Part IV

Concurrency and distribution

The fourth part introduces the concepts of parallel programming and presents models for shared and distributed memory. It is not necessary to have access to a parallel supercomputer to express concurrent algorithms or to implement distributed applications. In this preamble we define the different terms used in the following chapters.

In sequential programming, one instruction is executed after another. This is called causal dependency. Sequential programs have the property of being deterministic. For the same input, one program will always terminate or never terminate. In the case of termination, always the same result will be produced. Determinism implies that the same circumstances will always lead to the same effects. The only exceptions, which we have already met in Objective Caml, are provided by functions taking external information as input, such as the function `Sys.time`.

In parallel programming, a program is split into several active processes. Each process is sequential, but several instructions, belonging to different processes, are executed in parallel, "at the same time." The sequence is transformed into concurrency. This is called causal independence. The same circumstances may lead to different, mutually exclusive effects (only one effect is produced). An immediate consequence is the loss of determinism: the same program with the same input may or may not terminate. In the case of termination different results may be produced.

In order to control the execution of a parallel program, it is necessary to introduce two new notions:

- synchronization, which introduces a conditional wait to several processes;
- communication through the passing of messages between processes.

From the point of view of causality, synchronization assures that several independent circumstances have to be reproduced before an effect may take place. Communications have a temporal constraint: a message can not be received before it is sent. Communication can occur in different variants: communication directed from one process to one other (point-to-point) or as distribution (one-to-all, or all-to-all).

The two models of parallel programming described by figure 17.13 differ in execution control by the forms of synchronization and communication.

Each process P_i corresponds to a sequential process. The set of these processes, interacting via shared memory (**M**) or via a *medium*, constitutes a parallel application.

Shared Memory Model Communication is implicit in the shared memory model. An information is given through writing into a zone of the shared memory. It is received when another process reads this zone. The synchronization, in contrast, has to be explicit. Constructs of mutual exclusion and waiting conditions are used.

This model is used when shared resources are used in a concurrent way. The construction of operating systems can be cited as an example.

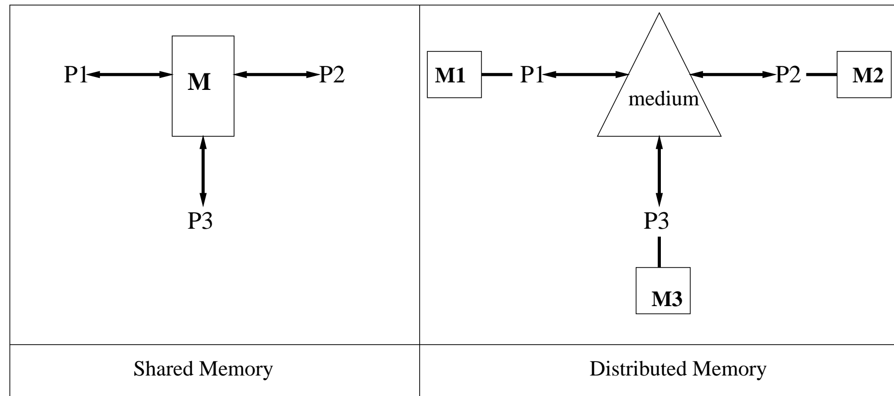

Figure 17.13: Models of parallelism

Distributed Memory Model In this model each sequential process P_i has a private memory M_i, to which no other process has access. The processes have to communicate in order to transmit information through a medium. The difficulties in this model arise from the implementation of the medium. The programs which care for this are called *protocols*.

Protocols are organized in layers. The higher-level protocols implement more elaborate services, using the lower-level services.

There exist several types of communication. They depend on the capability of the medium to store information and of the blocking, respectively non-blocking character of sender and receiver. We talk about synchronous communication when the transfer of information is not possible before a global synchronization between sender and receiver takes place. In his case both sender and receiver may be blocked.

If the medium has the storage capabilities, it can store messages for a later transmission. Therefore the communication can be asynchronous and non-blocking. It may be necessary to indicate the storage capacity of the medium, the order of transmission, the delay and the reliability of transmissions.

Finally, if the transmission is non-blocking with a medium not able to store messages, a volatile communication results: only the receiving processes which are ready will receive the sent message, which is lost for the other processes.

In the model of distributed memory the communication is explicit, but the synchronization is implicit (Synchronization is produced by communication). It is dual to the model of shared memory.

Physical and Logical Parallelism The model of distributed memory is valid in the case of physical and logical parallelism. Physical parallelism refers for example to a computer network. Examples for logical parallelism are Unix processes communicating

via *pipes*, or lightweight processes communicating via channels. There are no common global values known by all proceses like, for example, a global clock.

The model of distributed memory is closer to physical parallelism, where there is no effectively shared memory. Nevertheless, shared memory can be simulated across a computer nerwork.

