso hijo.

- En algunas de los *wrappers* de la familia de funciones de `exec(3)` (las que finalizan con la letra *e*), se les puede pasar un tercer argumento (o una lista de argumentos dependiendo del caso), con nuevas variables de entorno para la ejecución de ese proceso.

Supongamos, entonces, que en vez de utilizar `setenv(3)` por cada una de las variables, se guardan en un array y se lo coloca en el tercer argumento de una de las funciones de `exec(3)` .

Responder (opcional):

- ¿el comportamiento es el mismo que en el primer caso? Explicar qué sucede y por qué.
- Describir brevemente una posible implementación para que el comportamiento sea el mismo.

Respuesta: El comportamiento no será el mismo ya que estos wrappers utilizan ese array de variables de enotrno como únicas variables de entorno para el proceso: no 'heredan' las variables de entorno del proceso padre. Para mantener el comportamiento habría que copiar todas las variables de entorno a este arreglo además de agregarle las nuevas definiciones.

Procesos en segundo plano ☆☆☆

Se evita el `wait` para los procesos hijos que se corren en segundo plano. Luego se ejecuta el comando normalmente.

`runcmd.c`

```
    // background process special treatment
    if (parsed->type != BACK) {
        // waits for the process to finish
        waitpid(p, &status, 0);
        print_status_info(parsed);
    }
```

`exec.c`

```
        case BACK: {
            // runs a command in background
            exec_cmd(backcmd->c);
            break;
        }
```