Unix Special Files & IPC

By
Dr.Trilok Nath Pandey

SOA, Deemed to be University,
ITER, Bhubanewar

Book(s)

Text Book(s)

Kay A. Robbins, & Steve Robbins

Unix™ Systems Programming

Communications, concurrency, and Treads

Pearson Education

Reference Book(s)

Brain W. Kernighan, & Rob Pike

The Unix Programming Environment

PHI

Introduction

- Two other important examples of special files are pipes and FIFOs

- Pipes and FIFOS are the interprocess communication mechanisms that allow processes running on the same system to share information and hence cooperate.

Pipes

```
#include <unistd.h>

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

```
Returns:

(1) If successful, pipe returns 0.

(2) If unsuccessful, pipe returns -1 and sets errno.
```

- The simplest UNIX interprocess communication mechanism is the pipe, which is represented by a special file.
- The **pipe** function creates a communication buffer that the caller can access through the file descriptors **fd[0]** and **fd[1]**.
- The data written to **fd[1]** can be read from **fd[0]** on a first-in-first-out basis.

About `pipe()`

- A pipe has no external or permanent name, so a program can access it only through its two descriptors.
- For this reason, a pipe can be used only by the process that created it and by descendants that inherit the descriptors on **`fork`**.
- The pipe function described here creates a unidirectional communication buffer.

The following code segment creates a pipe.

```
int fd[2];

if (pipe(fd) == -1)
   perror("Failed to create the pipe");
```

If the **`pipe`** call executes successfully, the process can read from **`fd[0]`** and write to **`fd[1]`**.

`pipe()` Returns

- When a process calls **`read`** on a pipe, the **`read`** returns immediately if the pipe is not **empty**.
- If the pipe is empty, the **`read`** blocks until something is written to the pipe, as long as some process has the pipe open for writing.
- On the other hand, if no process has the pipe open for writing, a **`read`** from an empty pipe returns 0, indicating an end-of-file condition

Pipelines

- Pipeline describes how to use redirection with pipes to connect processes together.
- A process can redirect standard input or output to a file. The following commands use the sort filter in conjunction with **`ls`** to output a directory listing **sorted by size**.

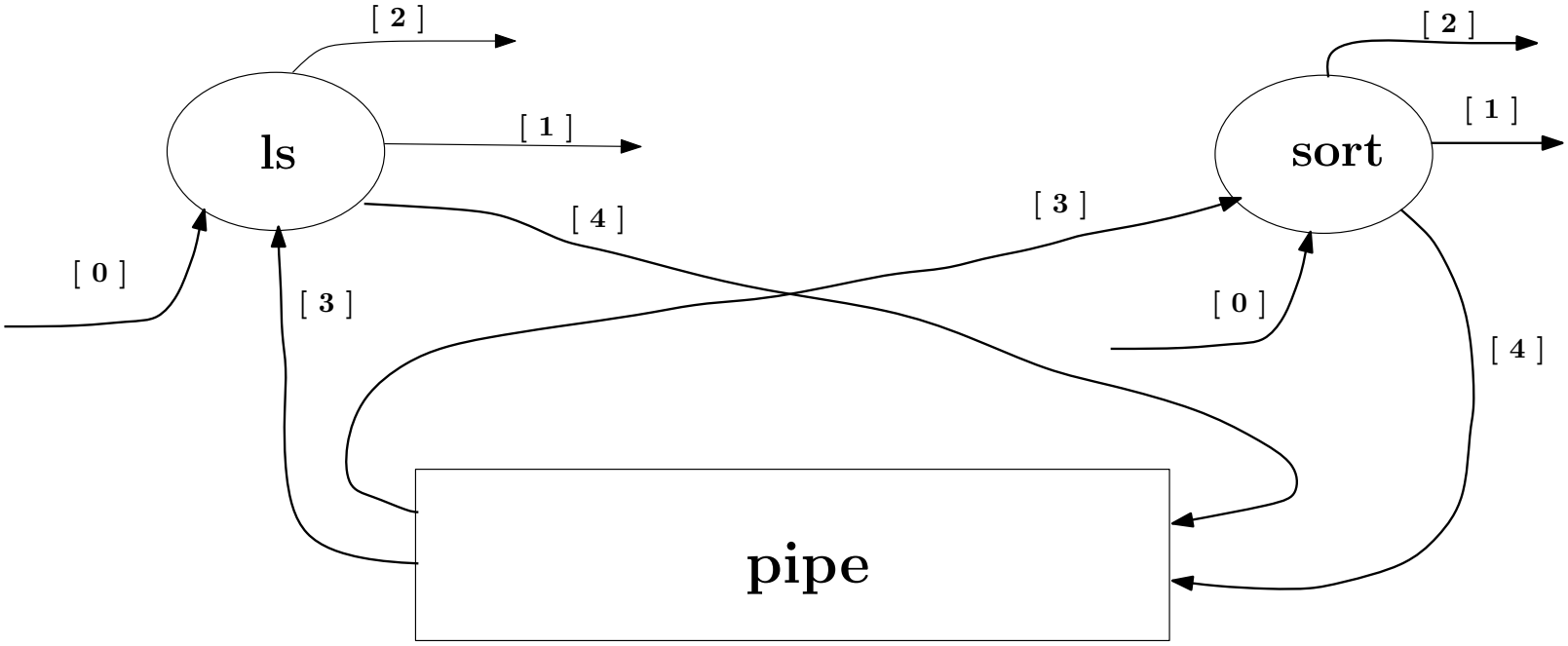
```
   ls -l > temp
sort -n  < temp
```

- An alternative approach for outputting a sorted directory listing is to use an interprocess communication (IPC) mechanism such as a **`pipe`** to send information directly from the **`ls`** process to the sort process.

