Advanced Operating Systems
Practical 4: Control and communication of processes and threads

UAH, Departamento de Automática, ATC-SOL
`http://atc1.aut.uah.es`

Themes 4 and 5

Abstract

The goal of this practical is to study the mechanisms of control and communication of processes and threads, and to practise their use making two programs —one with processes and another one with threads— that use them.

1 Introduction

The goal of this practical is to study the mechanisms of control and communication of processes and threads. Section 2 presents the mechanisms of control and communication of processes, and proposes an exercise which consists in making a program that creates five interconnected processes that communicate among themselves. Two versions of this program will be made: one will use anonymous pipes and the other one will use named pipes. Section 3 presents the mechanisms of control and communication of threads, and proposes a similar exercise, but using threads instead of processes.

2 Processes

UNIX operating systems have a set of system calls that define a powerful interface allowing an easy approach to concurrent programming. This interface provides the software developer with tools for the creation, synchronization and communication of new processes, as well as the ability to run new programs.

One of the main aspects of process management in UNIX is the way in which processes are created, and how new programs are executed. It is assumed that the student already knows the system calls `fork()` and `exec()`.

As important as understanding process creation is to know how they can communicate. Although the set of tools for communication between processes in UNIX is very large, we will see two methods of communication processes: by sending signals between them, and through the use of certain communication channels called pipes that allow the exchange of information between related processes (*pipes*) and between unrelated processes (*fifos*).

2.1 Process control

2.1.1 Creation of a process and execution of a new program

The `fork()` system call is a key mechanism in process control, since it is the method provided for applications to create new processes. If necessary, review the details of the system call `fork()` in Appendix A of this document.

Another key mechanism is the `exec()` system call, which allows "overlaying" an existing process —the one that calls `exec()`— with the code of a new program, making the process execute this new code instead of the one it was executing before. If necessary, review the details of the system call `exec()` in Appendix B of this document.

2.1.2 Process termination: `exit()` and `wait()` system calls

The `exit()` system call terminates the process from which it was called, leaving a final state code equal to the less significant byte of the argument passed in the system call. All file descriptors are closed and their buffers synchronized. If any children processes are running when their parent calls `exit()`, then PPID[1] of the children is changed to 1 (process "init"). This is the only system call that never returns.

The value of the parameter passed to `exit()` is used for letting the parent process know how did the child process terminate. By convention, this value is 0 if the process has completed successfully and any other value if it terminated abnormally. The parent process can obtain this value by calling the `wait()` system call.

The `wait()` system call suspends the execution of the caller process until one of its child processes has finished. If a child has already exited when the call is performed, then `wait()` returns immediately. All the resources allocated for the child process are freed.

If there are several child processes, `wait()` waits for the termination of any one. The calls to `wait()` can not specify which child to wait for[2]. The call to `wait()` returns the PID[3] of the child that terminated, or -1 if no child was created or there is no child to wait for, or if the child process receives a signal. A parameter can be passed by reference in order to retrieve the final state code of the child process —the value passed by the child process when it called `exit()`—. If the parent process prefers to ignore this code, it can simply pass NULL when calling `wait()`.

The value returned by `wait()` through this by-reference parameter depends on the reason why the child process terminated, as shown in Table 1. For more details, see the manual.

A process might finish in a moment when the parent is not waiting for it. As the kernel must ensure that the parent can wait for every child process, those that are not waited for, become zombie processes —most of their resources are freed, but they still occupy an entry in the process table of the kernel—. When the parent calls `wait()`, the child process is removed from the process table. Calling `wait()` for every child process is strongly recommended, though not mandatory.

Another system call allows a process to suspend its execution until a specific child process finishes. The `waitpid()` system call receives as arguments, in addition to the variable —passed

[1] PPID: identifier of the parent process of a process

[2] In order to wait for the termination of a specific child, use `waitpid()` instead of `wait()`

[3] PID: process identifier

Table 1: Possible state values written by `wait()`

Reason for termination	**State value**
`exit()` system call	The less significant byte is zero, and the second to less significant byte takes the value passed to `exit()`.
It received a signal	The 7 less significant bits store the number of the signal that terminated the process. If the 8^{th} bit is 1, then the process was terminated by the kernel, and a dump of the process was saved in a core file.
System breakdown (i.e.: power loss)	All processes disappear abruptly. There is nothing to return.

by reference— where to store the value given by the child, the PID of the child to wait for, and a set of options that modify the behaviour of the system call. You can find detailed information about this system call on the manual.

