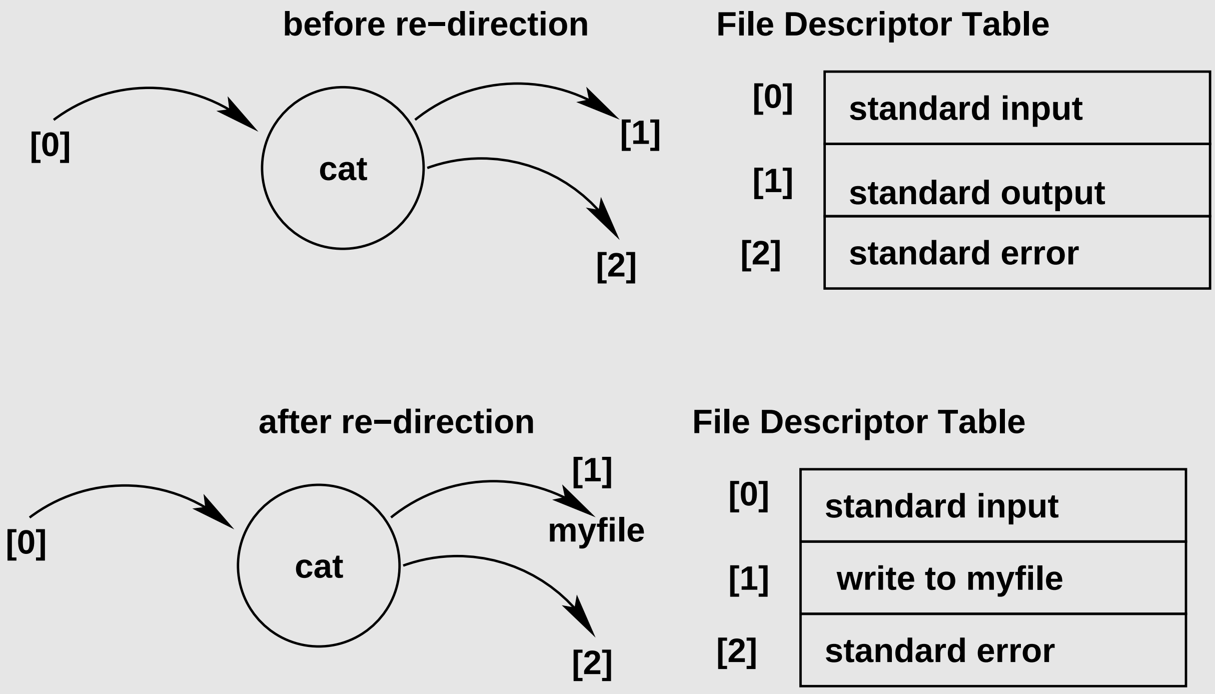
IPC - Pipes and FIFOs

Prof. Himanshu B. Dave
e-Infochips, Ahmedabad

June, 2010 / e-Infochips, Ahmedabad

Redirection in command

```
cat > myfile
```



File Descriptor Table

[0]	standard input
[1]	standard output
[2]	standard error

File Descriptor Table

[0]	standard input
[1]	write to myfile
[2]	standard error

Figure 1

Redirection in command

To achieve the same thing in a C program:

- open myfile to establish an entry in System File Table
- copy the pointer to this entry into entry for STDOUT
- to do this use dup2() function

See program `ex3_17.c`

Redirection in command

```
#include <unistd.h>
dup2(int filedesc, int filedesc2);
```