```
   ls -l |  sort -n
```

- A programmer can build complicated transformations from simple filters by feeding the standard output of one filter into the standard input of the other filter through an intermediate pipe. The pipe acts as a buffer between the processes,

Redirection Usages: `ls -l | sort -n`



fd[0]
Pipe read

fd[1]
Pipe write

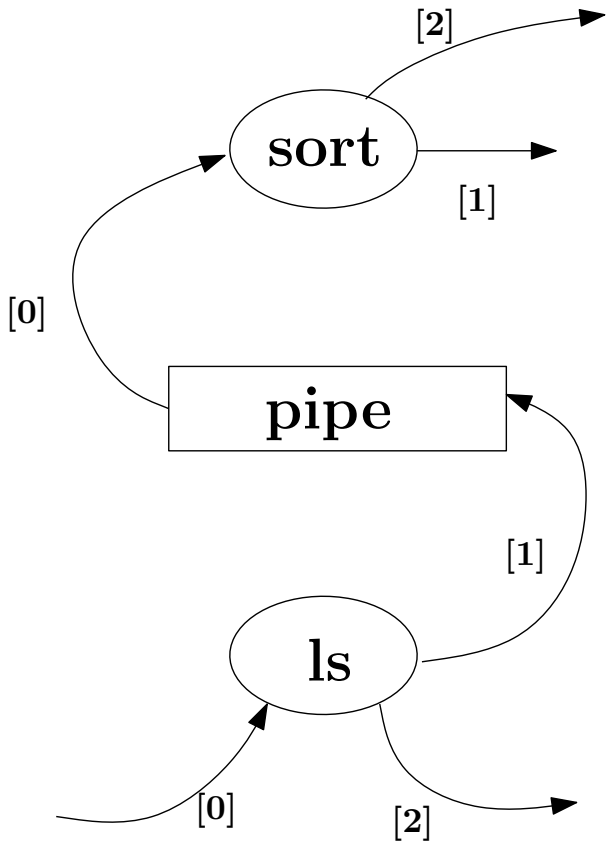
ls

	file descriptor table
[0]	`standard` *input*
[1]	`standard` *output*
[2]	`standard` *error*
[3]	`pipe` *read*