The next program is a usage example of `fork()` and `waitpid()`:

```
user@host:$ cat example_waitpid.c

/*
    example_waitpid.c
*/

#include <stdio.h>      // printf(), perror(), ...
#include <unistd.h>     // fork(), getpid(), getppid(), ...
#include <sys/wait.h>   // waitpid(), ...

int parent (int pid_child);   // Parent's exclusive code
int child ();                 // Child's exclusive code

int main ()
{
    int i;

    i = fork ();      // Fork the execution

    printf ("I'm %d, child of %d. To me, fork() returned %d\n",
            getpid(), getppid(), i);

                // Sleep one second (this way both processes will
    sleep (1);  // have time to show the previous message before
                // every one goes on with its own code)
    switch (i)
    {
        case -1:  perror ("Error in fork");   // Show error and get
                  break;                      // out of the switch

        case 0:   return child ();   // Execute child() and finish

        default:  return parent (i); // Execute parent() and finish
    }
```

```
    return -1;
}

int parent (int pid_child)
{
    int r;

    printf ("PARENT: I'm the parent of %d.\n", pid_child);

    printf ("PARENT: I'll wait until my child finishes.\n");
    waitpid (pid_child, &r, 0);
    printf ("PARENT: Done. The result is 0x%x.\n", r);

    printf ("PARENT: Now I can finish too.\n");
    return 0;
}

int child ()
{
    const int a = 0x2a;

    printf ("CHILD: I'm the child.\n");

    printf ("CHILD: I'll sleep some seconds.\n");
    sleep (5);

    printf ("CHILD: I stretch myself and exit with 0x%x.\n", a);
    return a;
}

user@host:$ gcc example_waitpid.c -Wall -o example_waitpid
user@host:$ ps
  PID TTY          TIME CMD
 2480 pts/0    00:00:00 bash
 3704 pts/0    00:00:00 ps
user@host:$ ./example_waitpid
I'm 3705, child of 2480. To me, fork() returned 3706
I'm 3706, child of 3705. To me, fork() returned 0
PARENT: I'm the parent of 3706.
PARENT: I'll wait until my child finishes.
CHILD: I'm the child.
CHILD: I'll sleep some seconds.
CHILD: I stretch myself and exit with 0x2a.
PARENT: Done. The result is 0x2a00.
PARENT: Now I can finish too.
```

Every time a process finishes, the operating system sends a `SIGCHLD` signal to its parent process. The treatment of this signal depends on the implementation of the parent process.

2.2 Process communication

2.2.1 Signals

Signals are a mechanism to notify a process of a given situation by sending an asynchronous message to the process. The message sent is an integer that will indicate the cause of the notification.

Some of the possible signals that a process might receive are: an action sent from the keyboard —pressing `Ctrl+C`, for instance—, an error detected by the hardware, an access to a memory zone that does not belong to the process, etc.

Processes can indicate what to do when they get a specific signal. Tough, for some signals they can not modify the standard behaviour[4]. When a process receives a signal, it can handle the event in three different ways:

- Ignore it.
- Let the system execute the default action.
- Execute its own routine, called "signal handler".

In this latter case, when the process receives a signal, its execution is interrupted —the state of the process is saved in its own stack— and the signal handler —usually a very short procedure— is called. When the execution of the signal handler ends, the state of the process is restored, and its execution continues at the point where it was interrupted —the behaviour of this mechanism is quite similar to that of hardware interrupts.

Table 2 shows some of the signals used more often. By reading the manual for `signal` in section 7 (`man 7 signal`) you can learn about other signals used in the system.

Table 2: Some signals of generalized use

Name	**Num.**	**Meaning**
`SIGINT`	2	Process interruption (sent from the terminal with `Ctrl+C`).
`SIGQUIT`	3	Quit producing a core file (sent from the terminal with `Ctrl+\`).
`SIGKILL`	9	Irrevocable, immediate termination of the process.
`SIGALRM`	14	Notification that a timer —created with `alarm()`— has expired.
`SIGTERM`	15	Controlled finalization of the process.
`SIGCHLD`	17	Notification that a child process stopped or terminated.
`SIGSTOP`	19	Stop process execution (sent from the terminal with `Ctrl+Z`).

In addition to the set of already defined signals with a specific purpose, there exist two signals intended to be defined and sent by the user. These are signals `SIGUSR1` (10) and `SIGUSR2` (12).

A first and easy method to modify the behaviour of a process when it receives a signal is provided by the `signal()` system call. It receives two arguments. One is the signal to be treated and the other specifies what to do with that signal. The second argument must be, depending on the desired behaviour:

- Ignore the signal → the second argument must be `SIG_IGN`.

[4]In particular, processes always behave the same way when they receive the signals `SIGKILL` and `SIGSTOP`

- Default action → the second argument must be `SIG_DFL`.
- Customized behaviour → the second argument must be the address —the bare name— of the function that will handle the signal.

The function installed as signal handler with `signal()` must receive an integer and return nothing:

```
void function_name (int signum);
```

Where the parameter it receives is the identifier of the signal that provoked the call[5]. On some systems, when a signal handler is executed, it is automatically uninstalled. Therefore, on such systems, the signal handler must reinstall itself at the end of the routine. On other systems, it is not automatically uninstalled, and thus the signal handler must uninstall itself at the beginning of the routine just in case the same signal was received again, which would otherwise interrupt the signal handler and start executing it again before the first signal was treated.

By using the `kill()` system call, a process can send a signal to another process —or to itself—. This system call receives as arguments the PID of the process that should receive the signal and the identifier of the signal to be sent. Two functions are very useful in combination with this system call: `getpid()` and `getppid()`, which return the identifier of the current process and its parent, respectively.

The next program is a usage example of `kill()` and `signal()`:

```
#include <unistd.h>       /* Example of kill and signal:     */
#include <signal.h>       /* It makes a fork, and then the   */
#include <sys/wait.h>     /* child sends a signal to the     */
#include <stdio.h>        /* parent, who then calls printf   */

void handler (int signum)
{
    printf ("Signal %d. Roger!\n", signum);
}

int main ()
{
    signal (SIGUSR1, handler);

    switch (fork())
    {
        case -1:  perror ("fork");              return -1;
        case 0:   kill (getppid(), SIGUSR1);    return 0;
        default:  wait (NULL);                  return 0;
    }
}
```

If you are unsure what process will execute the handler, or what is the time sequence of the execution, add commands like `printf("%d: I'm here.\n",getpid())` where deemed necessary.

