Exercises - Solutions
Input / output

Computadors

Grau en Ciència i Enginyeria de Dades

Xavier Verdú, Xavier Martorell

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC)

2020-2021 Q2

Creative Commons License

This work is under a Creative Commons Attribution 4.0 Unported License

The details of this license are publicly available at
https://creativecommons.org/licenses/by-nc-nd/4.0

Exercise 1

- Is there any performance difference comparing both codes?

CODE A

char buf[1024];

CODE B

```
while ((len = read(0, buf, 1)) > 0)
    write(1, buf, len);
```

```
while ((len = read(0, buf, 1024)) > 0)
    write(1, buf, len);
```

- And comparing these ones?

```
char buf[1024*1024];
```

```
while ((len = read(0, buf, 1024)) > 0)
    write(1, buf, len);
```

```
while ((len = read(0, buf, 2048)) > 0)
    write(1, buf, len);
```

Exercise 1

- Usually, performing a high number of system calls degrades performance, due to the overhead of entering and leaving the OS
  - As a consequence, code B is actually performing better to implement read's of 1024 bytes, when possible

CODE B

```
char buf[10
while ((len = read(0, buf, 1024)) > 0)
      write(1, buf, len);
```

- And if you have large data to read, e.g., 1MB, it is better to use 2048 bytes or even a larger chunk size at a time, rather than a smaller chunk size

```
char buf[1024*1024];
while ((len = read(0, buf, 2048)) > 0)
      write(1, buf, len);
```

Exercise 2

- What are we doing with the following code snippet?

```
…
pid = fork();
if (pid == 0){
    close(0);
    open("/dev/deviceA", O_RDONLY);
    close(1);
    open("/dev/deviceB", O_WRONLY);
    execlp("./myprog", "./myprog", NULL);
}
…
```

Hint: can you write it in a shell command syntax using channels redirections?

Exercise 2

- What are we doing with the following code snippet?

```
…
pid = fork();
if (pid == 0){
     close(0);
     open("/dev/deviceA", O_RDONLY);
     close(1);
     open("/dev/deviceB", O_WRONLY);
     execlp("./myprog", "./myprog", NULL);
}
…
```

It implements a common I/O redirection to start «myprog»

It would be equivalent to

```
$ ./myprog  >/dev/deviceB  </dev/deviceA
```

Exercise 3

- Comment – advantages and inconveniences - about this code snippet

```
…
pid = fork();
if (pid == 0){
    close(0);
    open("/dev/deviceA", O_WRONLY);
    close(1);
    open("/dev/deviceB", O_RDWR);
    execlp("./myprog", "./myprog", NULL);
}
…
```

Hint: can you write it in a shell command syntax using channels redirections?

Exercise 3

- Comment – advantages and inconveniences - about this code snippet

```
…
pid = fork();
if (pid == 0){
      close(0);
      open("/dev/deviceA", O_WRONLY);
      close(1);
      open("/dev/deviceB", O_RDWR);
      execlp("./myprog", "./myprog", NULL);
}
…
```

We are redirecting I/O in such a way that channel 0 will be opened for writing and channel 1 will be opened for read/write. It is not common, so it cannot be used to spawn usual programs (cat, wc, more, less...), but it may fit with the needs of «myprog»

Exercise 4

- Assuming the device /dev/urandom generates pseudorandom bytes, implement a code to:
  - read 100 random bytes from this device
  - write those values to the stdout using integer format (i.e. “int”)
  - write those values to the stderr using ASCII format
    - **HINT:** sprintf(buf,“%d”, num); //converts a number to ASCII format

- Redirect both outputs (stdout and stderr) to different output files
  - Compare both output files with “xxd” command as well as their size