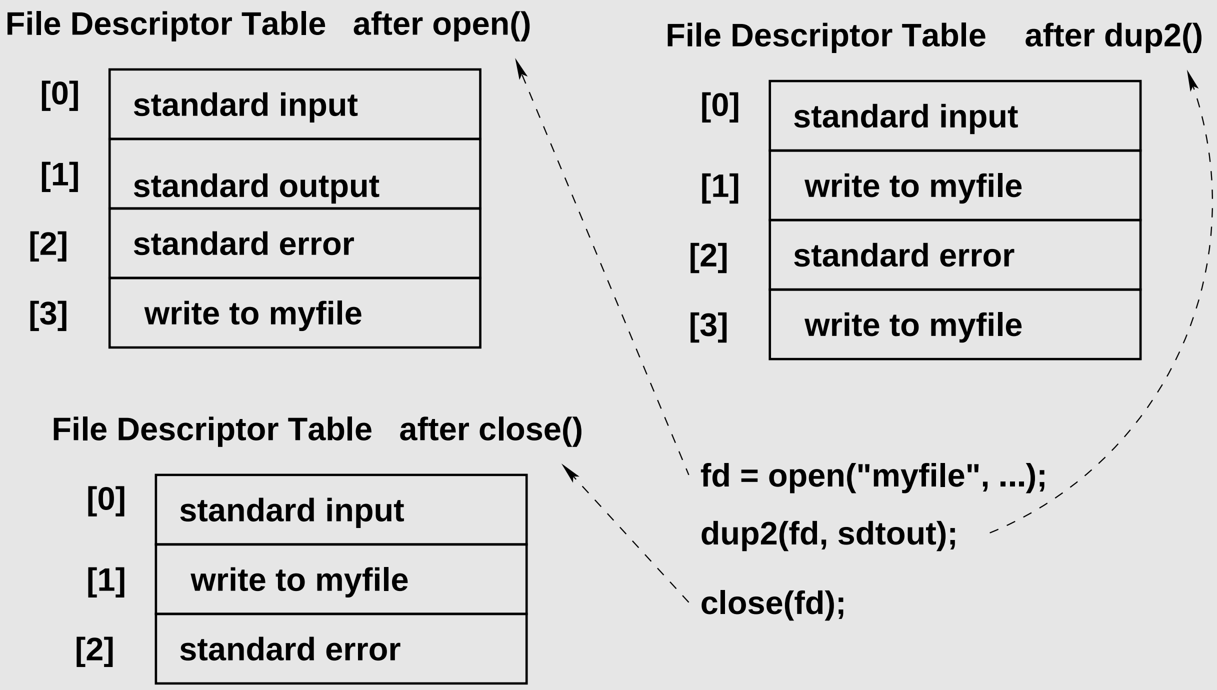
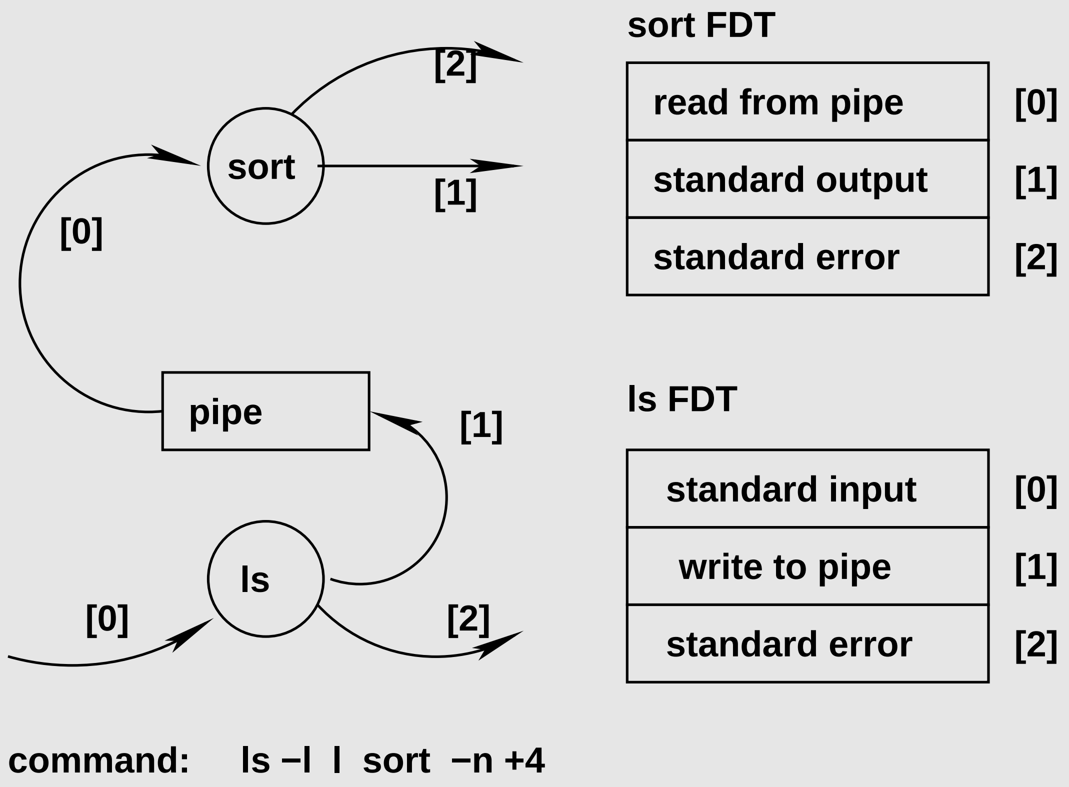