[5]One single routine can be the signal handler of several signals at the same time

2.2.2 Unnamed pipes

Most UNIX users are familiar with unnamed pipes (`|`), since they have used them directly in the *shell*. The next line, for instance, prints —to the default printer— in alphabetical order the names of the users logged in the system.

```
user@host:$ who | sort | lp
```

The previous command creates three processes connected by pipes. In this example, the data flow is unidirectional —from `who` to `sort`, and from `sort` to `lp`—. Nevertheless, it is possible to connect processes with bidirectional data flows, or even forming loops with them. However, these ways of using unnamed pipes are only available through the system call interface, so some UNIX users are unaware of these possibilities.

The `pipe()` system call creates a communication channel represented by two file descriptors, returned through an array passed as argument of the call. This communication channel works as a FIFO structure —*first in* → *first out*.

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

Assuming that the array passed to `pipe` is called `fd`, writing in the descriptor `fd[1]` will enter data into the pipe, and reading the descriptor `fd[0]` will extract data from it. The system calls `open()`, `read()`, `write()` and `close()` act in a slightly different when performed on unnamed pipes:

- `open()`: This system call is not used with unnamed pipes. The way to create —and open— an unnamed pipe is through the `pipe()` system call.
- `read()`: In pipes, data are read in arrival order, that is, in the same order in which they were written. It is not possible to make non-sequential read operations on unnamed pipes. As data are read, they are removed from the pipe, so that they can not be re-read (destructive read). Reads are usually blocking, which means that, if a process calls `read()` on an empty pipe, it will be blocked until another process writes something in the pipe. The exception to this rule is the case where all descriptors for writing in the pipe are closed. In this case, `read()` will immediately return 0. The reader process is unblocked as soon as some data are written in the pipe, even though the number of bytes does not satisfy the count argument passed to the `read()` system call.
- `write()`: Similarly, data are introduced in the pipe in arrival order. When the pipe gets full, `write()` blocks until a `read()` call removes enough data to let the write operation complete. Unlike the case of read, there are no partial writes. All the data passed to `write()` are written to the pipe. The capacity of an unnamed pipe varies across UNIX implementations, but it is in no case less than 4096 bytes.
- `close()`: Its importance is greater than for regular files, since closing the writing end is the only way to notify the end-of-line condition to the reader processes. Similarly, if the reading end is closed, any attempt to write will fail, returning the corresponding error value.

To connect two or more processes through an unnamed pipe, the usual technique is to create the pipe before the child processes as these, inheriting the table of open file descriptors from the parent, obtain a copy of the descriptors associated with the pipe. This way it possible to connect parent processes with their children —and even grandchildren—, and child processes between themselves. The only requisite is that they have obtained a copy of the pipe descriptors from an ancestor. However, there is no way to connect two unrelated processes through unnamed pipes.

One of the most common problems when working with unnamed pipes is the occurrence of deadlocks due to mismanagement of the pipes. Such deadlocks typically occur when a process reads from a pipe which is never written to —or vice versa—. The following program is an example:

```
#include <unistd.h>

int main ()
{
    int fd[2];
    char c;

    pipe (fd);                          // Create the pipe
    fork ();                            // Create child process
    read (fd[0], &c, sizeof(char));     // Read (both, parent and
                                        // child --> deadlock!)
    return 0;
}
```

Another source of deadlocks is the mismanagement of the descriptors: processes seem to work properly and do their job, but in the end they "hang", this usually happens when the processes do not close the unused pipe descriptors until they finish. A good rule of thumb is to close at the beginning of each process —right after the `fork()`— those pipe ends that will not be used.

2.2.3 Named pipes (*fifos*)

Named pipes combine the features of files and unnamed pipes. Like a file, a named pipe has an associated name, and any process with the appropriate permissions can open it to read and/or write. Unlike with unnamed pipes, several unrelated processes can communicate using a named pipe, not being required any inheritance relationship. However, once the named pipe has been opened, its behaviour is closer to an unnamed pipe than to a file. Read and write operations follow the same rules as those set for unnamed pipes —blocking and destructive read, blocking write, etc.

Named pipes can be created either from the shell using the command `mknod`, either from a program using the system call `mknod()`.

After creating a named pipe, any process can operate on it using standard system calls —`open()`, `read()`, `write()` and `close()`.

Usually, when a process opens a named pipe for reading —or writing—, it gets blocked until another process opens the pipe in the complementary mode. This allows synchronization between the processes before the start of data transmission.

