Functions used

You are allowed to use several functions in this project. You already know some of them like `write`, `ft_printf`, `malloc`, `free` and all the functions from your `libft`. However, other important functions that have never been used before will be **essential** to the success of this project. Let's look at them together.

access()

```
int access(const char *pathname, int mode);
```

`access()` checks whether the program can access the file `pathname`.

The `mode` specifies the accessibility check(s) to be performed, and is either the value `F_OK`, or a mask consisting of the bitwise OR of one or more of `R_OK`, `W_OK`, and `X_OK`. `F_OK` tests for the existence of the file. `R_OK`, `W_OK`, and `X_OK` test whether the file exists and grants read, write, and execute permissions, respectively.

On success (all requested permissions granted), zero is returned. On error (at least one bit in `mode` asked for a permission that is denied, or some other error occurred), -1 is returned, and `errno` is set appropriately.

access() example

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

int main()
{
    if (access("readfile", R_OK) == 0)
        printf("readfile is accessible in reading mode\n");
    if (access("writefile", W_OK) == 0)
        printf("writefile is accessible in writing mode\n");
    if (access("executefile", X_OK) == 0)
        printf("executefile is accessible in execution mode\n");
    if (access("rwfile", R_OK|W_OK) == 0)
        printf("rwfile is accessible in writing and reading mode\n");
}
```

In this example, we use the `access` function multiple times.

The first time we check whether the program can read `readfile` or not.

The second time we check whether the program can write in `writefile` or not.

The third time we check whether the program can execute `executefile` or not.

The fourth time is an example using the bitwise `OR` operator to check whether the file `rwfile` is accessible in read `AND` write mode or not.

You can find more information on `access()` here: https://linux.die.net/man/2/access

dup2()

```c
int dup2(int oldfd, int newfd);
```

`dup2()` makes `newfd` be the copy of `oldfd`, closing `newfd` first if necessary, but note the following:

- If `oldfd` is not a valid file descriptor, then the call fails, and `newfd` is not closed.
- If `oldfd` is a valid file descriptor, and `newfd` has the same value as `oldfd`, then `dup2()` does nothing, and returns `newfd`.

After a successful return from `dup2()`, the old and new file descriptor may be used interchangeably. They refer to the same open file description and thus share file offset and file status flags; for example, if the file offset is modified by using `lseek()` on one of the descriptors, the offset is also changed for the other.

On error, the `dup2()` function returns `-1`.

dup2() example

```c
#include <unistd.h>
#include <fcntl.h>

int main(int ac, char *av[], char *env[])
{
    (void) ac;
    (void) av;
    int in;
    int out;
    char *grep_args[] = {"grep", "Lausanne", NULL};

    // open input and output files (assuming both files exist)
    in = open("in", O_RDONLY);
    out = open("out", O_WRONLY);

    // replace standard input with input file
    dup2(in, 0);
    // close unused file descriptors
    close(in);
    close(out);

    // execute grep
    execve("grep", grep_args, env);
}
```

In this example, first we open both `in` and `out` file, in reading and writing mode respectively.

Then we use `dup2()` to replace the `stdin` file descriptor by the `in` file descriptor.
This way, whatever the command that comes after will read from the `stdin` will be whatever the content of `in` is since the `stdin` file descriptor now "points" to the `in` file.

Then, we can close `in` and `out`, we don't use them anymore, right ?

We set the `stdin` file descriptor to be the same as `in`, so now we only use `stdin`, `in` and `out` are not used anymore, we can close them.

Now, we use the `execve()` function to execute the `grep` shell command (this is explained below on this page). When `grep` is launched without any argument, it reads text from the standard input before executing. What will happen then ?

> Remember we replaced the `stdin` file descriptor by `in`, so `grep` will read from the standard input, the standard input now reads from the `in` file, so `grep` will execute on whatever the content of the `in` file is.

You can find more information on `dup2()` here: https://linux.die.net/man/2/dup2

pipe()

```
int pipe(int pipefd[2]);
```

`pipe()` creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array `pipefd` is used to return two file descriptors referring to the ends of the pipe. `pipefd[0]` refers to the read end of the pipe. `pipefd[1]` refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe

On success, `0` is returned. On error, `-1` is returned, and `errno` is set appropriately.

pipe() example

```c
/**
 * Executes the command "cat infile | grep Lausanne".
 */

#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int ac, char *av[], char *env[])
{
    (void) ac;
    (void) av;
    int pipefd[2];
    int pid;

    char *cat_args[] = {"/bin/cat", "infile", NULL};
    char *grep_args[] = {"/usr/bin/grep", "Lausanne", NULL};

    // make a pipe
    // fds go in pipefd[0] and pipefd[1]
    pipe(pipefd);

    if (pid == 0)
    {
        // child process gets here and handles "grep Lausanne"
        // replace standard input with input part of the pipe
        dup2(pipefd[0], 0);

        // close unused half of the pipe
        close(pipefd[1]);

        // execute grep
        execve("/usr/bin/grep", grep_args, env);
    }
    else
    {
        // parent process gets here and handles "cat scores"
        // replace standard output with output part of pipe
        dup2(pipefd[1], 1);