The fourth part will show how to construct parallel applications with Objective Caml using the two presented models. It relies on the `Unix` library, which interfaces Unix system calls to Objective Caml, and on the `Thread` library, which implements lightweight processes. A major part of the `Unix` library is ported to Windows, especially the functions on file descriptors. These are used to read and to write on files, but also for communication *pipes* and for *sockets* of computer networks.

Chapter 18 describes essential concepts of the `Unix` library. It concentrates on the communication of a process with it's exterior and with other processes. The notion of process in this chapter is that of a "heavyweight process" as in Unix. They are created by the `fork` system call which duplicates the execution context and the memory for the data, producing a chain of processes. The interaction between processes is implemented by signals or by communication pipes.

Chapter 19 concentrates on the notion of lightweight processes of the `Thread` library. In contrast to the heavy processes mentioned before, they duplicate nothing but the execution context of an existing process. The memory is shared between the creator and the *thread*. Depending on the programming style, the light Objective Caml processes permit to use the parallelism model of shared memory (imperative style) or the model of separated memory (purely functional style). The `Thread` library contains several modules allowing to start and stop *threads*, to manage locks for mutual exclusion, to wait for a condition and to communicate between *threads* via channels. In this model, there is no gain in execution time, not even for multi-processor machines. But the formulation of parallel algorithms is made easier.

Chapter 20 is devoted to the construction of distributed Internet applications. The Internet is presented from the point of view of low-level protocols. With the help of communication sockets several processes running on different machines are able to communicate with each other. The communication through sockets is an asynchronous point-to-point communication. The role of the different processes taking part in the communication of a distributed application is in general asymmetrical. This is the case for client-server architectures. The server is a process accepting requests and trying to respond. The other process, the client, sends a request to the server and waits for a response. Many services accessible in the Internet follow this architecture.

Chapter 21 presents a library and two complete applications. The library allows to define the communication between clients and servers starting from a given protocol. The first application revisits the robots of chapter 17 to give a distributed version. The second application constructs an HTTP server to manage a request form taking up again the management of associations presented in chapter 6.

18

Communication and Processes

This chapter approaches two important aspects of the interface between a programming language and the operating system: communication and processes. The `Sys` module presented in chapter 8 has already shown how to pass values to a program and how to start a program from another one. The goal of this chapter is to discuss the notions of processes and communication between processes.

The term “process” is used for an executing program. Processes are the main components of a parallel application. We introduce processes in the classical way originating from the Unix system. In this context a process is created by another process, establishing a parent-child relationship between them. This relationship allows the parent to wait for the child to terminate, as well as to set up privileged communications between the two. The underlying model of parallelism is that of distributed memory.

The term “communication” covers three aspects:

- input and output via *file descriptors*. The notion of file descriptors under Unix has a much broader meaning than the simple reading or writing of data from or to a storage medium. We will see this in chapter 20, where programs running on different machines are communicating via such descriptors;
- the use of *pipes* between processes which allow the exchange of data using the principle of waiting queues;
- the generation and handling of *signals*, which allow a simple interaction between processes.

The functions presented in this chapter are similar to those in the `Unix` module which accompanies the Objective Caml distribution. The terminology and the notions come from the Unix world. But many of the functions of this module can also be used under Windows. Later we will indicate the applicability of the presented functions.

Chapter Overview

The first section indicates how to use the `Unix` module. We will talk about the handling of errors specific to that module and about the portability of system calls to Windows.

The second section presents file descriptors in the Unix sense, and their use for input and output operations of a lower level than those provided by the preloaded module `Pervasives`.

Processes are introduced in the third section. We talk about their creation, their disappearance and about the way in which all processes support their descendence relation in the Unix model.

The fourth section describes the basic means of communications between processes: pipes and signals.

The two last sections will be continued in chapters 19 and 20 by the presentation of lightweight processes and sockets.

The `Unix` Module

This module contains the interfaces to the most important Unix library functions. Much of this module has been ported to Windows and can be used there. Whenever necessary we will indicate the restrictions in the use of the presented functions. A table resuming the restrictions is given by figure 18.1.

The `Unix` library belongs to the Objective Caml non-standard libraries which have to be bound by the `-custom` compiler command (see chapter 7, page 197). Depending on the desired form of the program, one of the following commands is used under Unix to produce bytecode, native code or an interaction loop:

```
$ ocamlc -custom unix.cma  fichiers.ml -cclib -lunix
$ ocamlopt unix.cma fichiers.ml -cclib -lunix
$ ocamlmktop -custom -o unixtop unix.cma -cclib -lunix
```

The purpose of constructing an interaction loop (of which the name will be `unixtop`) is to support an incremental development style. Each function can be compiled quickly from its type declaration. It is also possible to execute functional tests.

Depending on the version of Unix in use, the system library may not be located at the default place. If necessary the access path of the libraries may be indicated with the option `-ccopt` (see chapter 7).

