# 42: A Comprehensive Guide to Pipex

## Introduction

Pipex is a project you'll likely encounter on your 42 journey, and one which may give you a bit of a headache. Its purpose is to teach you some basic UNIX operations, and will greatly help you in the completion of Minishell (a mandatory project you'll face later).

Pipex focuses on three main concepts: pipelines, child processes and execution of commands. I will cover all three of these topics as best I can, and guide you through the main parts of the program.

## Understanding the Whitelisted Functions

### dup2(2)

`dup2(2)` helps you 'replace' open file descriptors. By default, FD 0, 1 and 2 are open and are set to `stdin`, `stdout` and `stderr` respectively. `dup2(2)` allows you to replace these with another FD, which you may obtain with `open(2)`. This can be useful for redirecting output from one FD to another, like using `printf(3)` to print to a file instead of the terminal.

Here is an example of using `dup2(2)` to redirect the output of a process from the terminal to a file:

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

int main()
{
	int fd;

	fd = open("example.txt", O_WRONLY | O_CREAT, 0644);
	dup2(fd, STDOUT_FILENO);
	close(fd);
	printf("This is printed in example.txt!\n");

	return (0);
}
```

This program opens a file called `example.txt` and uses `dup2(2)` to redirect `stdout` to the file descriptor returned by `open(2)`. This means that any output from `printf(3)` will be written to the file instead of the terminal. The file is then closed, and the `printf(3)` statement writes to the file.

### access(2)

`access(2)` checks whether a process has permission to access a file or directory. It takes two arguments: the path to the file or directory, and a mode representing the type of access being checked. The mode is specified using constants such as `R_OK`, `W_OK`, and `X_OK`, which represent read, write, and execute permissions, respectively.

For example, the following program checks whether the process has read permission for the file `example.txt`:

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

int main()
{
        if (access("example.txt", R_OK) != -1)
                printf("I have permission\n");
        else
                printf("I don't have permission\n");

        return (0);
}
```

## execve(2)

`execve(2)` is a system call that allows you to execute another program from within your program. It replaces the current process image with a new process image, effectively running a new program. It takes three arguments: the path to the program to be executed, an array of command line arguments, and an array of environment variables.

Here is an example of using `execve(2)` to run the `ls` command:

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

int main()
{
        char *args[3];

        args[0] = "ls";
        args[1] = "-l";
        args[2] = NULL;
        execve("/bin/ls", args, NULL);
        printf("This line will not be executed.\n");

        return (0);
}
```

In this code, the `args` array contains the command line arguments to be passed to the `ls` command. `execve(2)` is then called with the path to the `ls` command (`/bin/ls`), the `args` array, and `NULL` for the environment variables. This replaces the current process image with the `ls` command, and the output of `ls -l` will be printed to the terminal. The `printf()` statement after `execve(2)` will not be executed, as the process image has been replaced.

## fork(2)

`fork(2)` is a system call that creates a new process by duplicating the calling process. The new process is known as the child process, while the original process is known as the parent process. After the fork, both processes execute the same code, but each has a separate memory space.

Here is an example of using `fork(2)` to create a child process:

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

```c
#include <unistd.h>

int main()
{
	pid_t pid;

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

	if (pid == 0)
		printf("This is the child process. (pid: %d)\n", getpid());
	else
		printf("This is the parent process. (pid: %d)\n", getpid());

	return (0);
}
```

In the above program, `fork(2)` is used to create a child process. The child process prints "This is the child process." to the terminal, while the parent process prints "This is the parent process." to the terminal. Both processes have a different `pid` and both exit after printing.

## pipe(8)

`pipe(8)` creates a unidirectional data channel that can be used for interprocess communication. The data written to one end of the pipe can be read from the other end of the pipe. Pipes are often used in combination with `fork(2)` to create a communication channel between parent and child processes.

Here is an example of using `pipe(8)` to create a pipe and communicate between two processes:

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

int main()
{
	int fd[2];
	pid_t pid;
	char buffer[13];

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

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

	if (pid == 0)
	{
		close(fd[0]); // close the read end of the pipe
```

```
                write(fd[1], "Hello parent!", 13);
                close(fd[1]); // close the write end of the pipe
                exit(EXIT_SUCCESS);
        }
        else
        {
                close(fd[1]); // close the write end of the pipe
                read(fd[0], buffer, 13);
                close(fd[0]); // close the read end of the pipe
                printf("Message from child: '%s'\n", buffer);
                exit(EXIT_SUCCESS);
        }
}
```

In this code, `pipe(8)` is used to create a pipe, and `fork(2)` is used to create a child process. The child process writes the string "Hello, parent!" to the write end of the pipe using `write(2)`, and then exits. The parent process reads the string from the read end of the pipe using `read(2)`, and then prints it to the terminal using `printf(3)`. The pipe is then closed in both processes using `close(2)`.

## unlink(1)

`unlink(1)` is a command that removes a file from the file system. It takes a single argument, which is the path to the file to be removed.