        // close unused half of the pipe
        close(pipefd[0]);

        // execute cat
        execve("/bin/cat", cat_args, env);
    }
```

```
    }
    // close unused pipe
    close(pipefd[0]);
    close(pipefd[1]);

    // wait for the child process to finish
    waitpid(pid);
}
```

Read the comments to check what is happening, it's pretty hard to explain in an other way, I'll add schema when I will be finished with the project.

In this example we create a pipe, and replace the standard input with the input part of the pipe for our child process.

For the parent process, we replace the standard output with the output part of the pipe.

At the very end, we use the `waitpid()` function to wait for the child process to finish before making the prompt reappear.

fork()

```
pid_t fork(void);
```

`fork()` creates a new process by duplicating the calling process. The new process, referred to as the `child`, is an exact duplicate of the calling process, referred to as the `parent`, except for some points that you can find here (there's too may to write them all down here).

fork() example

```c
/**
 * Executes the command "cat infile | grep Lausanne".
 */

#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int ac, char *av[], char *env[])
{
    (void) ac;
    (void) av;
    int pipefd[2];
    int pid;

    char *cat_args[] = {"/bin/cat", "infile", NULL};
    char *grep_args[] = {"/usr/bin/grep", "Lausanne", NULL};

    // make a pipe
    // fds go in pipefd[0] and pipefd[1]
    pipe(pipefd);

    if (pid == 0)
    {
        // child process gets here and handles "grep Lausanne"
        // replace standard input with input part of the pipe
        dup2(pipefd[0], 0);

        // close unused half of the pipe
        close(pipefd[1]);

        // execute grep
        execve("/usr/bin/grep", grep_args, env);
    }
    else
    {
        // parent process gets here and handles "cat scores"
        // replace standard output with output part of pipe
        dup2(pipefd[1], 1);

        // close unused half of the pipe
        close(pipefd[0]);

        // execute cat
        execve("/bin/cat", cat_args, env);
    }
```

```
    }
    // close unused pipe
    close(pipefd[0]);
    close(pipefd[1]);

    // wait for the child process to finish
    waitpid(pid);
}
```

I used the same example as the pipe function because it goes with it (at leas for this project), each command you want to execute takes its own child process, so you have to fork your parent into as many child processes as commands you have to execute.

waitpid()

```
pid_t waitpid(pid_t pid, int *status, int options);
```

The `waitpid()` system call suspends execution of the calling process until a child specified by *pid* argument has changed state. By default, `waitpid()` waits only for terminated children.

waitpid() example

```
#include <sys/types.h>
#include <sys/wait.h>

int main(void)
{
    int pid;

    pid = fork();

    waitpid(pid);
}
```

In this example, we create a child process by forking the main process. We save the pid of the child in the `pid` variable and we wait for it at the end.

You can find more information about `wait()` and `waitpid()` here.

wait()

```
pid_t wait(int *status);
```

The `wait()` system call suspends execution of the calling process until one of its children terminates.

<details>
<summary>wait() example</summary>

```c
#include <sys/types.h>
#include <sys/wait.h>

// waiting for one child
int main(void)
{
    int status;

    wait(&status);
}

// waiting for multiple children
int main(void)
{
    int status;

    while (n > 0)
    {
        wait(&status);
        n--;
    }
}
```

In the first main function, we wait for one child process to terminate before going further.
In the second main function, we wait for children one at a time, each time one terminates, we remove `1` from `n` until it reaches `0` and then continue.
</details>

You can find more information about `wait()` and `waitpid()` here.

execve()

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```

`execve()` executes the program pointed to by `filename`.

execve() does not return on succes, the calling process is **replaced** by the executed `filename`.

<details>
<summary>execve() example</summary>

```
#include <unistd.h>

int main(int ac, char **av, char **envp)
{
    (void) ac;
    const char *filename = "/usr/bin/grep";
    char *const argv[] = {"/usr/bin/grep", "a", NULL};

    execve(filename, argv, envp);
}
```

</details>

unlink()

```
int unlink(const char *pathname);
```

`unlink()` deletes a name from the file system. If that name was the last link to a file and no processes have the file open the file is deleted and the space it was using is made available for reuse.

If the name was the last link to a file but any processes still have the file open the file will remain in existence until the last file descriptor referring to it is closed.

On success, `0` is returned. On error, `-1` is returned, and `errno` is set appropriately.

unlink() example

```c
#include <unistd.h>

int main(void)
{
    if (access("tmp", F_OK) == 0)
        unlink("tmp");
    return (0);
}
```

In this example, we use the `access()` function to check if the file called `tmp` exists, if its the case, we use the `unlink()` function to remove it.

You can find more information about `unlink()` here.

Final word

The way pipes are described generally is that it redirects the output of one command to the input of the next command. That is correct. But there's a catch, when you build pipex, you have to launch a new child process for each command you want to do, and they are all made at the same time.

That means all commands are run at the same time, it's just that they will wait for the writing end of the pipe to be closed before reading from the pipe.

Previous
Understand pipex

Next
Building the thing

Last updated 14 days ago