Under Windows the commands to compile become:

```
$ ocamlc -custom unix.cma  fichiers.ml %CAMLLIB%\libunix.lib wsock32.lib
$ ocamlopt unix.cma fichiers.ml %CAMLLIB%\libunix.lib wsock32.lib
$ ocamlmktop -custom -o unixtop.exe unix.cma %CAMLLIB%\libunix.lib wsock32.lib
```

The name of the obtained interaction loop is `unixtop.exe`.

Error Handling

Errors produced by system calls throw `Unix_error` exceptions, which can be handled by the Objective Caml program. Such errors contain three arguments: a value of type `Unix.error` which can be transformed into a character string by the function `error_message`, a string containing the name of the function producing the error and optionally, a string containing the argument of the function when the argument is of type *string*.

It is possible to define a generic calling function with error treatment:

```
# let wrap_unix funct arg =
    try (funct arg) with
     Unix.Unix_error (e,fm,argm) →
       Printf.printf "%s %s %s" (Unix.error_message e) fm argm ;;
val wrap_unix : ('a -> unit) -> 'a -> unit = <fun>
```

The function `wrap_unix` takes a function and its argument, and applies one to the other. If a Unix error occurs, an explaining message is printed. An equivalent function is defined in the `Unix` module:

```
# Unix.handle_unix_error ;;
- : ('a -> 'b) -> 'a -> 'b = <fun>
```

Portability of System Calls

Figure 18.1 indicates which of the communication and process handling functions presented in this chapter are accessible under Windows. The main shortcoming is the lack of the two functions `fork` and `kill` to create new processes and to send signals.

Furthermore, the function `wait` waiting for the end of a child process is not implemented, because `fork` is not.

File Descriptors

In chapter 3 we have seen functions from the standard module `Pervasives`. These functions allow us to access files via input / output channels. There is also a lower-level way to access files, using their descriptors.

A file descriptor is an abstract value of type `Unix.file_descr`, containing information necessary to use a file: a pointer to the file, the access rights, the access modes (read or write), the current position in the file, etc.

Three descriptors are predefined. They correspond to standard input, standard output, and standard error.

Fonction	Unix	Windows	Comment
`openfile`	×	×	
`close`	×	×	
`dup`	×	×	
`dup2`	×	×	
`read`	×	×	
`write`	×	×	
`lseek`	×	×	
`execv`	×	×	
`execve`	×	×	
`execvp`	×	×	
`execvpe`	×	×	
`fork`	×		use `create_process`
`getpid`	×	×	
`sleep`	×	×	
`wait`	×		
`waitpid`	×	×	only for a given number of processes
`create_process`	×	×	
`create_process_env`	×	×	
`kill`	×		
`pipe`	×	×	
`mkfifo`	×		
`open_process`	×		use the interpretation of `/bin/sh` commands
`close_process`	×		

Figure 18.1: Portability of the module `Unix` functions used in this chapter.

```
# ( Unix.stdin , Unix.stdout , Unix.stderr ) ;;
- : Unix.file_descr * Unix.file_descr * Unix.file_descr =
<abstr>, <abstr>, <abstr>
```

Be careful not to confuse them with the corresponding input / output channels:

```
# ( Pervasives.stdin , Pervasives.stdout , Pervasives.stderr ) ;;
- : in_channel * out_channel * out_channel = <abstr>, <abstr>, <abstr>
```

The conversion functions between channels and file descriptors are described at page 577.

File Access Rights. Under Unix each file has an associated owner and group. The rights to read, write and execute are attached to each file according to three categories of users: the owner of a file, the members of the file's group[1] and all other users.

The access rights of a file are represented by 9 bits divided into three groups of three bits each. The first group represents the rights of the owner, the second the rights of the members of the owner's group, and the last the rights of all other users. In each group of three bits, the first bit represents the right to read, the second bit the right to write and the third bit the right to execute. It is common to abbreviate these three rights by the letters `r`, `w` and `x`. The absence of the rights is represented in each case by a dash (`-`). For exampple, the right to read for all and the right to write only for the owner is written as `rw-r--r--`. This corresponds to the integer `420` (which is the binary number `0b110100100`). Frequently the more comfortable octal notation `0o644` is used. These file access rights are not used under Windows.

REVIEWER'S QUESTION: IS THIS STILL TRUE UNDER WIN2K?

File Manipulation

Opening a file. Opening a file associates the file to a file descriptor. Depending on the intended use of the file there are several modes to open a file. Each mode corresponds to a value of type *open_flag* described by figure 18.2.

`O_RDONLY`	read only
`O_WRONLY`	write only
`O_RDWR`	reading and writing
`O_NONBLOCK`	non-blocking opening
`O_APPEND`	appending at the end of the file
`O_CREAT`	create a new file if it does not exist
`O_TRUNC`	set the file to 0 if it exists
`O_EXCL`	chancel, if the file already exists

Figure 18.2: Values of type *open_flag*.

