OPERATING SYSTEMS
– Laboratory 10 –

UNIX IPC: PIPES

1. Inter Process Communication (IPC) in UNIX: PIPES

One of the ways to communicate between processes in Unix is through communication channels (called pipes, in English). It's basically a "buffer" through which messages are written at one end and reads at the other end - so it's a queue structure, meaning a FIFO list (First-In, First-Out). These Pipes are two types:

- **Internal pipes:** they are created and exist in the internal memory of the Unix operating system
- **External pipes (also called FIFOS):** they are special files accessible in the files system; they are also called names pipes

2. PIPES and FUNCTION `pipe()`

▪ So an internal channel is a channel in memory through which two or more processes can communicate. The creation of an internal channel is done with the help of the pipe function:

```
#include <unistd.h>

int pipe(int *p);
```

▪ Where p - must be an int[2] array that will be updated by the function pipe with file descriptor values of the pipe ends:

- P[0] reading end
- P[1] writing end of the pipe

▪ the function returns:

- 0 - in case of success
- -1 - if the call failed (error)

▪ Effect: at the execution of the pipe function, an internal channel is created and is opened at both ends - in writing we referred to the end by p [1], respectively in reading at the end referred to by p[0]. After creating an internal channel, writing to and reading from this channel is the same as for regular files. Namely, the reading in the channel will be done through the descriptor p[0] using the usual reading functions (read, respectively fread or fscanf if a FILE * descriptor is used), and the writing in the channel will be done through the descriptor p[1] using the usual write functions (write, respectively fwrite or fprintf if a FILE * descriptor is used).

- Note: In order for two or more processes to use a pipe to communicate, they must have at their disposal the two descriptors p[0] and p[1] obtained by creating the pipe, so the process that created the pipe will have to somehow "transmit" them to the other process. For example, if you want to use an internal channel for communication between two parent-child processes, then it is enough to call the primitive pipe creating the channel before the call of the primitive fork to create the child process. In this way we have access to the pipe in the child process the two descriptors required for communication. The same is true for calling exec primitives (because open file descriptors are inherit by exec). It should also be noted that if a process closes any of the ends of a channel internally, then it has no possibility to reopen that end of the channel later.

3. READING FROM A PIPE with `read()`

- prototype:

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

- The read call will read from the pipe and return immediately, without crashing, only if there is still sufficient information in the pipe to read, and in this case the value returned is the number of bytes read from the channel.

- Otherwise, if the pipe is empty or does not contain enough information, the read call will be blocked until it has enough information in the pipe (when someone writes in the pipe) to be able to read the specified amount of information, which will happen when another process writes to the channel.

- Another exception to reading, in addition to emptying the channel: if a process tries to read from the channel and no process is ever able to write to the channel (because all processes have closed already the end of writing), then the read call immediately returns the value 0 corresponding to a read EOF from channel.

- In conclusion, in order to be able to read the EOF from the pipe, all the processes must first close the pipe for writing (i.e. to close the descriptor p [1]).

4. WRITING TO A PIPE with `write()`

- prototype:

```
#include <signal.h>

sighandler_t signal(int signum, sighandler_t handler);
```

- The write call will write to the channel and return immediately, without crashing, if there is enough free space in the channel, and in this case the value returned is the number of bytes actually written to the channel (which may not always match the number of bytes you want to write, as errors may occur for the I / O operation

- Otherwise, if the channel is full or does not have enough free space, the write call will remains locked until it has enough free space (free when reading, as reading pops out information) in the pipe to write the information specified as argument, which will happen when another process reads from the channel.

- Another exception to writing, in addition to filling the pipe: if a process tries to write to the channel and no process is ever able to read from the channel (because all processes have the read end already closed), then the system will send the SIGPIPE signal to that process, which causes its interruption and the message "Broken pipe" is displayed on the screen.

Notes

1. The above, about blocking read or write calls in the case of the empty channel, respectively full, correspond to the default, *blocking* behaviour of the internal channels.
2. High-level functions (*fread* / *fwrite*, *fscanf* / *fprintf*, etc.) work buffer-ized. (see *fflush*)
3. The conversion of the file descriptors from the *int* type to the *FILE* * type is done with the help of the *fdopen* primitive.

5. FUNCTIONS `popen() and pclose()`

- prototype:

```
#include <stdio.h>

FILE *popen(const char *command, const char *type);

int pclose(FILE *stream);
```

- command is a string that contains a shell command, and type is a string that can have either the value "r" or the value "w".

- The effect of the pop-up system call is as follows: open a pipe, then execute a fork. The child process executes the command by exec. The parent and child processes communicate through pipes as follows:

  – if the type is "r", then the father process reads from the pipe, using the pointer to the file returned by the pope, the standard output given by the command

  – if the type is "w", then the father process writes in pipe, using the pointer to the file returned by popen, and what it writes is the standard input for the command. If the command was launched with the redirected entry from the calling process, then we can still send data to the command launched via `fwrite()` or `fprintf()`. If the command was launched with the output redirected to the calling process, then the command output can be read from the calling process with the functions `fread()` or `fscanf()`. In both cases, the conclusion is made by calling the function `pclose()`. The pclose call closes the pipe and returns the order return code. On failure returns -1.

6. EXAMPLES

- Implement a process that creates a child process, and communicates with it using pipe. Parent process sends to child process 2 numbers and the child process returns through pipe their sum. Possible solution below (Be aware that some solutions contain mistakes, as explained at the lab).

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

int main() {


int p2c[2]; // pipe parent to child
int c2p[2]; // pipe child to parent
int a, b, s, pid;

pipe(p2c);
pipe(c2p);

pid = fork();
if(pid == 0) { //child process


  read(p2c[0], &a, sizeof(int));
  read(p2c[0], &b, sizeof(int));
  s = a+b;
  write(c2p[1], &s, sizeof(int));
  close(p2c[0]);
  close(p2c[1]);
  close(c2p[0]);
  close(c2p[1]);
  exit(0);
  }
```

```
        //parent process
        a = 3;
        b = 7;
        write(p2c[1], &a, sizeof(int));
        write(p2c[1], &b, sizeof(int));
        read(c2p[0], &s, sizeof(int));
        printf("%d + %d = %d\n", a, b, s);

        close(p2c[0]);
        close(p2c[1]);
        close(c2p[0]);
        close(c2p[1]);

        wait(0);

        return 0;
        }
```

Process hierarchy:

P

|

C

Have you found the code issues:

1. Close pipe ends as soon as possible, especially those that will not be used: eg. in the child process (if fork()==0 { close p2c[1]; close c2p[0]; read (p2c[0]....) and the same in the parent process.
2. Check if fork fails (pid== -1), otherwise issues in main.

The following program uses popen to execute ls -l | sort

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(){

         FILE *output;
         FILE *f2;
         f1=popen("ls -l","r");
         f2=popen("sort","w");

       char line[50];
       while (fgets(line,50,output)){
                fprintf(f2,"%s",line);
       }

       pclose(f1);
       pclose(f2);

}
```