Operating Systems Lab (C+Unix)

Enrico Bini

University of Turin

Outline

1. Pipes
   - Recalling file descriptors
   - Pipes in C
   - Redirecting input/output via pipes

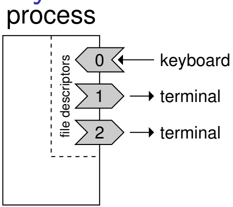
2. FIFOs

Outline

1. Pipes
    - Recalling file descriptors
    - Pipes in C
    - Redirecting input/output via pipes
2. FIFOs

File descriptors: indices of source/dest. of bytes

+1p .

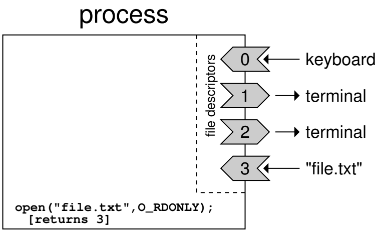


- file descriptor 0: read bytes from keyboard
- fd 1: write to terminal
- fd 2: write (errors) to terminal

- If a process opens a file by

```
open("file.txt",O_RDONLY);
```

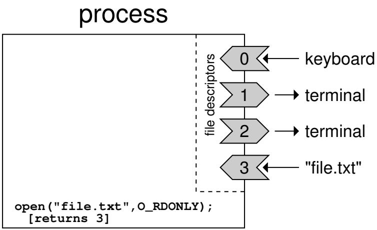
  then it gets a new file descriptor (3 in the example)



- the process can read from the file by specifying the file descriptor 3
- file descriptors identify sources/destinations of bytes
- closing a file descriptor by `close(...)` means:
  1. to cut the link between the file descriptor and what it is linked to
  2. to release the entry in the file descriptor table

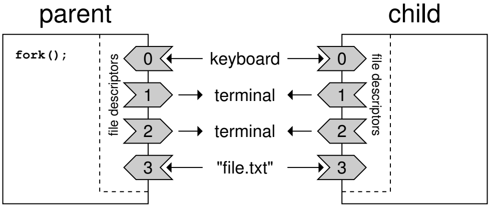
File descriptors: copied on `fork()`

- When a process forks a child, all file descriptors are copied
- Before `fork()`



```
open("file.txt",O_RDONLY);
  [returns 3]
```

- After `fork()`



```
fork();
```

- If a parent process closes any file descriptor, the child can still access (and viceversa)

Outline

1. Pipes
   - Recalling file descriptors
   - Pipes in C
   - Redirecting input/output via pipes
2. FIFOs

Pipes: the C interface

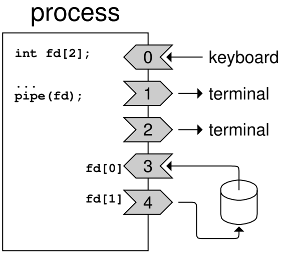
- Pipes are **uni-directional** byte streams
- Pipes are opened by

```
int pipefd[2]; /* declaring array of 2 int */

/* the call pipe sets two file descriptors */
pipe(pipefd);
```

- if successful (by returning 0) it opens two file descriptors in `pipefd`:
  - `pipefd[0]` is fd of the read end of the pipe
  - `pipefd[1]` is fd of the write end of the pipe

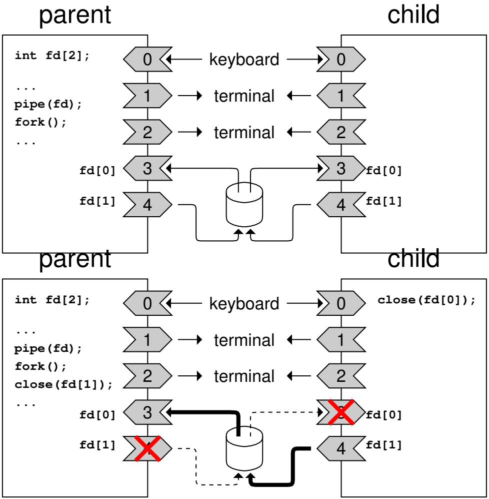
  anything that is written to `pipefd[1]` can be read from `pipefd[0]`