2.3 Exercise of processes

2.3.1 Unnamed pipes

Consider the following program code:

```
1  /*
2      processes_echo_caps.c
3  */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <unistd.h>
8  #include <sys/wait.h>
9  #include <string.h>
10 #include <time.h>
11
12 #define SIZE 100
13
14 int io (int readend, int writeend);      // Input / output
15 int caps (int readend, int writeend);    // Conversion to caps.
16
17 int main ()
18 {
19     int forth[2], back[2], r;
20
21     if (pipe (forth) || pipe (back))     // Open two pipes (one
22     {                                    // for each direction of
23         perror ("pipe");                 // the communication)
24         return -1;
25     }
26
27     switch (fork())                                  // Fork
28     {
29         case -1:  perror ("fork");                   // Error
30                   return -2;
31
32         case 0:   close (forth[1]);
33                   close (back[0]);                   // Child
34                   return io (forth[0], back[1]);     // I/O
35
36         default:  close (forth[0]);
37                   close (back[1]);                   // Parent
38                   r = caps (back[0], forth[1]);      // Caps.
39                   wait (NULL);
40                   return r;
41     }
42 }
43
44 int io (int readend, int writeend)      // Input / output
45 {
46     int r;
47     char buff[SIZE];
48
```

```
49      printf ("Type lines of text (Ctrl+D to finish):\n");
50
51      for (;;)     // (Almost) infinite loop
52      {
53          if (fgets (buff, SIZE, stdin))  // Read from keyboard
54          {
55              if (write (writeend, buff, 1+strlen(buff)) < 0)
56                  break;
57          }                               // If read, send (if error
58          else                            // while sending, abort)
59              close (writeend);           // If EOF, close pipe
60
61          r = read (readend, buff, SIZE);
62
63          if (r < 0)          // If error, abort
64              break;
65          else if (r == 0)    // If pipe was closed,
66              return 0;       // finish ok
67
68          fputs (buff, stdout);  // Show what the child process
69      }                          // returned, and repeat
70
71      perror ("io");       // Error! (aborted from the loop)
72      return -1;           // Print error message and exit
73  }
74
75  int caps (int readend, int writeend)  // Conversion to caps.
76  {
77      int i, r;
78      char buff[SIZE];
79
80      srand ((unsigned)getpid() ^ (unsigned)time(NULL));
81
82      for (;;)     // (Almost) infinite loop
83      {
84          r = read (readend, buff, SIZE);   // Receive string
85
86          if (r < 0)                // If error, abort
87              break;
88          else if (r == 0)          // If pipe was closed,
89          {                         // close the other one
90              close (writeend);     // and finish ok
91              return 0;
92          }                         // Convert some letters to
93                                    // capitals (with prob. 0.5)
94          for (i=0; buff[i]; i++)
95              if (rand() > RAND_MAX/2)
96                  if (buff[i] >= 'a' && buff[i] <= 'z')
97                      buff[i] -= ('a' - 'A');
98
99                                    // Send back
100
101         if (write (writeend, buff, 1+strlen(buff)) < 0)
102             break;
```

```
103         }
104
105         perror ("caps");     // Error! (aborted from the loop)
106         return -1;           // Print error message and exit
107     }
```

The program `processes_echo_caps` launches a second process by calling `fork()` (line 27). The `main()` function makes each process —parent and child— execute a different function —see calls to `io()` and `caps()` in lines 34 and 38 respectively—. The two processes will communicate with each other through the pipes created in line 21.

The function `io()` is responsible for the input/output of the program, asking the user to type phrases, and showing them later —after making the other process transform them.

The function `caps()` is in charge of transforming the strings received from the other process. The transformation consists in simply shifting to capital letters some of the characters of the string —specifically, each character has a 0.5 probability of being changed.

Both functions receive as arguments the file descriptors of the pipe ends where they should read/write.

When the user presses `Ctrl+D`, signalling the end of the input, the function `io()` closes the writing end of the *forth* pipe, thus making the other process finish in an orderly manner. For implementing this type of synchronization mechanism, it is essential to close in advance the pipe end descriptors that will not be used, as is done in lines 32, 33, 36 and 37.

Make a similar program with five processes that communicate with each other as shown in Figure 1. The same process that makes the input/output must divide the strings into two halves and send each half through a different path —I/O-and-multiplexer process—. You can use the following function to divide the strings:

```
#define SIZE 100
#define SIZE_HALF ((SIZE)/2 + 1)

void split (char all[SIZE], char half1[SIZE_HALF],
                            char half2[SIZE_HALF])
{
    int a;

    a = strlen (all) / 2;

    strncpy (half1, all, a);
    half1[a] = '\0';

    strcpy (half2, all+a);
}
```

The three intermediate processes must act exactly as the process of the function `caps()` of the previous program. Indeed, they three must run the same function. The arguments passed to the function in each process will make it communicate through the corresponding pipes.

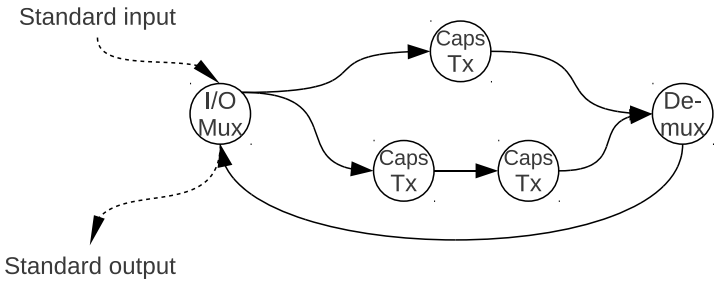

Figure 1: Data flow between processes in the exercise

The final process, which receives the two halves through different channels, must unite them —that is why we call it the demultiplexer process— and send the result back to the initial process, which displays the string on the screen. You can use the following function to rejoin the two halves:

```
void concatenate (char all[SIZE], char half1[SIZE_HALF],
                                  char half2[SIZE_HALF])
{
    strcpy (all, half1);
    strcat (all, half2);
}
```

In addition, all processes should display a message on the screen every time a string passes through them. The program output should look similar to this execution example:

```
user@host:$ ./processes_mux_tx_demux
Type lines of text (Ctrl+D to finish):
I'd like to have an argument, please.
        Mux  : separating 39 bytes
        Tx_B1: transmitting 20 bytes
        Tx_A : transmitting 20 bytes
        Tx_B2: transmitting 20 bytes
        Demux: rejoining 20+20 bytes
I'D likE TO HaVE AN ArGUMENt, PLEAse.
```

The messages from the `Tx`.. processes can appear in a different order every time, since these processes run concurrently. The message from `Tx_B1` will always appear before the message from `Tx_B2`, but the message from `Tx_A` might appear before the other two, in the middle, or after them.

