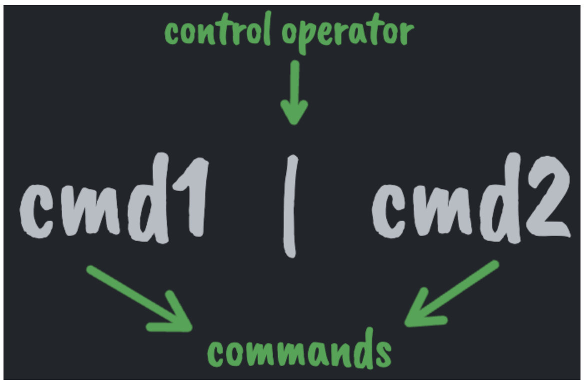
PIPEX

<u>SIMPLE DESCRIPTION OF THE PROJECT :</u>

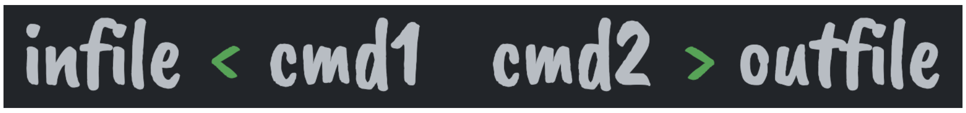
This project is a simulation of piping.

The pipeline is a sequence of one or more commands separated by one of the control operators.

example:



Which command need an <u>input</u>, after the command execution an <u>output</u> release. in this project the input will be the '***infile***' and the output will redirected to the '***outfile***'**:**



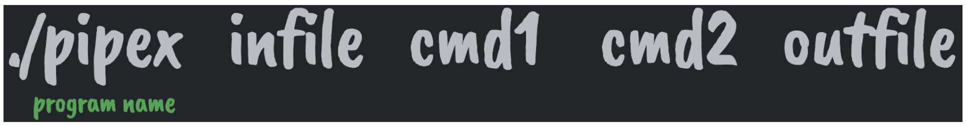
```
infile < cmd1   cmd2 > outfile
```

> The "<" symbol is used to **redirect** input from a file to a command.

> The ">" symbol is used to **redirect** output from a command to a file.

> **NB:** the ‘outfile’ will be created after the end of execution even if it wasn’t created

But in our project we’ll write that pipeline as arguments of our program:



```
./pipex  infile  cmd1  cmd2  outfile
```
program name

EXPLAINING THE ESSENTIAL FUNCTION OF THE PROJCT :

- **open()** & **close() & write() & read()**:

TAP HERE

- **access()** :

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

> checks the accessibility of the file named by the path argument for the access permissions indicated by the mode argument.

All the modes :

R_OK : for read permission

W_OK : for write permission

X_OK : for execute/search permission

F_OK : the existence test

NB: we could use 2 modes to check the accessibility of a file:

```
int access(const char *file_path, mode1 | mode2);
```

Return value:

0 : for successful completion

-1 : for error and and the global variable errno is set to indicate the error

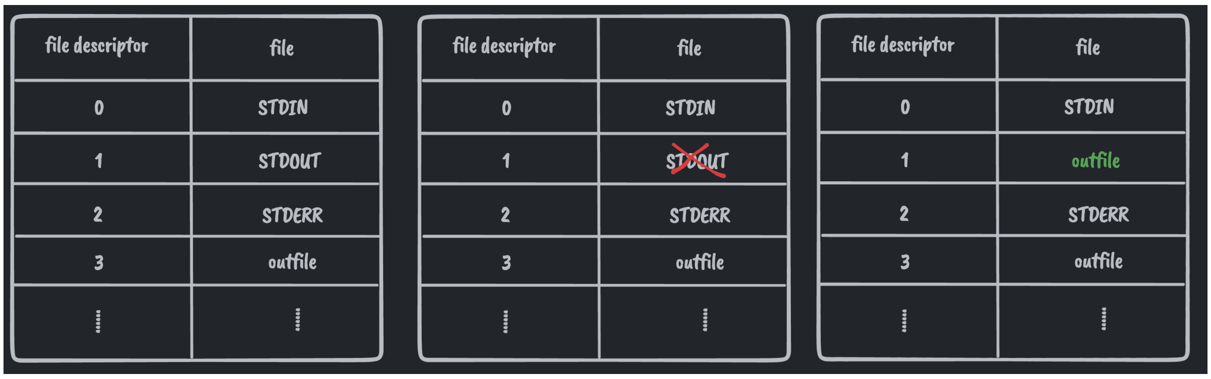
- **dup2()** :

```
  int dup2(old_fd, new_fd);
```

> duplicating an existing file descriptor. there's 2 processes happened:
>
> 1. close "**new_fd**" (if open)
> 2. copies the "**old_fd**" to the **fd[new_fd]**