Two processes communicating via pipe

- When a process forks a child after creating a pipe, a communication channel between parent and child is created

- If the two processes close the unused file descriptor, a uni-directional channel is created



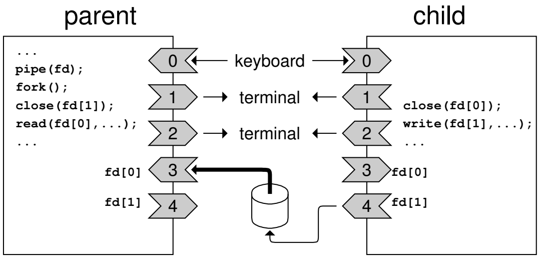
```
int fd[2];
...
pipe(fd);
fork();
...
```

```
int fd[2];
...
pipe(fd);
fork();
close(fd[1]);
...
```

```
close(fd[0]);
```

- If unused file descriptors are not closed, then we run into problems (explained later)

Reading from a pipe

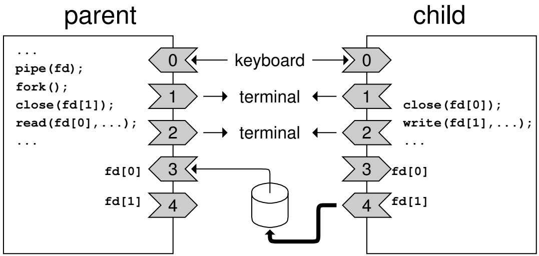


```
char buf[100];   /* stores bytes read from pipe */
int num_bytes;

num_bytes = read(pipefd[0], buf, sizeof(buf));
```

- Reading consumes the data, which will be unavailable for next `read()`
- After a `read(...)` from a pipe:
  - if data is present, it is stored in `buf`, returned number of read bytes
  - if no data and some write end is open, it waits for some writes
  - if no data and no write end is open, it returns zero

Writing to a pipe



```
...
pipe(fd);
fork();
close(fd[1]);
read(fd[0],...);
...
```

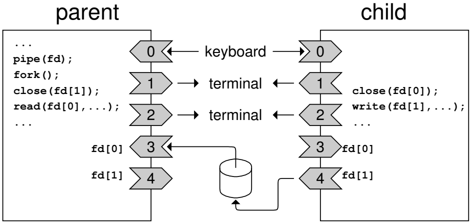
```
close(fd[0]);
write(fd[1],...);
...
```

```
char buf[100]; /* stores bytes written to pipe */
int num_bytes;

num_bytes = write(pipefd[1], buf, sizeof(buf));
```

- After a `write(...)` to a pipe:
  - if the pipe is full, the process waits for some `read(...)`
  - if enough space, it returns the number of bytes written
  - if no read end is open, a signal `SIGPIPE` (default action: `Term`) is generated (to notify that the written data will never be read)

Necessary to close unused file descriptors of pipes



```
...
pipe(fd);
fork();
close(fd[1]);
read(fd[0],...);
...
```

```
close(fd[0]);
write(fd[1],...);
...
```

- file descriptors of **unused write ends must be closed**
  - The “end-of-file” value is returned to the reader (`read()` returning 0) only when the **last** file descriptor of any writer is closed
  - if the write ends of a pipe are not closed, then the reader will wait on `read()` believing the some writer will `write()`
- file descriptors of **unused read ends must be closed**
  - When a writer tries to `write()` to a fd where all the readers have closed their read end, it gets a `SIGPIPE` signal
  - if some read end is left open, the signal `SIGPIPE` is not sent and the writer believes that somebody will read its data

Writing/Reading via pipes: examples

- Normally, the **writer** decides that a pipe is no longer needed
  1. the writer closes its write end
  2. the reader reads all the data until `read(...)` returns zero
- The size of the pipe is `PIPE_BUF` (4096 bytes on my machine):
  - reading/writing data not greater than `PIPE_BUF` is **atomic**
- If a process is waiting on `read(...)` or `write(...)` and it gets a signal, it returns `-1` and `errno` is set to `EINTR`
- Examples of pipe usage
  - *test-pipe-single.c*, single writer, single reader
  - *test-pipe-kids.c*, many writers, single reader, comment atomicity