[4]	`pipe` *write*

sort

	file descriptor table
[0]	`standard` *input*
[1]	`standard` *output*
[2]	`standard` *error*
[3]	`pipe` *read*
[4]	`pipe` *write*

Execution: `ls -l | sort -n`



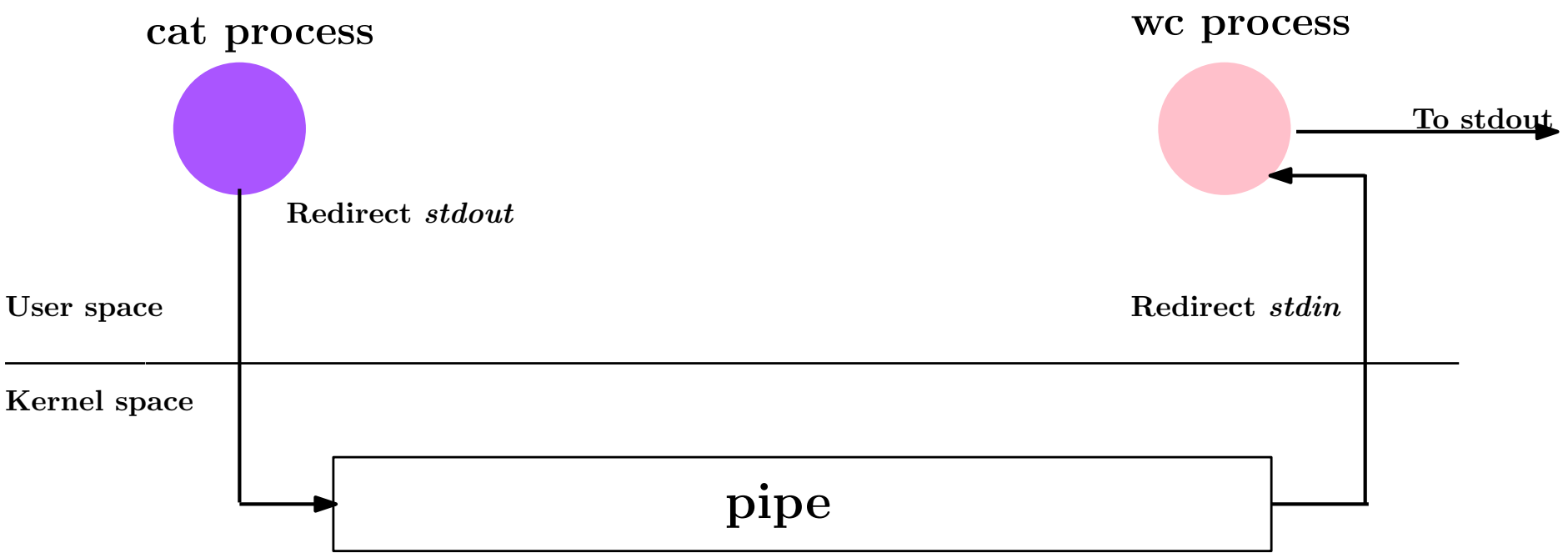
sort

	file descriptor table
[0]	`pipe` *read*
[1]	`standard` *output*
[2]	`standard` *error*

ls

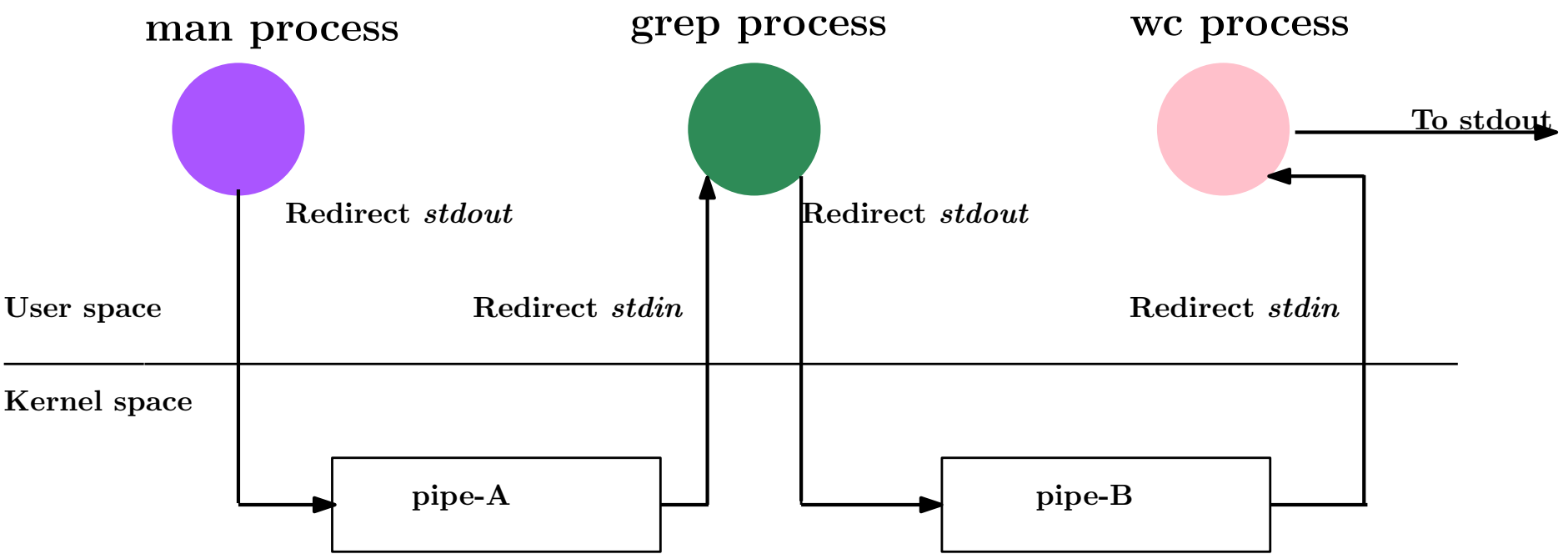
	file descriptor table
[0]	`standard` *input*
[1]	`pipe` *write*
[2]	`standard` *error*

Pipeline: `cat test.txt | wc -l`




