IPC and Pipes 3: FIFOs

1. Introduction and Basic Usage
2. Usage Details
3. **FIFOs**
   - **FIFOs (named pipes)**
   - **mkfifo() call and mkfifo command**
   - **FIFO usage**
   - **popen() and pclose()**

FIFOs (Named Pipes)

One of the limitations of basic, unnamed pipes is that they can be used only among *related* processes.

A **FIFO** or **named pipe** is a pipe that has an *entry in the filesystem*.

This allows the FIFO to be opened and used by *unrelated processes*.

As a result, FIFOs are a better choice than basic pipes for clients to communicate with a server in a *client-server application*.

However, FIFOs are still *unidirectional*, so they are a reasonable choice for client-server architectures only if clients send simple requests to a server without requiring any responses.

mkfifo Call

The call to create a FIFO is the library function `mkfifo`:

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

- `pathname` is the filesystem path for the FIFO
- `mode` are the permissions for the entry
- returns 0 or -1 on error

The actual syscall that gets invoked to create a FIFO is `mknod`:

```
int mknod(const char *pathname, mode_t mode, dev_t dev)
```

Direct calls to `mknod` to create FIFOs are discouraged.

FIFOs may also be manually created with the `mkfifo` *command*:

```
mkfifo [OPTION] NAME...
```

FIFO Usage

Once a FIFO exists (in the filesystem) a process gains access to it by calling `open()` (a FIFO is a type of **special file**).

Typically, a process will either want to read from the FIFO or write to it but not both, and one process will do one and another process do the opposite.

Thus, each process will open the file either `O_RDONLY` or `O_WRONLY`.

Since a FIFO will initially be opened only for reading or for writing (unlike a plain pipe), the semantics of `open()` for FIFOs is that `open()` will *block* until the same FIFO is opened by a different process for the opposite purpose.

FIFO Example

Process #1:

```
//Create FIFO:
mkfifo("/tmp/myfifo",0600);

//Open FIFO to read from:
int fifo_in = open("/tmp/myfifo",O_RDONLY);

char buff[11];
int nread = read(fifo_in,buff,10);
buff[nread] = '\0';  //Turn buff into a string
printf("FIFO read: %s\n",buff);

close(fifo_in);
```

Process #2:

```
//Open FIFO to write to:
int fifo_out = open("/tmp/myfifo",O_WRONLY);

write(fifo_out,"testing",7);

close(fifo_out);
```

popen() and pclose()

C provides a couple of special functions that allow one to simplify running another program in a subprocess with a pipe to the parent:

```
FILE *popen(const char *command, const char *type)
int pclose(FILE *stream)
```

- `command` is a command line one could give the shell
- `type` is `"r"` or `"w"`, depending on whether the parent wants to read from the pipe or write to it

`popen()` creates a pipe, forks off a subprocess, closes unused pipe ends, and exec's the shell in it, passing `command` to the shell.

`pclose()` waits for the subprocess to terminate and returns the exit status of the command.

popen() and pclose() (contd.)

Example executing `mychild` program using `popen()`:

```
//Start mychild program with pipe to send info to it:
FILE *progfp = popen("mychild norman 110","w");
...
fprintf(progfp,"info to send");
...
pclose(progfp);
```

popen() and pclose() (contd.)

Example executing `mychild` program using syscalls:

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

switch(fork()) {
  case -1:
    perror("fork failed");
    exit(EXIT_FAILURE);
  case 0:
    close(pipefd[1]);
    execlp(mychild,"mychild","norman","110",(char*)NULL);
    perror("execlp failed")
    exit(EXIT_FAILURE);
  default:
    close(pipefd[0]);
    ...
    write(pipefd[1],"info to send",12);
    ...
    wait(NULL);
  }
```