Pipes

Pipes are a mechanism provided by most operating systems for inter-process communication (IPC). They allow data to flow in one direction, from one process to another. In simple terms, pipes enable one process to send its output to another process as input. Pipes can be classified into two categories: **unnamed pipes** and **named pipes**.

Unnamed Pipes

Unnamed pipes are the simplest form of IPC. These pipes are created in memory and exist only as long as the processes communicating through them are alive. They are typically used for communication between a parent and its child process, as both processes share the same file descriptors.

- One-way communication: Data flows only in one direction, either from the parent process to the child or vice versa.
- Limited to related processes: Unnamed pipes work only between processes that share a common ancestor (e.g., a parent-child relationship).
- Temporary: They exist only while the processes communicating via the pipe are running.

Functions for using unnamed pipes:

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

Parameters:

- `pipefd`: An array of two integers where the file descriptors for the pipe are stored.
  - `pipefd[0]`: File descriptor for the read end of the pipe.
  - `pipefd[1]`: File descriptor for the write end of the pipe.

```
int close(int fd);
```

Close an end of a pipe. Since pipes are unidirectional, the processes must close the unused ends of the pipe.

```
ssize_t write(int fd, const void *buf, size_t count);
```

Parameters:

1. `fd`: The file descriptor to which data will be written. This file descriptor must be open for writing.
2. `buf`: A pointer to the buffer containing the data to be written. The data is copied from this buffer to the file descriptor.
3. `count`: The number of bytes to write from the buffer.

```
ssize_t read(int fd, void *buf, size_t count);
```

Parameters:

1. `fd`: The file descriptor from which data will be read. This file descriptor must be open for reading.
2. `buf`: A pointer to the buffer where the read data will be stored. The buffer should be large enough to hold the number of bytes specified by `count`.
3. `count`: The maximum number of bytes to read from the file descriptor.

Blocking Behaviour: If the pipe's buffer is empty (i.e., no data is available), the `read()` call will block (pause execution) until data is written to the pipe or the pipe is closed.

Named Pipes

Named pipes (also known as FIFOs, which stands for "First In, First Out") are similar to unnamed pipes but with key differences:

- Unrelated processes: Unlike unnamed pipes, named pipes allow communication between unrelated processes.
- Persistent: Named pipes persist in the filesystem even after the processes communicating through them terminate.
- Accessed via a filename: A named pipe is associated with a name in the filesystem, so unrelated processes can access it by name.

Functions for using named pipes:

```
int mkfifo(const char *pathname, mode_t mode);
```

Creates a named pipe.

Parameters:

1. **`pathname`**: The path where the FIFO will be created. This is the file name or file path where the named pipe will reside (e.g., `/tmp/my_fifo`).
2. **`mode`**: This specifies the **permissions** for the named pipe when it is created. The permission is given as an octal value (such as `0666`, `0644`, etc.) which tells the **rwx** permissions. We will use 0666 to give **rw-** (read and write) permissions to all users.

```
int open(const char *pathname, int flags);
```

Open an end of a pipe to read or write.

Parameters:

1. **`pathname`**: A string representing the named pipe you want to open. (e.g., `/tmp/my_fifo`).

2. `flags`: This argument specifies how the file should be opened (i.e., for reading or writing). Common flag options: **O_RDONLY**: Open the file for **read-only** access. **O_WRONLY**: Open the file for **write-only** access.

`read()` and `write()` are same as that in unnamed pipes.

Examples:
Go through the codes for:
Unnamed pipes: **unnamed.c**
Named pipes: **writer.c and reader.c**

Some Useful Functions:

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

`dup2()` is a system call in C used to duplicate a file descriptor, specifically allowing you to redirect one file descriptor to another, such as redirecting `stdin`, `stdout`, or `stderr` to a different file descriptor. It's commonly used for redirection in scenarios like setting up pipes between processes.

Parameters:
`oldfd`: The file descriptor you want to duplicate.
`newfd`: The file descriptor you want `oldfd` to be copied to. If `newfd` is already open, it will be closed before being reused.
FDs for stdin and stdout: STDIN_FILENO and STDOUT_FILENO respectively.

exec() family
The `exec` family of functions are used to execute a new program within the context of the current process. These functions replace the current process image with a new one. This means that after a successful call to an `exec` function, the original process image is gone, and the new program starts executing in its place. Hence if a process wants to execute a command using exec, it should create a child to execute it, so that it itself can continue after the exec.

1. `execl()`:

```
int execl(const char *pathname, const char *arg0, ..., NULL);
```

   Executes a program specified by `pathname`. The argument list must be terminated by a NULL pointer.
   Example: `execl("/bin/ls", "ls", "-l", NULL);`

2. execlp()

```
int execlp(const char *file, const char *arg0, ..., NULL);
```

Similar to `execl()`, but searches for the `file` in the directories listed in the `PATH` environment variable.
Eg: `execlp("ls", "ls", "-l", NULL);`

3. execvp()

```
int execvp(const char *file, char *const argv[]);
```

Similar to `execlp()`, but takes an argument vector (`argv`) instead of a list of arguments.
Eg:

```
char *args[] = {"ls", "-l", NULL};
execvp("ls", args);
```

4. execv()

```
int execv(const char *pathname, char *const argv[]);
```

Executes a program specified by `pathname` with the argument vector `argv`.
Eg:

```
char *args[] = {"ls", "-l", NULL};
execv("/bin/ls", args);
```