Figure 2

Pipe in a command line

```
ls -l | sort -n +4
```



sort FDT

read from pipe	[0]
standard output	[1]
standard error	[2]

ls FDT

standard input	[0]
write to pipe	[1]
standard error	[2]

Figure 3

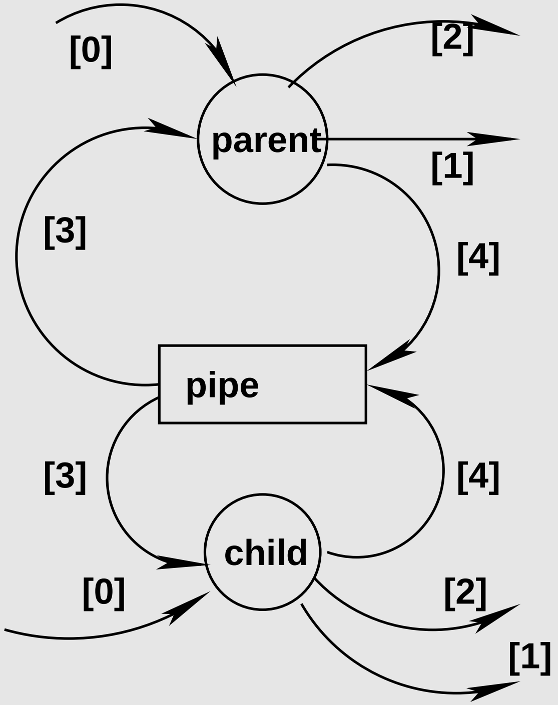
Commands connected via Pipe

To achieve the same thing in a C program:

- open a pipe;
- use fork() to create two processes : a child and a parent
- in Child: use dup2() to connect STDOUT to pipe write-end
- then run ls
- in Parent: use dup2() to connect STDIN to pipe read-end
- then run sort

See program `ex3_20.c`

Pipe in a C program



Re−direction in program:

after execution of fork()

Parent FDT

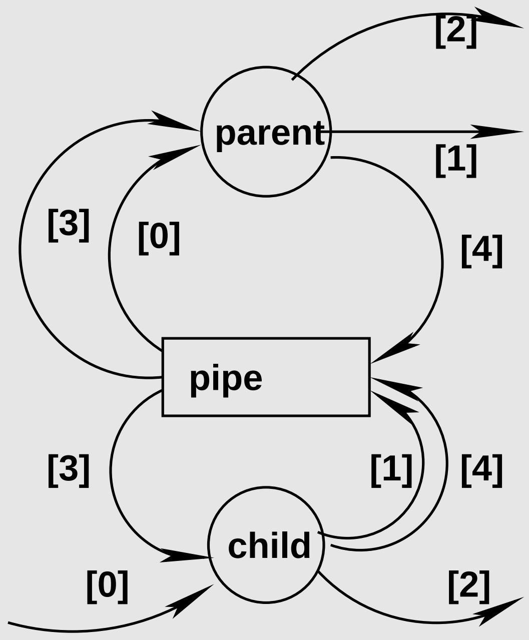
standard input	[0]
standard output	[1]
standard error	[2]
read from pipe	[3]
write to pipe	[4]

Child FDT

standard input	[0]
standard output	[1]
standard error	[2]
read from pipe	[3]
write to pipe	[4]

Figure 4

Pipe in a C program



Parent FDT

read from pipe	[0]
standard output	[1]
standard error	[2]
read from pipe	[3]
write to pipe	[4]

Child FDT

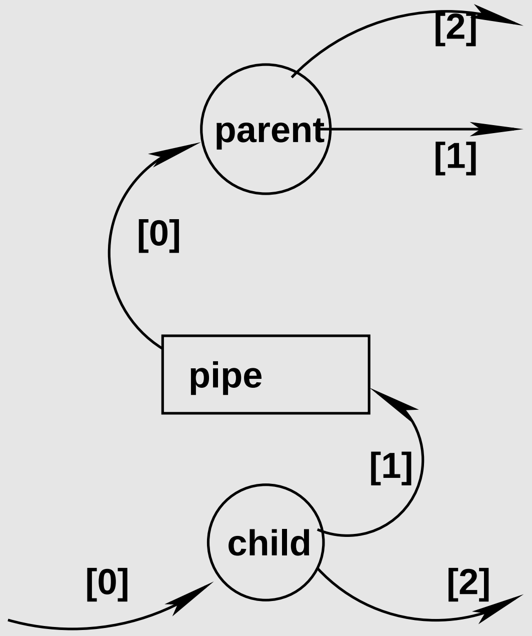
standard input	[0]
write to pipe	[1]
standard error	[2]
read from pipe	[3]
write to pipe	[4]