Here is an example of using `unlink(1)` to remove a file called `example.txt`:

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

int main()
{
        if (unlink("example.txt") == 0)
                printf("File successfully deleted");
        else
                printf("Error deleting file");

        return (0);
}
```

In this code, `unlink(1)` is used to remove the file `example.txt` from the file system. `unlink(1)` returns `0` if all the files were deleted, and `-1` if an error occurred.

## wait(2)

`wait(2)` suspends the execution of its calling process until a child process terminates.

Here is an example of using `wait(2)` to wait until the 2s delay from the child process has terminated:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
        pid_t pid;
```

```c
	pid = fork();
	if (pid == -1)
	{
		perror("fork");
		exit(EXIT_FAILURE);
	}
	else if (pid == 0)
	{
		printf("I am the child process.\n");
		sleep(2);
		exit(EXIT_SUCCESS);
	}
	else
	{
		printf("I am the parent process.\n");
		wait(NULL);
		printf("Child process terminated after a 2s delay.\n");
	}

	return (EXIT_SUCCESS);
}
```

Notice that the message only appears after the child process' `sleep(1)` command has ended, and `exit(1)` is called, terminating its process.

# The Program

## Parsing

This is the first step of your program. It needs to make sure the input is correct, and how to handle unexpected data. This includes handling `here_doc`, opening the `infile` and `outfile`, and, if necessary, exiting the program (this includes closing all open FDs, freeing all memory, and using `unlink(1)` to remove the temporary file from `here_doc`, we'll get back to this later).

Your `main` function may follow a similar pattern to the below pseudocode. It's a simple structure, and provides you with the potential to exit the program at any given stage.

```
main()
{
	ft_init_pipex()
	ft_check_args()
	ft_parse_cmds()
	ft_parse_args()
	while (cmds)
		ft_exec()
	ft_cleanup()
}
```

`ft_init_pipex` is used to fill your struct with some default data, which may otherwise cause problems with Valgrind, as you may conditionally check the properties within your `struct` in your cleanup function.

I have a function `ft_check_args` which simply opens all files needed and handles `here_doc` as well as `/dev/urandom`. You should be able to get away with a custom `get_next_line` to complete them.

I recommend having two functions to parse and store the commands: one which will find the correct path using `envp` and store it in an array, and the other which will contain the arguments to the program. This will then help you build the required arguments for `execve(2)`.

In this scenario, `ft_parse_cmds` will create an array like this: `["/bin/cat", "/usr/bin/head", "/usr/bin/wc"]`, and the `ft_parse_args` will use `ft_split` to yield a 2D array like this one: `[["cat"], ["head", "-n", "5"], ["wc", "-l"]]` (remember to NULL terminate your arrays!). I recommend using a struct and storing all the data within it (see the struct I used below).

## Execution

The core idea of this program is to mimic the data flow between programs. Running the below scripts in your terminal *should* output the same data (you need to use double quotes for commands with arguments):

```
# input from a file

$ < Makefile cat | head -n 5 | wc -l > out
$ ./pipex Makefile cat "head -n 5" "wc -l" out

# input from `here_doc`

$ << EOF cat | head -n 5 | wc -l >> out
$ ./pipex here_doc EOF cat "head -n 5" "wc -l" out
```

The `ft_exec()` is where the magic happens. I'm not going to give you the code for it, because that would be too easy, so I will again write it in pseudocode:

```
ft_exec()
{
        pipe()
        fork()
        if (child)
        {
                dup2()
                execve()
        }
        else
        {
                close()
        }
}
```

After creating the pipe and creating the child process, the child runs the command and redirects the `stdout` from the command into the write end of the pipe using `dup2(2)`. The parent then 'catches' the output from the read end of the pipe, and outputs it back to the `stdin`. This is the main mechanism behind the relationship between parent and child process.

You will however have to have a unique `dup2(2)` call for both the first and last command, as they need to be redirected towards the input/output that the user requested.

You may have noticed you can't even use `printf(3)` as `dup2(2)` replaces `stdout` with other FDs. The best solution I found was to hijack my `ft_printf` and turn it into a `dprintf(3)`, which essentially does the same, except we can specify an FD, which we set to 2 (`stderr`).

### Cleanup

After all your code has executed (or if an unlucky `malloc(3)` failed), you will need to clean up all the open FDs, allocated memory, and potentially the temporary `here_doc` file. For this, it's best to allocate all memory within a `t_pipex` struct, and conditionally close/free whatever is necessary. As well the above mentioned, you will need to make sure you wait for any child process to terminate.

My struct looked like this, and it allowed me to exit the program at any stage, provided I had the struct within the scope of my function.

```c
typedef enum e_bool
{
	false,
	true
} t_bool;

typedef struct s_pipex
{
	int in_fd;
	int out_fd;
	t_bool here_doc; // use `int` if you prefer
	t_bool is_invalid_infile;
	char **cmd_paths;
	char ***cmd_args;
	int cmd_count;
} t_pipex;
```

## Common Mistakes

- Not using `unlink(1)` to remove temporary files.
- Using the wrong permissions when using `open(2)`. The `outfile` needs to be opened with different permission depending on whether or not `here_doc` was used.
- Not appending `NULL` to the end of `argv` in `execve(2)`. Doing so may lead to an invalid read.
- Not setting default values to your `struct`. This may lead to warnings from Valgrind (which shouldn't cause a fail) if you use these properties in a conditional check (`if`, `else`, `while`, etc…).
- Mishandling invalid commands. You may not face this issue depending on how you developed your program, but in mine, it was possible to get `NULL` in `cmd_paths`, due to the command being invalid. If that's your case too, it's not a problem, just make sure you know what you're doing.
- Not mimicking the behaviour of BASH. An invalid infile or command DOES NOT mean you exit the program.
- Special edge cases: `/dev/urandom` and `/dev/stdin`