Exercise 4

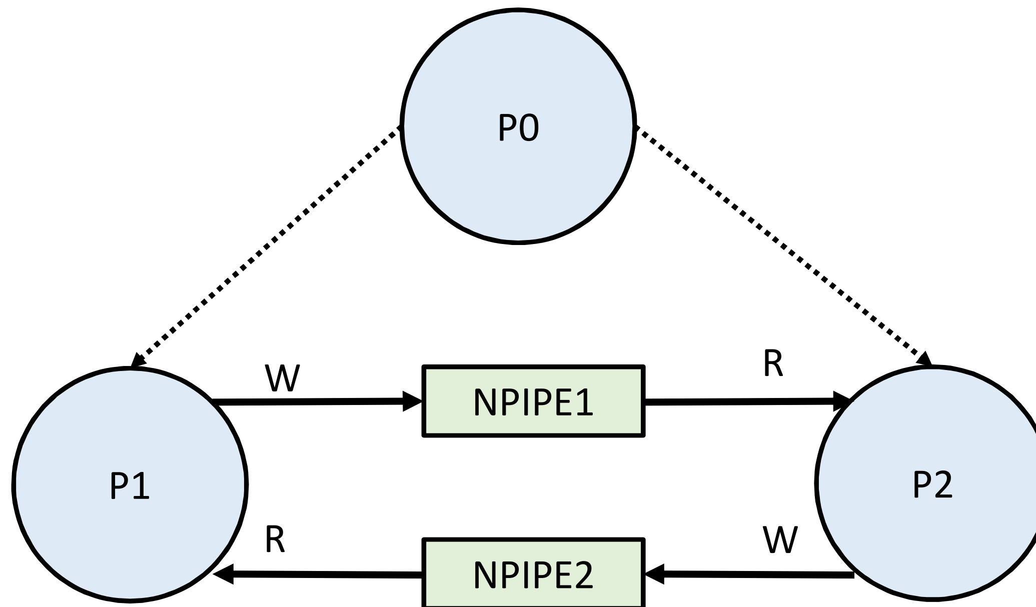
- The main body of the app would be like this:

```
char buf[20];        // for “write”
char valors[100]; // for collecting bytes from urandom

fd = open ("/dev/urandom", O_RDONLY);    // open urandom
if (fd < 0) { perror ("open"); exit(1); }
res = read (fd, valors, sizeof(valors));          // read 100 bytes from urandom
for (int i=0; i < 100; i++) {
   int valor = valors[i];                         // change each byte into “int” type
   write(1, &valor, sizeof(valor));               // write to “stdout” in binary format
}
sprintf(buf, "\n");                               // write a new line to “stdout” to end output
write(1, buf, strlen(buf));
for (int i=0; i < 100; i++) {
   sprintf (buf, "%d ", (int) valors[i]);         // change each byte to ASCII
   write(2, buf, strlen(buf));                    // write it into “stderr”
}
sprintf(buf, "\n");                               // include a final new line also to “stderr”
write(2, buf, strlen(buf));
```

Exercise 5

- Assuming there are two named pipes (“NPIPE1” and “NPIPE2”) implement the following communication diagram