Outline

1. Pipes
   - Recalling file descriptors
   - Pipes in C
   - Redirecting input/output via pipes
2. FIFOs

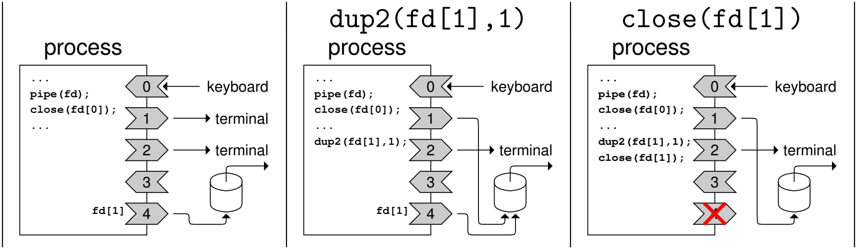
Copying a file descriptor onto another

- It is possible to "copy" a file descriptor onto another one (which is then overwritten)

```
int dup2(int fd_src, int fd_dst);
```

  which copies `fd_src` onto `fd_dst`. If `fd_dst` was previously open, then `dup2()` also close it.

- Example of redirecting stdout to another process via pipe



dup2(fd[1],1)

close(fd[1])

```
...
pipe(fd);
close(fd[0]);
...
```

```
...
pipe(fd);
close(fd[0]);
...
dup2(fd[1],1);
```

```
...
pipe(fd);
close(fd[0]);
...
dup2(fd[1],1);
close(fd[1]);
```

- whatever the process sends to fd 1 (`printf(), ...`) goes to the pipe
- *`test-dup2-simple.c`*

Input/output redirection from command line

- What happens when it is launched the following command?

```
ps -Af | wc -l > num_proc
```

- The shell (`bash` for example) is responsible for parsing the command line and mixing the ingredients properly

  1. It creates two processes: PID1 and PID2
  2. It attaches the output of PID1 to the input of PID2
  3. It attaches the output of PID2 to the file `num_proc`
  4. It makes PID1 execute the command `ps -Af` with `execve`
  5. It makes PID2 execute the command `wc -l` with `execve`

- The two process are not aware of the presence of the pipe. It is the shell (their parent process) which connected the streams differently

Getting input/output from another command

- By the system call

```
FILE * popen(const char *command, "r");
```

it is possible to:

1. fork a new process
2. attach (by pipe) the `stdout` stream of `command` to the returned stream
3. invoke the command

- Analogously, by the system call

```
FILE * popen(const char *command, "w");
```

it is possible to

1. fork a new process
2. attach (by pipe) the `stdin` stream of `command` to the returned stream
3. invoke the command

- this type of streams must be closed by

```
int pclose(FILE *stream);
```

which also waits for the child process created by `popen`

`test-popen.c`

Outline

1. Pipes
   - Recalling file descriptors
   - Pipes in C
   - Redirecting input/output via pipes
2. FIFOs

Pipes and Named pipes (FIFOs)

- Pipes are identified by file descriptors: they can be used only among processes sharing an ancestor
- Named pipes, called FIFO (First In First Out), solves this issue
- FIFOs are pipes with a global visible name in the file system
- Any process knowing the name of the FIFO can access to it

FIFO

1. Open two terminals: terminal A and terminal B, both well visible on the screen
2. term A: `mkfifo my-1st-fifo`
3. term A: `ls -latr`, you can notice "p" in the 1st column
4. term A: `ls > my-1st-fifo`, to write something to the pipe
5. term B: `cat my-1st-fifo`, to print the content of `my-1st-fifo`

- the last two commands can also be exchanged
- try with two terminals doing `cat my-1st-fifo`, and then one doing `ls > my-1st-fifo`
- Comments: the write blocks until some process reads and viceversa
- In C, a FIFO can be created by

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

which creates a FIFO with at `pathname`, with read/write/execute permissions as specified by `mode`