Also noteworthy is that, counting the two halves, the message transmitted occupies one more byte than the original string. This is because, having two strings —one for each half—, there are two end-of-string zeros(`'\0'`) instead of just one.

Organize your source code in different `.c` and `.h` files, distributing the functionality of the program in five modules:

1. `main()` function: it will create the pipes and launch the processes.

2. String handling functions: `split()` and `concatenate()`.
3. I/O-and-multiplexer process.
4. Caps-and-transmitter processes. Remember: Only one function is required, even though three processes will be executing it at the same time!
5. Demultiplexer process.

Coordinate the compilation using a `makefile` and add comments to the source code.

2.3.2 Named pipes (*fifos*)

Make a new version of the previous exercise by separating the processes into three different programs —each with its own `main()` function—. These processes must communicate with each other via named pipes. The names of the pipes will be specified through the command line when invoking the programs. For every process, the input pipe(s) —where it must read— will be specified first, and then the output pipe(s) —where it must write—. Additionally, an optional final parameter can specify the name of the process.

For example, to repeat in this section a configuration equivalent to that of Figure 1 we should invoke the three programs with a command sequence like the next one:

```
user@host:$ (./tx T0 T1 Tx_A &) ; (./tx T2 T3 Tx_B1 &) ; \
> (./tx T3 T4 Tx_B2 &)
user@host:$ (./demux T1 T4 T5 Demux &) ; ./io_mux T5 T0 T2 IoMux
Type lines of text (Ctrl+D to finish):
...
```

Where:

- `tx`, `demux` and `io_mux` are the names of the **programs**.
- `Tx_A`, `Tx_B1`, `Tx_B2`, `Demux` y `IoMux` are the names of the **processes**.
- `T0`, `T1`, `T2`, `T3`, `T4` y `T5` are the names of the **pipes**.

The `makefile` will carry out its usual tasks and it will also create ten named pipes called `T0`, `T1`, ... and `T9` by using the command `mknod` —see `man 1 mknod`—. The programs `tx`, `demux` and `io_mux` just have to open the pipes in the adequate mode —`O_RDONLY` for reading, and `O_WRONLY` for writing.

To avoid deadlocks, the order in which pipes are opened is important. If every process opens first its input pipes —in read mode—, they all will get blocked, waiting for someone to open them in write mode, but nobody will, because they are all blocked. In this case, the problem can be solved easily by making one of the programs —`io_mux`, for example— open first its output pipes.

Nevertheless, that is not enough to prevent all possible deadlock situations. If `io_mux` communicates directly with `demux` through two pipes, and they do not open them in the same order, then another deadlock is produced. Thus, the command "`(./demux T1 T2 T9 &);./io_mux T9`

`T1 T2`" can work correctly while the command "`(./demux T1 T2 T9 &);./io_mux T9 T2 T1`" provokes a deadlock[6]. The problem is that, by allowing the user specify the pipes at will, we lose control over the order in which programs open them. Appendix C discusses two methods for solving this situation, but you do not need to apply them in your exercise.

In any case, the following **conditions** must be met:

1. No busy wait is allowed.
2. All processes must start correctly, regardless of the order in which programs are invoked.
3. When the user presses `Ctrl+D`, pipes must start to be closed, making the processes terminate in chain. No process should get blocked. Check the status of processes with the command `ps -l`
4. All of the following settings must work properly:
   - `(./demux T1 T2 T9 &) ; ./io_mux T9 T1 T2`
   - `(./demux T1 T3 T9 &) ; (./tx T2 T3 &) ; ./io_mux T9 T1 T2`
   - `(./demux T1 T5 T9 &) ; (./tx T2 T3 &) ; \`
     `(./tx T3 T4 &) ; (./tx T4 T5 &) ; ./io_mux T9 T1 T2`
   - `(./tx T0 T1 &) ; (./tx T2 T3 &) ; (./tx T3 T4 &) ; \`
     `(./demux T1 T4 T5 &) ; ./io_mux T5 T0 T2`
5. The target `clean` of the `makefile` must delete all object files, executable files, and pipes.

Note: During the development and debugging of the programs, if they get blocked at some point, then you should make sure to kill all processes before going on. This way you will avoid them to interfere in subsequent experiments. Use the command `ps -l` to list them and the command `kill` to kill them. In addition, the command `ps -l` reports the PID and status of processes. The PID can be useful to obtain more information about the sate of a process —before killing it, of course—. For example: `ls -Fal /proc/`*PID*`/fd`

3 Threads

Current operating systems feature a mechanism to provide concurrency between processes within the same address space. This kind of "process" is usually called thread (*hilo*) or lightweight process (*proceso ligero*).

The process that runs a multithreaded program —usually called "task" in order to avoid confusions— contains several execution threads instead of only one. The programmer writes parts of the program that will run concurrently —even in parallel, if there are several processors or cores—, and each one of this parts is called thread.