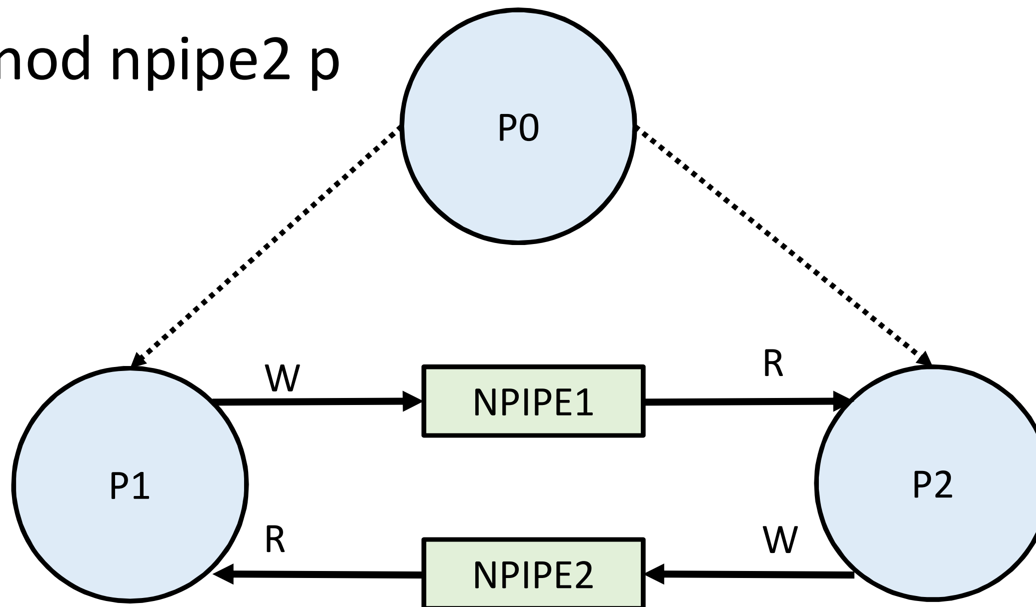


Exercise 5

Creating the named pipes from the shell

$ mknod npipe1 p

$ mknod npipe2 p

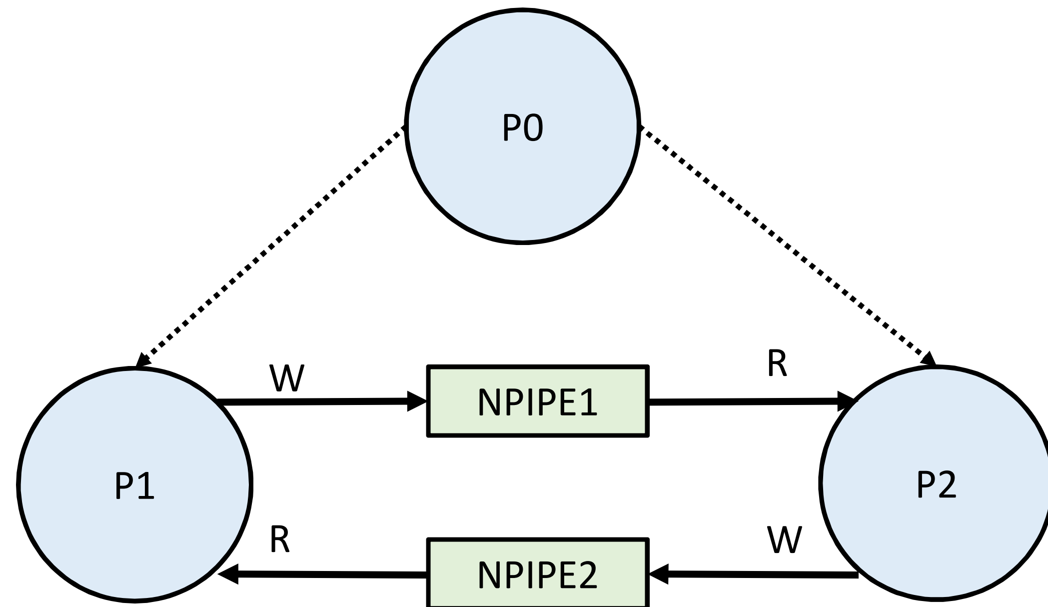


Exercise 5

Program (i)

```
#include <stdio.h>
#include <sys/file.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    int pid1, pid2;
```

