Resolución de la segunda parte

Comandos built-in

La implementación de los comandos built-in pedidos se lleva a cabo en los esqueletos de las funciones con su mismo nombre en el módulo `builtin.c`

```
#include "builtin.h"
#include "defs.h"

// returns true if the 'exit' call
// should be performed
int exit_shell(char* cmd) {
    char command[ARGSIZE];
    if (sscanf(cmd, "%s", command) < 0) {
        perror("exit sscanf");
        return 0;
    }

    if (strncmp(command, "exit", ARGSIZE) != 0) {
        return 0;
    }

    return 1;
}

// 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) {
    char command[ARGSIZE];
    char dir[BUFLEN] = {0};
    if (sscanf(cmd, "%s %s", command, dir) < 0) {
        perror("cd sscanf");
        return 0;
    }

    if (strncmp(command, "cd", ARGSIZE) != 0) { // not cd
        return 0;
    }

    if (!strlen(dir)) {
        strncpy(dir, getenv("HOME"), ARGSIZE);
    }

        if (chdir(dir) < 0) {
        perror("cd chdir");
        return 0;
    }

    if (getcwd(dir, BUFLEN) < 0) {
        perror("cd getcwd");
        return 0;
    }

    snprintf(promt, PRMTLEN, "(%s)", dir);
        return 1;
}

// returns true if 'pwd' was invoked
// in the command line
int pwd(char* cmd) {
    char command[ARGSIZE];
    if (sscanf(cmd, "%s", command) < 0) {
        perror("pwd sscanf");
        return 0;
    }

    if (strncmp(command, "pwd", ARGSIZE) != 0) {
        return 0;
    }
```

```
    char wd[BUFLEN];
    if (getcwd(wd, BUFLEN) < 0) {
        perror("cd getcwd");
        return 0;
    }
    printf("%s\n", wd);
    return 1;
}
```

> *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`)

`pwd` podría ser implementado como un programa aparte de la shell, `cd` no.

`pwd` imprime por salida estándar el directorio actual del usuario y sale, eso puede hacerse tranquilamente en un programa de C, sin embargo es implementado como built-in porque el directorio actual es una variable que la shell ya debe mantener actualizada y es más eficiente imprimir esa variable que hacer el proceso de `fork` y siguiente `exec` a `pwd`. Por esta misma razón `true` y `false` son built-ins en shells, porque su función de salir con status codes `0` y `1` respectivamente es tan simple que no amerita una llamada a `exec`. `cd` no puede ser implementado como un programa aparte porque su función es modificar el directorio de trabajo actual del usuario, para ello esta modificación debe ser hecha en el proceso del shell en sí, y no puede hacerse un `fork` y ejecutarse aparte. Si se hiciera un `fork` se modificaría el *working directory* del proceso hijo, mantentiendo el padre (la shell en sí) sin cambiar.

Variables de entorno adicionales

Para la implementación de esta funcionalidad se modifica la función `run_cmd` de `runcmd.c` para agregar una llamada a una función que se encarga de setear las variables de entorno pedidas inmediatamente después de hacer el `fork`:

```
    ...

    // forks and run the command
    if ((p = fork()) == 0) {
        set_env_variables(parsed);
        ...
    }

    ...
```

La función `set_env_variables`:

```
void set_env_variables(struct cmd* base_cmd) {
    struct execcmd* cmd = NULL;
    if (base_cmd->type == EXEC) {
        cmd = (struct execcmd*) base_cmd;
    } else if (base_cmd->type == BACK) {
        cmd = (struct execcmd*) ((struct backcmd*) base_cmd)->c;
    } else {
        fprintf(stderr, "Unrecognized type: %d\n", base_cmd->type);
        return;
    }

    for (int i = 0; i < cmd->eargc; ++i) {
        char* value = NULL;
        char name[ARGSIZE] = {0};
        size_t name_len = 0;
        for (int j = 0; j < strlen(cmd->eargv[i]); ++j) {
            if (cmd->eargv[i][j] == '=') {
                value = cmd->eargv[i] + j + 1;
                name_len = j;
                break;
            }
        }
        if (!value) {
            fprintf(stderr, "eargv without '=' found\n");
            return;
        }

        strncpy(name, cmd->eargv[i], name_len);
        if (setenv(name, value, 1) < 0) {
            perror("setenv");
```

```
        }
    }
}
```

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

La funcionalidad deseada de este seteo de variables de entorno es que las variables existan únicamente para el proceso a ejecutar, y que no existan más para los siguientes (comportamiento probado con `bash`). Haciendo la llamada luego del `fork` se garantiza que estas variables existan únicamente en el proceso hijo, y que no "contaminen" al entorno del padre.

> 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.

Según la *manpage* de `exec` :

```
The execle() and execvpe() functions allow the caller to specify the
environment of the executed program via the argument envp. The envp
argument is an array of pointers to null-terminated strings and must be
terminated by a null pointer. The other functions take the environment
for the new process image from the external variable environ in the
calling process.
```

Esto da a entender que si se setea el ambiente de la nueva imagen del proceso con `execle` o `execvpe` , el entorno definido con `envp` será la totalidad de las variables definidas, y no se encontrarán las variables definidas en `environ` . Este comportamiento fue verificado con dos programas de ejemplo, a continuación se muestra su código fuente: `parent.c` :

```c
#define _GNU_SOURCE
#include <unistd.h>

int main() {

    char* argv[] = {"./target", "PATH", NULL};

    char* env[] = {NULL};
    execvpe("./target", argv, env);
}
```

target_app.c

```c
#include <stdio.h>
#include <stdlib.h>


int main(int argc, char** argv) {
    char* expanded = getenv(argv[1]);
    if (!expanded) {
        printf("Error expandiendo %s\n", argv[1]);
        return 1;
    }
    printf("%s\n", expanded);
    return 0;
}
```

Asumiendo que `PATH` es una variable definida en el entorno "default", el programa `target` debería imprimir su expansión. Sin embargo, falla cuando es ejecutada a partir de una imagen de proceso creada desde `parent` , debido a que la llamada a `execvpe` con el entorno vacío no incluye a la variable `PATH` .

La manera de implementar este seteo de variables adicionales usando `execvpe` sería copiar **todas** las variables de entorno definidas en `environ` a un array propio, luego agregar a este array las definidas por el usuario en la entrada de su comando, y ahí pasar ese array a la función.

Procesos en segundo plano

Para esta funcionalidad se modificó de nuevo `run_cmd` de `runcmd.c`, y `exec_cmd` de `exec.c`:

`run_cmd`

```
int run_cmd(char* cmd) {
    ...
        // 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);
        free_command(parsed);
        return 0;
    }
    ...
```

`exec_cmd`

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

        switch (cmd->type) {
        ...

                case BACK: {
                        // runs a command in background
                        //
            handle_back(cmd);
                        _exit(-1);
                        break;
                }
                ...
```

con `handle_back`:

```
void handle_back(struct cmd* cmd) {
    struct backcmd* back = (struct backcmd*) cmd;
    struct execcmd* execcmd = (struct execcmd*) back->c;
    execvp(execcmd->argv[0], execcmd->argv);

}
```

> Detallar cúal es el mecanismo utilizado.

La diferencia principal entre los comandos "comunes" y los de tipo *background* en la aplicación, más allá de los parseos particulares para detectarlos, es que en los comunes la shell espera a que el proceso hijo forkeado termine de ejecutarse haciendo un `waitpid` del pid devuelto por `fork`. En los de background simplemente no se hace eso, y se le permite al usuario ejecutar otro comando inmediatamente después del `fork`.