These modes can be combined. In consequence, the function `openfile` takes as argument a list of values of type *open_flag*.

```
# Unix.openfile ;;
- : string -> Unix.open_flag list -> Unix.file_perm -> Unix.file_descr =
```

1. Under Unix each user belongs to one or more user groups, which allows the organization of their rights.

```
<fun>
```

The first argument is the name of the file. The last is an integer[2] coding the rights to attach to the file in the case of creation.

Here is an example of how to open a file for reading, or to create it with the rights `rw-r--r--` if it does not exist:

```
# let file = Unix.openfile "test.dat" [Unix.O_RDWR; Unix.O_CREAT] 0o644 ;;
val file : Unix.file_descr = <abstr>
```

Closing a file. The function `Unix.close` closes a file. It is applied to the descriptor of the file to close.

```
# Unix.close ;;
- : Unix.file_descr -> unit = <fun>
# Unix.close file ;;
- : unit = ()
```

Redirecting file descriptors. It is possible to attach several file descriptors to one input / output. If there is only one file descriptor available and another one is desired we can use:

```
# Unix.dup ;;
- : Unix.file_descr -> Unix.file_descr = <fun>
```

If we have two file descriptors and we want to assign to the second the input / output of the first, we can use the function:

```
# Unix.dup2 ;;
- : Unix.file_descr -> Unix.file_descr -> unit = <fun>
```

For example, the error output can be directed to a file in the following way:

```
# let error_output = Unix.openfile "err.log" [Unix.O_WRONLY;Unix.O_CREAT] 0o644 ;;
val error_output : Unix.file_descr = <abstr>
# Unix.dup2 Unix.stderr error_output ;;
- : unit = ()
```

Data written to the standard error output will now be directed to the file `err.log`.

Input / Output on Files

The functions to read and to write to a file `Unix.read` and `Unix.write` use a character string as medium between the file and the Objective Caml program.

```
# Unix.read ;;
- : Unix.file_descr -> string -> int -> int -> int = <fun>
# Unix.write ;;
```

2. The type *file_perm* is an alias for the type *int*.

```
- : Unix.file_descr -> string -> int -> int -> int = <fun>
```

In addition to the file descriptor and the string the functions take two integers as arguments. One is the index of the first character and the other the number of characters to read or to write. The returned integer is the number of characters effectively read or written.

```
# let mode = [Unix.O_WRONLY;Unix.O_CREAT;Unix.O_TRUNC] in
  let fl = Unix.openfile "file" mode 0o644 in
  let str = "012345678901234565789" in
  let n = Unix.write fl str 4 5
  in Printf.printf "We wrote %s to the file\n" (String.sub str 4 n) ;
     Unix.close fl ;;
We wrote 45678 to the file
- : unit = ()
```

Reading a file works the same way:

```
# let fl = Unix.openfile "file" [Unix.O_RDONLY] 0o644 in
  let str = String.make 20 '.' in
  let n = Unix.read fl str 2 10 in
   Printf.printf "We read  %d characters" n;
   Printf.printf " and got the string %s\n" str;
   Unix.close fl ;;
We read  5 characters and got the string ..45678.............
- : unit = ()
```

Access to a file always takes place at the current position of its descriptor. The current position can be modified by the function:

```
# Unix.lseek ;;
- : Unix.file_descr -> int -> Unix.seek_command -> int = <fun>
```

The first argument is the file descriptor. The second specifies the displacement as number of characters. The third argument is of type `Unix.seek_command` and indicates the origin of the displacement. The third argument may take one of three posssible values:

- `SEEK_SET`: relative to the beginning of the file,
- `SEEK_CUR`: relative to the current position,
- `SEEK_END`: relative to the end of the file.

A function call with an erronous position will either raise an exception or return a value equal to 0.

Input / output channels. The `Unix` module provides conversion functions between file descriptors and the input / output channels of module `Pervasives`:

```
# Unix.in_channel_of_descr  ;;
- : Unix.file_descr -> in_channel = <fun>
```

```
# Unix.out_channel_of_descr ;;
- : Unix.file_descr -> out_channel = <fun>
# Unix.descr_of_in_channel  ;;
- : in_channel -> Unix.file_descr = <fun>
# Unix.descr_of_out_channel ;;
- : out_channel -> Unix.file_descr = <fun>
```

It is necessary to indicate whether the input / output channels obtained by the conversion transfer binary data or character data.

```
# set_binary_mode_in ;;
- : in_channel -> bool -> unit = <fun>
# set_binary_mode_out ;;
- : out_channel -> bool -> unit = <fun>
```

In the following example we create a file by using the functions of module `Unix`. We read using the opening function of module `Unix` and the higher-level input function `input_line`.

```
# let mode = [Unix.O_WRONLY;Unix.O_CREAT;Unix.O_TRUNC] in
  let f = Unix.openfile "file" mode 0o666 in
  let s = "0123456789\n0123456789\n" in
  let n = Unix.write f s 0 (String.length s)
  in Unix.close f ;;
- : unit = ()
# let f = Unix.openfile "file" [Unix.O_RDONLY;Unix.O_NONBLOCK] 0 in
  let c = Unix.in_channel_of_descr f in
  let s = input_line c
  in print_string s ;
     close_in c ;;
0123456789- : unit = ()
```