The difference between having several processes and having several threads is that threads share share their memory[7]. The different threads of a task do not require expensive mechanisms

[6] Note how `T1` and `T2` have been swapped at the end of the command

[7] The threads of a task access the same global variables, and can also share, through pointers, the variables allocated in their stacks or dynamically

to communicate with each other. In addition, they do not need to store much of their context —which processes do to preserve their variables—, reducing the cost of context switching between threads of the same task.

Each thread has its own program counter, a copy of the registers —where the value of the registers is stored when the thread is taken out of the processor— a user mode stack —for its local variables etc.— and a supervisor mode stack —used by the kernel during system calls.

All the other resources of a task —global variables, dynamic memory, file descriptors, memory mappings, signals and the code itself— are shared by its different threads. Indeed, the address space itself is shared by the threads of the process, so that a thread can access with a pointer to the stack of another thread[8].

In a multithreaded environment, the unit of protection and resource assignment is the process. Threads are supposed to cooperate with each other with good will and extreme care.

In return, starting a new thread within an existing task has a lower cost than creating a new process, and switching from one tread to another within the same task also has a lower cost than switching to a different process. The termination of a thread also has a lower cost than the termination of a process, since there are fewer resources to release.

Adding to all this the cost reduction in communications, threads are a great alternative to processes in, for example, "server" programs that attend a variable —and potentially very large— number of clients.

On UNIX-like operating systems, the implementation of threads usually follows the POSIX specification, thus receiving the name of "POSIX Threads".

3.1 Semaphores

The most common mechanism for thread synchronization —and indeed, for processes too— is the semaphore.

A semaphore is a special object that contains a counter and only allows access to it through two primitives: `P` y `V`[9]. It is also usual to call them `wait` and `signal` respectively. In POSIX semaphores, their names are `sem_wait()` and `sem_post()` respectively.

The `V` primitive increments the counter, while the `P` primitive decrements it. The key detail is that the counter can never take negative values, so that if a thread tries to decrement the counter when it is already zero, this thread will go to sleep until another thread increments the counter. Only then, will the first thread be able to decrement the counter and continue running.

The next scenario, for example, is very usual: a thread X makes `P` on a semaphore whose counter is already zero, in order to wait sleeping until another thread Y reaches some given point in its execution. When Y reaches that point, it makes `V` on the same semaphore, which wakes thread X up. If thread Y reaches the agreed point first, and makes `V` first, then thread X will simply make `P` without blocking. In both cases, the counter will end up being zero.

Note that, in this example, the `P` and `V` operations are carried out by different threads.

[8] Yes: it can also corrupt it. Every other thread of the process has the same ability to corrupt it as the thread that "owns" the stack.

[9] The name P comes from the Dutch *proberen* —to try or attempt— and the name V comes from the Dutch *verhogen* —to increment

3.2 Mutexes

Mutexes are useful for ensuring mutual exclusion —hence the name— of several threads in their access to shared resources. They are not exactly semaphores, though a basic mutex can be implemented using a semaphore.

The difference between mutexes and semaphores is, mainly, on the way they are used. Semaphores are typically used to ensure a particular order in the execution of different sections of the code of several threads —as in the example above—. Mutexes, instead, are not used to impose a particular order of access to the shared resource, but only to prevent two threads from accessing it simultaneously, no matter which one goes first.

Despite of this difference, the best way to explain what mutexes are, is comparing them with semaphores. A basic mutex is a semaphore initialized with its counter set to 1, and used by threads following the sequence of operations { `P`; Access the shared resource; `V` }. The code between the `P` and the `V` is called "critical section". The mutex guarantees that there will never be more than one thread executing the critical section simultaneously. The first thread that reaches the `P` is not blocked, but if another thread arrives just after that, the newcomer will get blocked until the first thread reaches the `V` —that is, until the thread that "owns" the mutex gets out of its critical section and releases the mutex.

There is a special type of mutex called "recursive mutex" that allows the counter to take negative values in a particular circumstance: when the thread that makes the `P` is the same one that already owns the mutex.

The POSIX specification states that a mutex must be released by the thread that owns it. In the most basic —and faster— mutexes, if this rule is violated, then the behaviour is undefined and implementation dependent.

3.3 Exercise of threads

The exercise of this section will consist in completing a modified version of the program `processes_echo_caps` of Section 2.3.1. This modified version works with threads instead of processes. Thus, in one single process or task, several instances of the `caps()` function will run concurrently —one per thread—, and they all will work on the same shared buffer.

See the listing of the program `threads_echo_caps` provided with this text in the web page. The function `main()` parses the parameters passed in the command line, creates a couple of semaphores, launches the requested number of threads —making them execute the function `caps()`— and calls the function `io()`.

The functions `caps()` and `io()` receive, as their only parameter, a pointer to a structure that contains all the data they work with. The function `main()` calls `pthread_create()` passing a pointer to the function that the thread should execute —third parameter—, and the pointer that this function itself should receive as parameter —fourth parameter of `pthread_create()`.

In the previous exercises, pipes were used as a means of communication, but also solved the problem of synchronization between processes. Working with threads, no special mechanism of communication is required, since all threads access the same address space, so they can directly share the data. However, they still need a synchronization mechanism.

The threads of the program `threads_echo_caps` are synchronized using semaphores. The input/output thread waits on a semaphore, and the "caps" threads —those who convert from small letters to capital letters— wait on another semaphore. The input/output thread reads a line from the standard input and passes the baton to the "caps" threads, waking them up with `sem_post()` applied to `semaphore_caps`, and falling asleep with `sem_wait()` applied to `semaphore_io`. Then the "caps" threads do their work and give the baton back to the input/output thread, waking it up with `sem_post()` applied to `semaphore_io`, and falling asleep with `sem_wait()` applied to `semaphore_caps`. Then the input/output thread shows the resulting string and starts again.