```
cat test.txt | wc -l
```

Pipeline: `man ls | grep ls | wc -l`




```
man ls | grep ls | wc -l
```

Pipe Limitations

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have limitations.

- Half duplex (i.e., data flows in only one direction).
- Pipe can be used only between processes that have a common ancestor.
- Normally, a pipe is created by a process, that process calls **`fork`**, and the pipe is used between the parent and the child.

FIFOs get around the related and unrelated processes.

FIFO: Named Pipe

- Pipes are temporary. They disappear when no process has them open.
- POSIX represents FIFOs or named pipes by special files that persist even after all processes have closed them.
- FIFO unidirection. It can be used for both related and unrelated processes.
- A FIFO has a name and permissions just like an ordinary file and appears in the directory listing given by **`ls`**
- A FIFO is created by executing the **`mkfifo`** command from a shell or by calling the **`mkfifo`** function from a program.

Note: FIFO– Named Pipe

- fifo - first-in first-out special file, named pipe.
- A FIFO special file (a named pipe) is similar to a pipe, except that it is accessed as part of the filesystem.
- It can be opened by multiple processes for reading or writing.
- When processes are exchanging data via the FIFO, the kernel passes all data internally without writing it to the filesystem.
- Thus, the FIFO special file has no contents on the filesystem; the filesystem entry merely serves as a reference point so that processes can access the pipe using a name in the filesystem.
- The kernel maintains exactly one pipe object for each FIFO special file that is opened by at least one process. The FIFO must be opened on both ends (reading and writing) before data can be passed. Normally, opening the FIFO blocks until the other end is opened also.

Shell Command `mkfifo`

```
$ mkfifo filefifo          /* To create FIFO file */

$ ls -l filefifo

output::
------
prw-rw-r-- 1 abc abc 0 Mar  6 12:12 filefifo

/* The first character p represents the FIFO file
   . */
```

Shell Command to remove `fifo` File

```
$ rm FiFOfilename
```

mkfifo System Call

```
#include <sys/stat.h>

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

```
Returns:

(1) If successful, mkfifo returns 0.

(2) If unsuccessful, mkfifo returns -1 and sets errno.

(3) A return value of -1 means that the FIFO was not
    created.
```

Code Segment to Create a FIFO

```
#define FIFO_PERMS (S_IRUSR | S_IWUSR |
    S_IRGRP | S_IROTH)

if (mkfifo("myfifo", FIFO_PERMS) == -1){
    perror("Failed to create myfifo");
    return 1;
}
```

The above code segment creates a **FIFO**, `myfifo`, in the current working directory. This **FIFO** can be read by everybody but is writable only by the owner.

Communication Between 2 unrelated processes

Program 1 : fifowriter.c

```
#include<stdio.h>
#include<unistd.h>
#include<sys/stat.h>
#include<fcntl.h>
int main()
{
 int fd;
 mkfifo("temp.txt",0600);
 fd=open("temp.txt",O_WRONLY);
 write(fd,"Writer\n",7);
 close(fd);
 return 0;
}
```

Communication Between 2 unrelated processes

Program 2 : fiforeader.c

```
#include<stdio.h>
#include<unistd.h>
#include<sys/stat.h>
#include<fcntl.h>
int main()
{
 int fd;
 char buf[20];
 fd=open("temp.txt",O_RDONLY);
 read(fd,buf,7);
 write(1,buf,7);
 close(fd);
 unlink("temp.txt");
 return 0;
}
```

Communication Between 2 related processes

The parent reads what its child has written to a named pipe.

```
int main (int argc, char *argv[]) {
pid_t childpid;int fd,fd1;char buf[20];
if (argc != 2) {      /* command line has pipe name */
   fprintf(stderr, "Usage: %s pipename\n", argv[0]);
   return 1;
}
mkfifo(argv[1], 0600); /* create a named pipe */
childpid = fork();
if (childpid == 0){           /* The child writes */
   fd=open(argv[1],O_WRONLY);
   write(fd,"I am child\n",11);
}
else{
   fd1=open(argv[1],O_RDONLY);
   read(fd1,buf,11);
   write(1,buf,11);wait(NULL);
   unlink(argv[1]);
}
return 0;
}
```