Availability. A program may have to work with multiple inputs and outputs. Data may not always be available on a given channel, and the program cannot afford to wait for one channel to be available while ignoring the others. The following function lets you determine which of a given list of inputs/outputs is available for use at a given time:

```
# Unix.select ;;
- : Unix.file_descr list ->
    Unix.file_descr list ->
    Unix.file_descr list ->
    float ->
    Unix.file_descr list * Unix.file_descr list * Unix.file_descr list
= <fun>
```

The first three arguments represent lists of respectively inputs, of outputs and error--outputs. The last argument indicates a delay in seconds. A negative value means the null delay. The results are the lists of available input, output and error-output.

Warning `select` is not implemented under Windows

Processes

Unix associates a *process* with each execution of a program. In [CDM98] Card, Dumas and Mével describe the difference between a program and a process: "a program itself is not a process: a program is a passive entity (an executable file on a disc), while a process is an active entity with a counter specifying the next instruction to execute and a set of associated resources."

Unix is a *multi-task* operating system: many processes may be executed at the same time. It is *preemptive*, which means that the execution of processes is entrusted to a particular process. A process is therefore not totally master of its resources. Especially a process can not determine the time of its execution. A process has to be created.

Each process has his own private memory space. Processes can communicate via files or communication channels. Thus the distributed memory model of parallelism is simulated on a single machine.

The system gives each process a unique identifier: the PID (Process IDentifier). Under Unix each process, except the initial process, is created by another process, which is called its *parent*.

The set of all active processes can be listed by the Unix command `ps`[3]:

```
$ ps -f
PID    PPID    CMD
1767   1763   csh
2797   1767   ps -f
```

The use of the option `-f` adds for each active process its identifier (`PID`), that of its parent (`PPID`) and the name of the started program (`CMD`). Here we have two processes, the command line interpreter `csh` and the command `ps` itself. It can be seen that `ps` has been started from the command line interpreter `csh`. The parent of its process is the process associated with the execution of `csh`.

Executing a Program

Execution Context

Three values are associated with an executing program, which is started from the command line:

1. The command line used to start it. It is contained in the value `Sys.argv`.
2. The environment variables of the command line interpreter. These can be accessed by the command `Sys.getenv`.

3. The options and the behavior of this command are not standardized. The given example may not be reproducible.

3. An execution status until the program is terminated.

Command line. The command line allows you to read arguments or options of a program call. The behavior of the program may depend from these values. Here is a small example. We write the following program into the file `argv_ex.ml`:

```
if Array.length Sys.argv = 1 then
 Printf.printf "Hello world\n"
else if Array.length Sys.argv = 2 then
 Printf.printf "Hello %s\n" Sys.argv.(1)
else Printf.printf "%s : too many arguments\n" Sys.argv.(0)
```

We compile it:

```
$ ocamlc -o argv_ex argv_ex.ml
```

And we execute it:

```
$ argv_ex
Hello world
$ argv_ex reader
Hello reader
$ argv_ex dear reader
./argv_ex : too many arguments
```

Environment variables. Environment variables may contain values necessary for execution. The number and the names of these variables depend on the operating system and on the user configuration. The values of these variables can be accessed by the function `getenv`, which takes as argument the name of a variable in form of a character string:

```
# Sys.getenv "HOSTNAME";;
- : string = "zinc.pps.jussieu.fr"
```

Execution Status

The return value of a program is generally a fixed integer, indicating if the program did terminate with an error or not. The exact values may differ from one operating system to another. The programer can always explicitly stop his program and return the execution status value with the function call:

```
# Pervasives.exit ;;
- : int -> 'a = <fun>
```

Process Creation

A program is started by another process, which is called the current process. The executed program becomes a new process. There are three different relations between the two processes:

- The two processes are independent from each other and can be executed concurrently.
- The parent process is waiting for the child process to terminate.
- The created process replaces the parent process, which terminates.

It is also possible to duplicate the current process to obtain two instances. The two instances of the process do not differ but in their PID. This is the famous `fork` which we will describe later.

Independent Processes

The `Unix` module offers a portable function to create a process.

```
# Unix.create_process ;;
- : string ->
    string array ->
    Unix.file_descr -> Unix.file_descr -> Unix.file_descr -> int
= <fun>
```

The first argument is the name of the program (it may be a path). The second is the array of arguments for the program. The last three arguments are the descriptors indicating the standard input, standard output and standard error output of the process. The return value is the PID of the created process.

There also exists a variant of this function which allows you to indicate the values of environment variables:

```
# Unix.create_process_env ;;
- : string ->
    string array ->
    string array ->
    Unix.file_descr -> Unix.file_descr -> Unix.file_descr -> int
= <fun>
```

These two functions can be used under Unix and Windows.

