...Shell v2

Everything You Need To Know - **Shell v2**

DUP2

```
ls > ls_result
```

The previous command will execute `ls` and redirect `stdout` to the file named `ls_result`. If a file named `ls_result` already exists, it will be overwritten. If it does not exist, it will be created.

In this case, we need to tell `ls` not to print on `stdout` but instead in the file `ls_result`. If you remember your first version of the shell, we use `execve` to copy the `ls` executable into the current process. Once `execve` is called, we lose control on the process, and we just wait for it to exit using `wait` in the parent process.

So we need to manage the output redirection before executing any command with `execve`. Our solution is a system call (man 2) named `dup2`.

```
alex@~$ cat main_0.c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

/**
 * main - dup2
 *
 * Return: EXIT_SUCCESS, or exit code
 */
int main(void)
{
    if (dup2(STDERR_FILENO, STDOUT_FILENO) == -1)
    {
        perror("dup2");
        return (EXIT_FAILURE);
    }

    /* Let's try to print something on stdout */
    printf("Test on stdout\n");
    printf("Holberton School\n");
    return (EXIT_SUCCESS);
}
alex@~$ gcc -Wall -Wextra -Werror -pedantic main_0.c
alex@~$ ./a.out
Test on stdout
Holberton School
alex@~$ ./a.out > /dev/null
Test on stdout
Holberton School
alex@~$ ./a.out 2> /dev/null
alex@~$
```

Here you can see that what was initially printed on `stdout` with `printf`, is now printed on `stderr`. You can check that by redirecting the output on `/dev/null`, and only `stderr` should remain.

Exercises

0. stdout to a file

Write a program that opens a file (in write mode) and redirects `stdout` to this file. Try to print anything on `stdout` using `printf`, `puts`, `write`, etc and it should be printed in the file you opened. You can try to enhance your program by passing the file to write in as an argument.

1. stdout to end of a file

Write a program that opens a file (in write mode) and redirects `stdout` at the end of this file. Try to print anything on `stdout` using `printf`, `puts`, `write`, etc and it should be appended at the end of the file you opened. You can try to enhance your program by passing the file to write in as an argument.

2. file to command stdin

Write a program that opens a file, and executes the command `/usr/bin/rev` with the file's content

as input.

PIPE

```
ls | cat -e
```

You probably used it a lot without even knowing how it works: the `pipe`. The pipe is used to make the output of a process the input of another process. In the above command, the output of the `ls` command will be piped to be the input of the `cat -e` command.

Do not confuse input and arguments:

```
cat -e
```

Here, `-e` is an argument, but the command `cat` will read its input from the standard input (a.k.a `stdin`)

To reproduce this behaviour we will need another fantastic system call (man 2) ——> `pipe` !

Please read carefully the manual page of `pipe(2)`, along with the manual page of `pipe(7)`.

Example from the manual page of `pipe(2)`:

The following program creates a pipe, and then `fork(2)`s to create a child process; the child inherits a duplicate set of file descriptors that refer to the same pipe. After the `fork(2)`, each process closes the descriptors that it doesn't need for the pipe (see `pipe(7)`). The parent then writes the string contained in the program's command-line argument to the pipe, and the child reads this string a byte at a time from the pipe and echoes it on standard output.

```
alex@~$ cat pipe_example.c
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define READ_END    0
#define WRITE_END   1

/**
 * main - pipe(2) manual page example
 * @argc: Arguments counter
 * @argv: Arguments vector
 *
 * Return: EXIT_SUCCESS or EXIT_FAILURE
 */
int main(int argc, char *argv[])
{
    int pipefd[2];
    pid_t cpid;
    char buf;

    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s <string>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (pipe(pipefd) == -1)
    {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    cpid = fork();
    if (cpid == -1)
    {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (cpid == 0)                /* Child reads from pipe */
    {
        close(pipefd[WRITE_END]);          /* Close unused write end */
        while (read(pipefd[READ_END], &buf, 1) > 0)
            write(STDOUT_FILENO, &buf, 1);
        write(STDOUT_FILENO, "\n", 1);
        close(pipefd[READ_END]);
        _exit(EXIT_SUCCESS);
    }
    else                          /* Parent writes argv[1] to pipe */
    {
        close(pipefd[READ_END]);          /* Close unused read end */
```

```
        write(pipefd[WRITE_END], argv[1], strlen(argv[1]));
        close(pipefd[WRITE_END]);          /* Reader will see EOF */
        wait(NULL);                /* Wait for child */
        exit(EXIT_SUCCESS);
    }
}
alex@~$ gcc -Wall -Wextra -Werror -pedantic pipe_example.c
alex@~$ ./a.out Holberton
Holberton
alex@~$
```

DUP2 + PIPE

To create a `pipe` between two processes, we need to combine the functions `dup2` and `pipe`. Here's the process: We first need to create a pipe using `pipe (2)`, and then use `dup2 (2)` to redirect the output of the first command into the write end of our pipe. Finally, we need to redirect the read end of our pipe into the input of our second command. Easy right?

Exercises

3. stdout to command input

Write a program that pipes `stdout` to the command `/usr/bin/rev`. For this exercise, you'll need to use `dup2`, `pipe` and `execve`. You have to create a pipe, redirect `stdout` to the write end of the pipe, and redirect the read end of the pipe to be the input of `rev`, so everything printed on `stdout` using `printf`, `write`, ... should be printed reversed.

The result should be the same as if you would do something like:

```
alex@~$ echo "Something" | /usr/bin/rev
gnihtemoS
alex@~$
```

4. pipe between two processes

Here's the command we will simulate in this exercise:

```
ls | rev
```

For this exercise, you'll need to use the functions `fork`, `execve`, `dup2` and `pipe`. Write a program that executes the command `/bin/ls` in a forked process. Then, execute the command `/usr/bin/rev` in another forked process, but the output from `ls` must be piped to `rev`. (Try to do this by yourself. If ever you're struggling, you can look at the code example in the `pipe (2)` manual page)

Now, is your program running infinitely, waiting for input after displaying a reversed `ls`? Take a look at the `pipe (2)` manual page, and look at the code example. When reading from a pipe, it is safer to close the write end of the pipe, and vice versa.