EX:

```
dup2(outfile, STDOUT);
```



file descriptor	file
0	STDIN
1	STDOUT
2	STDERR
3	outfile
⋮	⋮

file descriptor	file
0	STDIN
1	~~STDOUT~~
2	STDERR
3	outfile
⋮	⋮

file descriptor	file
0	STDIN
1	outfile
2	STDERR
3	outfile
⋮	⋮

Return value:

new_fd value for successful completion

-1 : for error and and the global variable errno is set to indicate the error

- **fork()** :

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

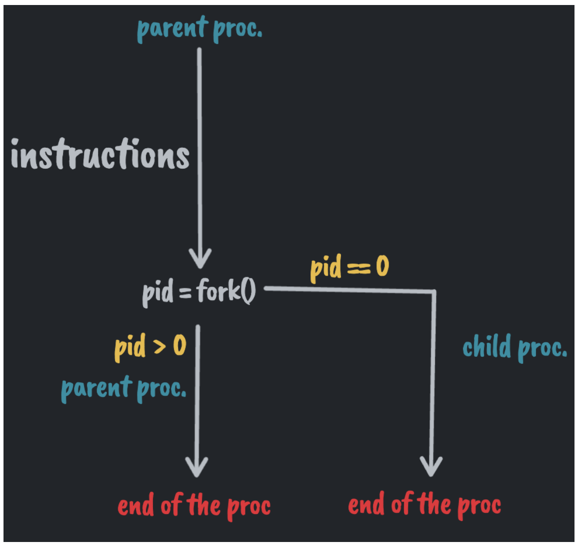
> This function causes creation of a new process called the **child process.** It's important to note that when a child process is created using `fork()`, a new process is created with its own memory space, and any subsequent changes made to the variables in either the parent or child process will not affect the other process. If you want to share data between the parent and child processes, you will need to use some form of inter-process communication such as pipes.

Return value:

1. **-1**: if an error occur
2. **0** : that means that you are in the child process
3. **>0** : that means you still at the parent process

```
int pid;

pid = fork();
if (pid == -1)
{
  // ERROR
  exit(EXIT_FAILURE);
}
if (pid == 0)
```



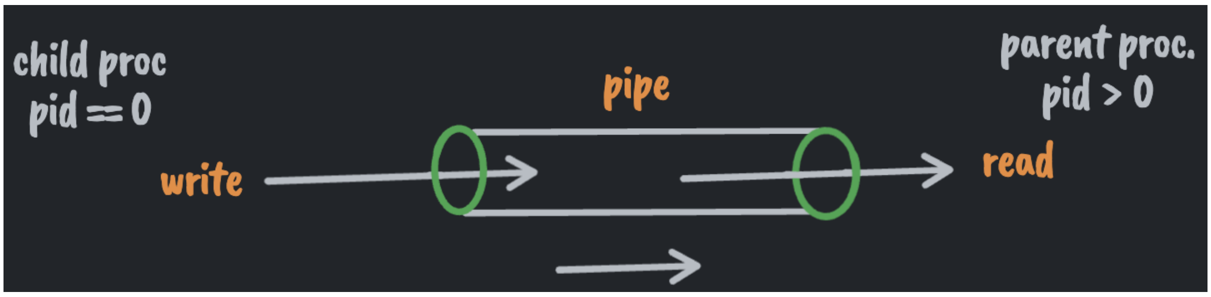
```
{
  // CHILD PROC.
}
else
{
  // PARENT PROC.
}
```

- **pipe()** :

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

> This function create a tunnel that make a connection of 2 processes, it takes an array of 2 elements and create a file descriptor for read end, also create a file descriptor for write end, these 2 files used to connect the child process with the parent process.
>
> fd[0] : read
>
> fd[1]: write

NB: these 2 files are not associated with files on disk, but rather with an in-memory buffer that is used for communication between the processes.



```
int fd[2];

if (pipe(fd) == -1)
  // ERROR
pid = fork();
if (pid == 0)
{
  // CHILD PROC.
  // close all the unused files descriptors
  close(fd[0]);
  int x = 5;
  write(fd[1], &x, sizeof(int));
  close(fd[1]);
}
else
{
  // PARENT PROC.
  // close all the unused files descriptors
  close(fd[1]);
  int y;
  read(fd[0], &y, sizeof(int));
  // y will take the sent value of the child proc.
  // y = 5;
  close(fd[0]);
}
```

- **wait() & waitpid()** :

> blocks the calling process until one of its child processes exits or a signal is received. After child process terminates,
> parent ***continues*** its execution after wait system call instruction.

TAP HERE

- **execve()** :

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

> this function execute the **cmds** that came in as argument (you should split it first), if the execution was successful, it will end the process or we can say that it transforms the process into a new one. for example if we want to print after **execve,** nothing will be printed.

```
  char *bin_path = "/bin/ls";
  char *args[] = {bin_path, "-a", "-l", NULL;

  execve(bin_path, args, NULL);
```

Return value :

-1 : error occur

- **unlink() :**

```
unlink("file_path");
```

> This function delete the file from file system (disk), but **not closing it**

RESOURCES :

- Youtube playlist
- Youtube (pipe and dup)