The function `tsrandom()` computes a random number calling `rand()`. In fact, `tsrandom()` adds nothing in terms of the calculation. It only ensures that the concurrent execution of `rand()` by several threads does not corrupt the calculation. Even though several threads call `tsrandom()` at the same time, the call to `rand()` will be executed in mutual exclusion, thanks to the `mutex` used in `tsrandom()`. The keyword `static` in the declaration of the variable `mutex` is vital to achieving this goal. This keyword makes the variable unique —like global variables— instead of being repeated, in the stack, for every call to `tsrandom()`. In other words, if several threads call `tsrandom()`, each thread will have its own `i` variable, but there will be only one `mutex` variable for all of them.

See the internal loop of `caps()`, where some letters are converted to capital letters. It has been intentionally programmed in a way that increases the chances of generating errors due to race conditions[10]:

```
150                                             // Convert some
151         for (i=0; sd->buff[i]; i++)         // letters to caps.
152             if (tsrandom() > RAND_MAX/2)    // (with prob. 0.5)
153             {
154                 c = sd->buff[i];              // Read character
155
156                 if (c >= 'a' && c <= 'z')     // If small letter
157                 {
158                     if (sd->wtime)                // Take a
159                         usleep (sd->wtime);       // nap
160
161                     sd->buff[i] -= ('a' - 'A');   // Convert
162                 }
163             }
```

If two threads execute concurrently —more or less at the same time in different processors/-cores, or by small chunks in a single core— the lines 154-161, and with the same value of `i`, then chances are that the decrement of line 161 will be applied twice, corrupting the character `dc->buff[i]` instead of just converting it to capital letter[11]. For example:

[10] Race condition: flaw in the implementation of a concurrent algorithm in which, unexpectedly, the timing of events —due to the speed of some processes/threads with respect to others— might critically alter the result.

[11] It is not very important in this case, but the decrement itself at line 161 probably will not be executed atomically. Paradoxically, two simultaneous decrements might have the same effect as only one. Furthermore, the probability of data corruption will also depend on how fast is the computer, whether it is heavily loaded or not, and the numbers of cores/processors.

```
user@host:$ ./hilos_eco_mayus 1 20
Type lines of text (Ctrl+D to finish):
With one single "caps" thread, nothing strange happens.
WiTH oNE SINGLe "capS" thrEAD, nOtHINg StRaNgE HApPENS.

user@host:$ ./hilos_eco_mayus 2 1
Type lines of text (Ctrl+D to finish):
With two threads and a small delay, only some letters change.
W)4h 4WO THreaD3 AN$ A S-All DELAY, OnlY so-E LETTeRS CHANG%.

user@host:$ ./hilos_eco_mayus 5 10
Type lines of text (Ctrl+D to finish):
With 5 threads and a significant delay, barely any letter survives.
W       4H 5 T(2E!DS !Nd ! 3))& #A.T DE
                                        AY, bARELY ANY LE4TER 3U
V%S.
```

Compile the program[12]. Execute it increasing the number of threads and the delay until observing the aforementioned effect.

Fix the program making the threads execute lines 154-161 in mutual exclusion. It is only allowed to add code[13]. Do not delete, amend or cancel any of the existing lines of code.

After fixing the program, verify that no letters are corrupted, even though many threads are executed and/or a long delay is used.

[12] Do not forget to add the option `-pthread`

[13] Three lines will suffice

Appendixes:

A Review of the `fork()` system call

When the kernel creates a new process, it does so by making a copy —a clone— of the process that requested it through the `fork()`[14] system call. The process that invoked `fork()` is then called "parent process", and the new process is called "child process". Thus, except for the PID[15] and the PPID[16], the two processes will be initially identical. This way, new processes obtain a copy of the resources of their parent process —they inherit the environment—. However, in principle they do not run any new external program. To achieve this, a process must use another system call: `exec()`, whose mission consists in resetting, or rather replacing or "covering" the user-data and instructions segments of the process with those of other program, read from the disc. At this point, the operating system does not create any new process, it just replaces some key parts of an existing process.

When a process ends —dies—, the operating system deletes it, recovering its resources in order to make them available to other processes.

The `fork()` system call creates an exact duplicate of the original process —who made the call—. After the call to `fork()`, the original and the copy process —parent and child— continue to run separately. Right after the call, all variables of the two processes have the same value, but the successive changes in the data of one of them will not affect the other. The user-data and system-data segments of the new process —child— are almost exact copies of those of the calling process —parent—. The code, being immutable during the whole lifetime of the processes, is shared between parent and child.

The child process inherits most of the parent's attributes, since they are copied from its system-data segment. Only some elements differ between parent and child:

- Value returned by `fork()` —zero for the child; child's PID for the parent.
- Process identifier (PID) and parent process identifier (PPID).
- Files: The child obtains a copy of the parent's file descriptor table. Thus they share the pointers to files. That is, both processes point to the same entries in the open files table. Therefore, if one of them modifies the entry referenced by that pointer, for instance making an I/O operation —and changing the current position in the file—, then the next I/O operation made by the other process will be affected by the change —it will start in the new current position—. However, since there are two separate descriptor tables, if a process closes its descriptor, the other is not affected.