GGH

Process Stacks

It is not always useful for a created process to be of concurrent nature. The parent process may have to wait for the created process to terminate. The two following functions take as argument the name of a command and execute it.

```
# Sys.command;;
- : string -> int = <fun>
# Unix.system;;
- : string -> Unix.process_status = <fun>
```

They differ in the type of the return code. The type *process_status* is explained in more detail on page 586. During the execution of the command the parent process is blocked.

Replacement of Current Processes

The replacement of current processes by freshly created processes allows you to limit the number of concurrently executed processes. The four following functions allow this:

```
# Unix.execv ;;
- : string -> string array -> unit = <fun>
# Unix.execve ;;
- : string -> string array -> string array -> unit = <fun>
# Unix.execvp ;;
- : string -> string array -> unit = <fun>
# Unix.execvpe ;;
- : string -> string array -> string array -> unit = <fun>
```

Their first argument is the name of the program. Using `execvp` or `execvpe`, this name may indicate a path in the file system. The second argument contains the program arguments. The last argument of the functions `execve` and `execvpe` additionally allows you to indicate the values of system variables.

Creation of Processes by Duplication

The original system call to create processes under Unix is:

```
# Unix.fork ;;
- : unit -> int = <fun>
```

The function `fork` starts a new process, not a new program. Its effect is to *duplicate* the calling process. The code of the new process is the same as that of its parent. Under Unix the same code can be shared by several processes, each process possessing its own execution context. Therefore we speak about *reentrant code*.

Let's look at the following small program (we use the function `getpid` which returns the PID of the process associated with the execution):

```
Printf.printf "before fork : %d\n" (Unix.getpid ()) ;;
flush stdout ;;
Unix.fork () ;;
Printf.printf "after fork : %d\n" (Unix.getpid ()) ;;
flush stdout ;;
```

We obtain the following output:

```
before fork : 10529
after fork : 10529
after fork : 10530
```

After the execution of `fork`, two processes execute the code. This leads to the output of two PID's "`after`" the `fork`. We note that one process has kept the PID of the

beginning (the parent). The other one has a new PID (the child), which corresponds to the return value of the `fork` call. For the parent process the return value of `fork` is the PID of the child, while for the child, it is `0`.

It is this difference in the return value of `fork` which allows *in one program source* to decide which code shall be executed by the child and which by the parent:

```
Printf.printf "before fork : %d\n" (Unix.getpid ())  ;;
flush stdout ;;
let pid = Unix.fork () ;;
if pid=0 then  (* -- Code of the child *)
  Printf.printf "I am  the child: %d\n" (Unix.getpid ())
else           (* -- Code of the father *)
  Printf.printf "I am the father: %d of child: %d\n"  (Unix.getpid ()) pid ;;
flush stdout ;;
```

Here is the trace of the execution of this program:

```
before fork : 10539
I am the father: 10539 of child: 10540
I am  the child: 10540
```

It is also possible to use the return value for matching:

```
match Unix.fork () with
   0  → Printf.printf "I am the child: %d\n" (Unix.getpid ())
| pid → Printf.printf "I am the father: %d of child: %d\n"
                       (Unix.getpid ()) pid ;;
```

The fertility of a process may be very big. Therefore the number of descendents of a process is limited by the configuration of the operating system. The following example creates two generations of processes with grandparent, parents, uncles and cousins.

```
let pid0 = Unix.getpid ();;
let print_generation1 pid ppid =
  Printf.printf "I am %d, son of %d\n" pid ppid;
  flush stdout ;;

let print_generation2 pid ppid pppid  =
  Printf.printf "I am %d, son of %d, grandson of %d\n"
                pid ppid pppid;
  flush stdout ;;

match Unix.fork() with
    0 → let pid01 = Unix.getpid ()
        in ( match Unix.fork() with
                0 → print_generation2 (Unix.getpid ()) pid01 pid0
              | _ → print_generation1 pid01 pid0)
  | _ → match Unix.fork () with
            0 → ( let pid02 = Unix.getpid ()
                  in match Unix.fork() with
                        0 → print_generation2 (Unix.getpid ()) pid02 pid0
                      | _ → print_generation1 pid02 pid0 )
          | _ → Printf.printf "I am %d, father and grandfather\n" pid0 ;;
```

We obtain:

```
I am 10644, father and grandfather
I am 10645, son of 10644
I am 10648, son of 10645, grandson of 10644
I am 10646, son of 10644
I am 10651, son of 10646, grandson of 10644
```

Order and Moment of Execution

A sequence of process creations without synchronization may lead to surprising effects. This is illustrated by the following poem writing program à la M. Jourdain[4]:

```
match Unix.fork () with
  0 → Printf.printf "fair Marquise " ; flush stdout
| _ → match Unix.fork () with
          0 → Printf.printf "your beautiful eyes " ; flush stdout
        | _ → match Unix.fork () with
                  0 → Printf.printf "make me die " ; flush stdout
                | _ → Printf.printf "of love\n" ;  flush stdout ;;
```