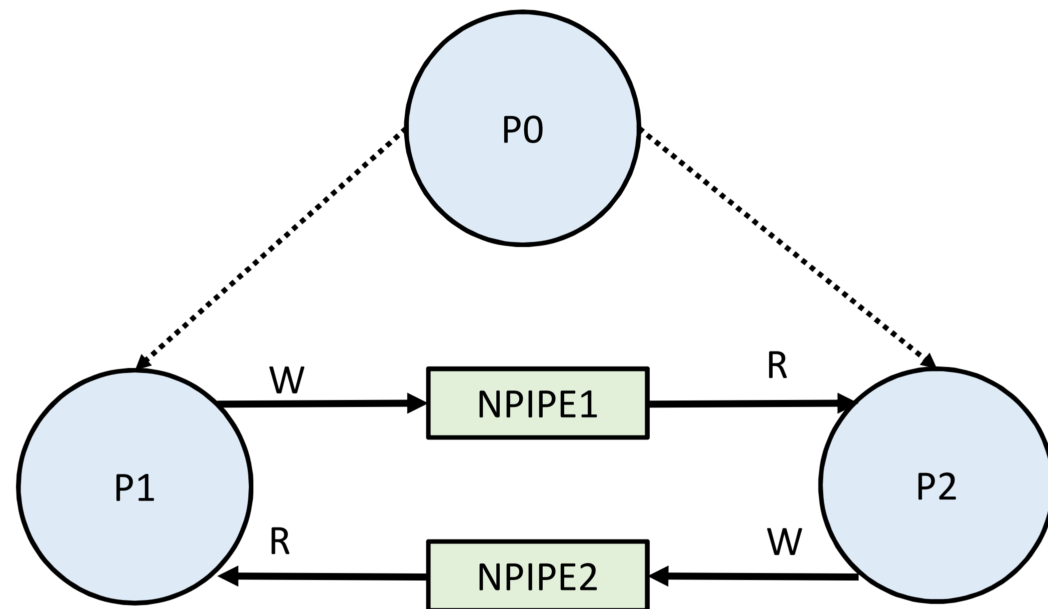



```
pid1 = fork();
if (pid1 == 0) { // P1
        int res;
        char buffer[100];
        int pw = open("npipe1", O_WRONLY, 0);
        if (pw < 0) { perror ("child1: open npipe1"); exit(1); }
        int pr = open("npipe2", O_RDONLY, 0);
        if (pr < 0) { perror ("child1: open npipe2"); exit(1); }

        write(pw, "message 1: hola\n", 16);
        res = read (pr, buffer, 100);
        write(1, buffer, res);
        close(pw);
        close(pr);
        exit(0);
}
else if (pid1 < 0) { perror ("fork1"); exit(1); }
```

Exercise 5

Program (ii)



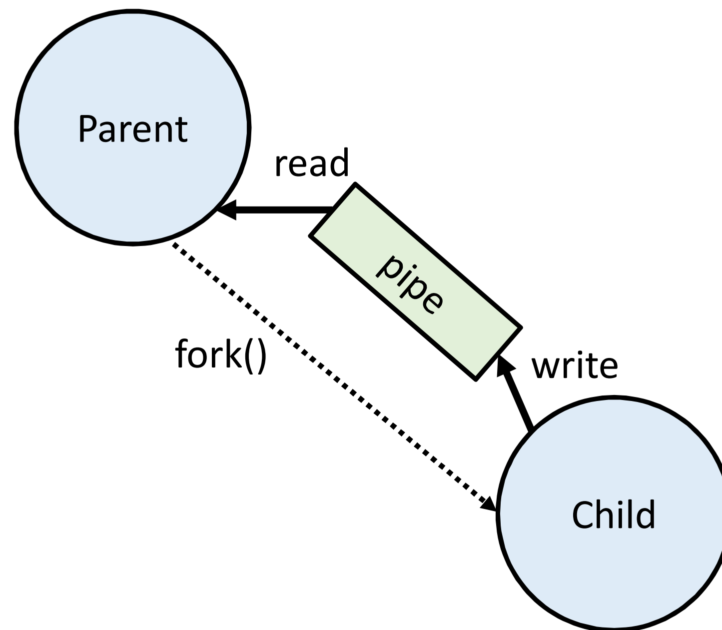

```
    pid2 = fork();
    if (pid2 == 0) { // P2
        int res;
        char buffer[100];

      // important to open RDONLY pipe first!!
      // otherwise: DEADLOCK!!
        int pr = open("npipe1", O_RDONLY, 0);
        if (pr < 0) { perror ("child2: open npipe1"); exit(1); }
        int pw = open("npipe2", O_WRONLY, 0);
        if (pw < 0) { perror ("child2: open npipe2"); exit(1); }

        res = read (pr, buffer, 100);
        write(1, buffer, res);
        write(pw, "message 2: adeu\n", 16);
        close(pw);
        close(pr);
        exit(0);
    }
    else if (pid2 < 0) { perror ("fork2"); exit(1); }
  int status;
    waitpid(pid1, &status, 0);     // check for errors
    waitpid(pid2, &status, 0);     // missing here!!!
    return 0;
}
```