Re−direction in program:

after execution of dup2() in both

Figure 5

Pipe in a C program



Parent FDT

read from pipe	[0]
standard output	[1]
standard error	[2]

Child FDT

standard input	[0]
write to pipe	[1]
standard error	[2]

Re−direction in program:

just before execution of exec()

Figure 6

Assignment 1

Application of pipes:

We want to invoke the Linux standard "sort" utility from within our program. One way to do that would be to have:

```
system("sort <data >output");
```

within our program. Remember that doing so would run "sort" in a separate shell. We do not want to do that.

Assignment 1: contd.

We want to pipe the data to be sorted to "sort", and then make "sort" pipe the sorted results back to us.
Since "sort" can read and write on `stdin` and `stdout` respectively (it is a filter, in Unix terminology), this should be posible to achieve.
Basically, we should be able to force an arbitrary file-descriptor as `stdin` and `stdout` on "sort".

Assignment 1: contd.

In this exercise, we want to send the contents of a text file (one word per line, unsorted) to "sort" and let it return to us the sorted list, which our program should print out.
First try to write a C program with only one pipe, which will do this and report your results. The program is not expected to do the job prperly (why?).
Then write an improved program with two pipes and make it work.

Assignment 2

Pipes application and resolution of Dead-lock.

In this exercise we want to set up an interactive *wrapper* for the standard Linux `ed` text editor program. The wrapper will be the parent invoking the `ed` as a child. It will send editor commands to `ed` and get back the reults. The two together should search a given text file for lines containing given patterns.

Assignment 2: contd.

A typical session is:

```
$ search
File? testdata.txt
Search pattern? ^a
apple
apricot
Search pattern? apple
apple
pineapple
Search pattern? O$
tomato
mango
Search pattern? CTRL-D
$
```

Assignment 2: contd.

Remember that, unlike **`sort`**, which reads all its data in one go, **`ed`** is interactive. You will have to figure out how to handle this.
It is strongly suggested that you play with **`ed`** (which behaves like a stripped-down version of vi) to get familliar with it.
First read its man-page, you may need to refer to it.

Assignment 2: contd.

One problem you will face is: how to detect that `ed` has completed its response to a command sent by your wrapper?
You will have to figure this out, as this is the basic synchronization requirement.

- (a) Get famillar with ed.
- (b) Write the wraper to send and get executed one command only and get the result back to your wrapper.
- (c) Extend it for continuous interactive working, solving any sync problems that may arise.

Problems with pipes

They can be used only between processes belonging to a family tree i.e., parent-child or sibling relationship should exist.

Fifo - basic characteristics

Fifo: first-in first-out special file, named pipe

- accessed as part of the file system
- can be opened by multiple processes for reading or writing
- must be opened on both ends (reading and writing) before data can be passed.
- opening the FIFO blocks until the other end is opened also (normal working)
- command mkfifo or C-library function mkfifo() is used to create a fifo in the file system.

Fifo - example, step 1

Setting up a fifo:
use mkfifo command or mkfifo()
see `fifo1.c`

Fifo - example, step 2

Preparing for i/o:
use open() to initiate read/write to the fifo
see `fifo2.c`

Fifo - example, step 3

A simple file-server using fifo's

see `fifo3.c`

Assignment 3, 4, 5:

- 3. Change our simple "file-server" so that there are two independent programs - **`fsclient`** and **`fserver`** - communicating via fifo's.
- 4. Convert the **`sortinvoke`** to utilize fifo's instead of pipes.
- 5. Convert the **`edinvoke`** to utilize fifo's instead of pipes.

Some example programs

`unpv22e/`

- `pipe/mainpipe.c`
- `pipe/client.c`
- `pipe/server.c`
- `pipe/mainfifo.c`
- `pipe/server_main.c`
- `pipe/client_main.c`
- `fifocliserv/mainserver.c`
- `fifocliserv/mainclient.c`

Reference books

- "Practical Unix Programming" Robbins K. A. and Robbins Steven
- "Advanced Unix Programming", 2nd Ed. Marc Rochkind