It may produce the following result:

```
of love
fair Marquise your beautiful eyes make me die
```

We usually want our program to be able to assure the order of execution of its processes. More generally speaking, an application which makes use of several processes may have to synchronize them. Depending on the model of parallelism in use, the synchronization is realized by communication between the processes or by waiting conditions. This subject is presented more profoundly by the two following chapters. For the moment, we can improve our poem writing program in two ways:

- Give the child the time to write its phrase before writing the own.
- Wait for the termination of the child, which will then have written its phrase, before writing our own phrase.

Delays. A process can suspend its activity by calling the function:

```
# Unix.sleep ;;
- : int -> unit = <fun>
```

The argument provides the number of seconds during which the process wants to suspend its activities.

Using this function, we write:

4. Molière, *Le Bourgeois Gentilhome*, Acte II, scène 4.
Link: http://www.site-moliere.com/pieces/bourgeoi.htm

```
match Unix.fork () with
    0 → Printf.printf "fair Marquise " ; flush stdout
  | _ → Unix.sleep 1 ;
         match Unix.fork () with
             0 → Printf.printf"your beautiful eyes "; flush stdout
           | _ → Unix.sleep 1 ;
             match Unix.fork () with
                 0 → Printf.printf"make me die "; flush stdout
               | _ → Unix.sleep 1 ; Printf.printf "of love\n" ; flush stdout ;;
```

And we can obtain:
`fair Marquise your beautiful eyes make me die of love`
Nevertheless, this method is not sure. In theory, it would be possible that the system gives enough time to one of the processes to sleep and to write its output at the same turn. Therefore we prefer the following method for assuring the execution order of our processes.

GGH

Waiting for the termination of the child. A parent process may wait for his child to terminate through a call to the function:

```
# Unix.wait ;;
- : unit -> int * Unix.process_status = <fun>
```

The execution of the parent is suspended until one of its children terminates. If `wait` is called by a process not having any children, a `Unix_error` is thrown. We will discuss later the return value of `wait`. For the moment, we will just use the command to pronounce our poem:

```
match Unix.fork () with
    0 → Printf.printf "fair Marquise " ; flush stdout
  | _  → ignore (Unix.wait ()) ;
          match Unix.fork () with
              0 → Printf.printf "your beautiful eyes " ; flush stdout
            | _ → ignore (Unix.wait ()) ;
                  match Unix.fork () with
                      0 → Printf.printf "make me die " ; flush stdout
                    | _ → ignore (Unix.wait ()) ;
                   Printf.printf "of love\n" ;
                   flush stdout
```

Indeed, we obtain:
`fair Marquise your beautiful eyes make me die of love`

Warning `fork` is proprietary to the Unix system

Descendence, Death and Funerals of Processes

The function `wait` is useful not only to wait for the termination of a child. It also has the responsibility to complete the death of the child process.

Whenever a process is created, the system adds an entry in a table. The table serves to keep track of all processes. When a process terminates, the entry does not disappear automatically in the table. It is the responsibility of the parent to assure the deletion by the call of `wait`. If this is not done, the child process keeps an entry in the table. This is called a *zombie* process.

When the system is started, a first process called `init` is started. After the initialization of some parameters, the essential role of this "forefather" is to take care of orphan processes and to call the `wait` which deletes them from the process table after their termination.

Waiting for the Termination of a Given Process

There is a variation of the function `wait`, named `waitpid`. This command is supported on Unix and Windows:

```
# Unix.waitpid ;;
- : Unix.wait_flag list -> int -> int * Unix.process_status = <fun>
```

The first argument specifies the waiting modalities. The second indicates which process or which group of processes are treated.

After the termination of a process, two pieces of information can be accessed by its parent as a result of the function calls `wait` or `waitpid`: the number of the terminated process and its exit status. The status is represented by a value of type *Unix.process_status*. This type has three constructors. Each of them takes an integer as argument.

- `WEXITED n`: the process has terminated normally with the return code `n`.
- `WSIGNALED n`: the process has been killed by the signal `n`.
- `WSTOPPED n`: the process has been stopped by the signal `n`.

The last value only makes sense for the function `waitpid` which can listen for such signals as indicated by its first argument. We will discuss signals and their treatment at page 590.

Managing of Waiting by Ancestors

In order to avoid having to care for the termination of child processes oneself, it is possible to delegate this responsibility to an ancestor process. "Double fork" allows a process not to take care of the funerals of all its child processes, but to delegate this responsibility to the `init` process. Here is the principle: a process P_0 creates a process P_1, which in turn creates a third process P_2. Then P_1 terminates. So P_2 is orphan and will be adopted by `init`, which waits for its termination. The initial process P_0 can

execute a `wait` for P_1 which will be of short duration. The idea is to delegate to the grandchild the work which otherwise would have been for the child.