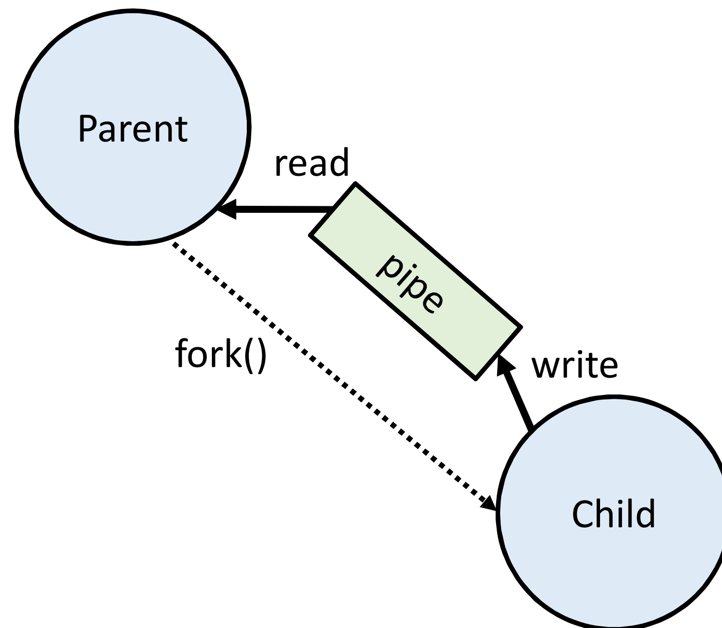
Exercise 6

- Implement the most basic communication of a **single message** passing through a pipe between (from) a child and (to) its parent process



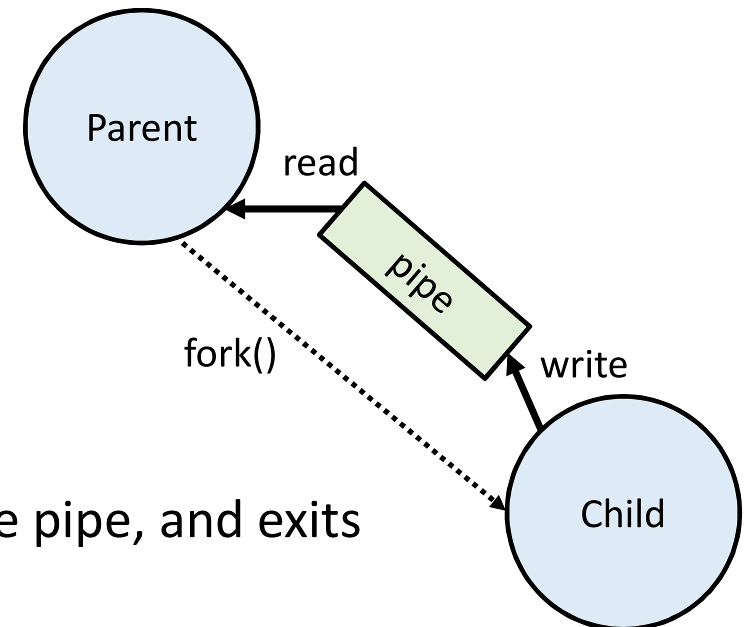
Exercise 6

- Solution provided in recording
  - Enunciat i explicació: creació i ús d'una pipe
    (avís al Racó "Videos sobre el T8 - E/S - ús de pipes" amb l'enllaç)



Exercise 7

- Implement the complete communication schema supporting **several messages** passing through a pipe between (from) a child and (to) its parent process
  - Get the messages from the command line
  - Parent and Child processes close the non-used pipe descriptors
  - The Child process sends the messages one by one
  - The Parent process receives the messages and displays them on the standard output
  - The Child process closes the output channel of the pipe, and exits This signals an end-of-file to the Parent process
  - The Parent process detects the end-of-file condition and waits for the child



Exercise 7

- Solution provided in recording
  - Solució completa, suportant múltiples missatges
    (avís al Racó “Videos sobre el T8 - E/S - ús de pipes” amb l’enllaç)