It is usually desirable that parent and child perform different tasks. Since the code is the same, the key to distinguish them is the value returned by `fork()`. In the child process this value is always zero, while in the parent process this value will be the PID of the child process if it has been created successfully, or `-1` if `fork()` failed, not being able to create the new process.

[14] to fork: *bifurcarse*
[15] PID: identifier of the current process
[16] PPID: identifier of the parent of the current process

Example (the order of execution of processes and their PIDs do not have to match those shown here):

```
user@host:$ cat example_fork.c
#include <stdio.h>      // printf(), ...
#include <unistd.h>     // fork(), getpid(), getppid(), ...

int main ()
{
    int i;

    printf ("I'm %d, child of %d.\n", getpid(), getppid());

    i = fork ();

    printf ("I'm %d, child of %d, and fork() returned to me %d\n",
            getpid(), getppid(), i);

    return 0;
}

user@host:$ gcc -Wall example_fork.c -o example_fork
user@host:$ ps
  PID TTY          TIME CMD
 2309 pts/0    00:00:00 bash
 2957 pts/0    00:00:00 ps
user@host:$ ./example_fork
I'm 2958, child of 2309.
I'm 2958, child of 2309, and fork() returned to me 2961
I'm 2961, child of 2958, and fork() returned to me 0
user@host:$
```

Note that the parent —with PID 2958— receives form `fork()` the child's PID —2961—, while the child receives from `fork()` a zero value.

B Review of the `exec()` system call

UNIX-like operating systems offer a family of system calls that allow launching the execution of a program, stored as an executable file. Although there is basically only one system call —`exec()`—, the standard C libraries have several functions that use this call. They differ in the way they pass parameters to the executable. The functions of the `exec()` family replace the image in memory of the current process —the caller— with a new program specified as argument. This action has the effect of making a process run a different code from the one it was running before the call to `exec()` —without changing its PID.

The typical call used when we know a priori the number of arguments to be passed to the new program, is `execl()`. Its syntax is:

```
int execl (const char * path, const char * arg, ...);
```

The name —and path— of the executable file is passed first, and then all the arguments follow —including by convention the name of the executable—, ending with a NULL pointer.

For example, in order to execute the command "`/bin/ls -l /usr/include`", we would call:

```
execl ("/bin/ls", "ls", "-l", "/usr/include", NULL);
```

In case of not knowing in advance the number of arguments, we must use the function `execv()`, which has the following syntax:

```
int execv (const char * path, char * const argv[]);
```

The parameter `argv` is an array of pointers to strings of characters that contain the arguments for the launched program, being the last element a NULL pointer. The previous example could be solved as follows:

```
char * strings[] = {"ls", "-l", "/usr/include", NULL};
...
execv ("/bin/ls", strings);
```

In the previous examples, the full path and name of the executable were specified —"`/bin/ls`" instead of just "`ls`"—. This was due to the fact that both `execl()` and `execv()` ignore the `PATH` environment variable, which contains the default paths where executables should be searched. In order to take this variable into account, we can use the functions `execlp()` or `execvp()`, of the same family. The next call, for example, would execute the program "`/bin/ls`" if the path "`/bin`" were included in the variable `PATH`:

```
execvp ("ls", strings);
```

All the `exec..()` calls return the value -1 if the program can not be executed. Otherwise, if the program can be executed, then the control is transferred to it, and the `exec..()` call never returns to the caller program.

C Pipes and deadlocks

Section 2.3.2 presented the issue of the deadlocks that might occur when processes communicate using named pipes. This appendix will discuss two possible solutions.

C.1 Enforcing the order in which operations are carried out

One possible solution is to ensure that all processes follow the same order when they open/read-/write in several pipes. The idea is not, for example, opening first the pipe for reading and then the pipe for writing, but the other way around; the trick consists in having both processes opening first the pipe A —one for reading and the other one for writing— and then the pipe B —one for writing and the other one for reading.

In the case of Section 2.3.2, the difficulty is in the fact that it is the user who provides the names of the pipes to every process. The names themselves are not reliable enough to establish an order, because a single pipe might have several names —hard links—. The order must be established basing on something that identifies uniquely each pipe, providing the same "id" value for anyone who accesses it. That something is the couple {file system id., number of index-node}. These data can be obtained using the `stat()` system call.

The weakness of this approach is that the information provided by `stat()` to a process might not be valid right before it opened the pipes, if another process changed the names in that small time interval —linking those names with other pipes, for example.

C.2 Opening pipes in non-blocking mode

Another possible solution consists in using the `O_NONBLOCK` flag to open the pipes for reading, and then opening normally —without `O_NONBLOCK`— the pipes for writing.

This way, the initial opening progresses without deadlocks, but we loose the warranty that there will be somebody ready to write at the other end of the pipes opened for reading. With the pipes opened in this mode for reading, `read()` never gets blocked, returning a zero value when there are no data to be read[17].

Therefore, once all the pipes have been opened, we should put them in blocking mode by removing the `O_NONBLOCK` flag. This can be achieved using `fcntl()`.

However, even after removing the `O_NONBLOCK` flag, the first `read()` can return 0 without blocking if no process has opened the pipe for writing. Furthermore, the problem of deadlocks in read operations remains unsolved. Both problems can be solved by calling `select()` before trying to read, in order to wait until there is something to read and, at the same time, find out which is the pipe that contains these data.

[17] That's bad