The schema is the following:

```
# match Unix.fork() with                        (* P0 creates P1 *)
    0 → if Unix.fork() = 0 then exit 0 ;       (* P1 creates P2 and terminates *)
          Printf.printf "P2 did its work\n" ;
          exit 0
  | pid → ignore (Unix.waitpid [] pid) ;       (* P0 waits for P1 to terminate *)
            Printf.printf "P0 can do other things without waiting\n" ;;
P2 did its work
P0 can do other things without waiting
- : unit = ()
```

We will apply this principle to handle requests sent to a server in chapter 20.

Communication Between Processes

The use of processes in application development allows you to delegate work. Nevertheless, these jobs may not be independent and it may be necessary for the processes to communicate with each other.

We introduce two methods of communication between processes: communication pipes and signals. This chapter does not discuss all possibilities of process communication. It is only a first approach to the applications developed in chapters 19 and 20.

Communication Pipes

It is possible for processes to communicate directly between each other in a file oriented style.

Pipes are something like virtual files from which it is possible to read and to write with the input / output functions `read` and `write`. They are of limited size, the exact limit depending from the system. They behave like queues: the first input is also the first output. Whenever data is read from a pipe, it is also removed from it.

This queue behavior is realized by the association of two descriptors with a pipe: one corresponding to the end of the pipe where new entries are written and one for the end where they are read. A pipe is created by the function:

```
# Unix.pipe ;;
- : unit -> Unix.file_descr * Unix.file_descr = <fun>
```

The first component of the resulting pair is the exit of the pipe used for reading. The second is the entry of the pipe used for writing. All processes knowing them can close the descriptors.

Reading from a pipe is blocking, unless all processes knowing its input descriptor (and therefore able to write to it) have closed it; in the latter case, the function `read` returns 0. If a process tries to write to a full pipe, it is suspended until another process has done a read operation. If a process tries to write to a pipe while no other process is available

to read from it (all having closed their output descriptors), the process trying to write receives the signal `sigpipe`, which, if not indicated otherwise, leads to its termination.

The following example shows a use of pipes in which grandchildren tell their process number to their grandparents.

```
let output, input = Unix.pipe();;

let write_pid input =
  try
    let m = "(" ^ (string_of_int (Unix.getpid ())) ^ ")"
    in ignore (Unix.write input m 0 (String.length m)) ;
       Unix.close input
   with
     Unix.Unix_error(n,f,arg) →
       Printf.printf "%s(%s) : %s\n" f arg (Unix.error_message n) ;;

match Unix.fork () with
    0 → for i=0 to 5 do
          match Unix.fork() with
              0 → write_pid input ; exit 0
            | _ → ()
        done ;
        Unix.close input
  | _  → Unix.close input;
         let s = ref ""  and  buff = String.create 5
         in while true do
              match Unix.read output buff 0 5 with
                  0 → Printf.printf "My grandchildren are %s\n" !s ; exit 0
                | n → s := !s ^ (String.sub buff 0 n) ^ "."
           done ;;
```

We obtain the trace:

```
My grandchildren are (1067.3).(1067.4).(1067.8).(1067.7).(1067.6).(1067.5).
```

We have introduced points between each part of the sequence read. This way it is possible to read from the trace the succession of contents of the pipe. Note how the reading is desynchronized: whenever an entry is made, even a partial one, it is consumed.

Named pipes. Some Unix systems support named pipes, which look as if they were normal files. It is possible then to communicate between two processes without a descendence relation using the name of the pipe. The following function allows you to create such a pipe.

```
# Unix.mkfifo ;;
- : string -> Unix.file_perm -> unit = <fun>
```

The file descriptors necessary to use the pipe are obtained by `openfile`, as for usual files, but their behavior is that of pipes. In particular, the command `lseek` can not be used, since we have waiting lines.

Warning `mkfifo` is not implemented for Windows.

Communication Channels

The `Unix` module provides a high level function allowing you to start a program associating with it input or output channels of the calling program:

```
# Unix.open_process ;;
- : string -> in_channel * out_channel = <fun>
```

The argument is the name of the program, or more precisely the calling path of the program, as we would write it to a command line interpreter. The string may contain arguments for the program to execute. The two output values are file descriptors associated with the standard input / output of the started program. It will be executed in parallel with the calling program.

Warning The program started by `open_process` is executed via a call to the Unix command line interpreter `/bin/sh`. The use of that function is therefore only possible for systems that have this interpreter.

We can end the execution of a program started by `open_process` by using:

```
# Unix.close_process ;;
- : in_channel * out_channel -> Unix.process_status = <fun>
```

The argument is the pair of channels associated with a process we want to close. The return value is the execution status of the process whose termination we wait.

There are variants of that functions, opening and closing only one input or output channel:

```
# Unix.open_process_in ;;
- : string -> in_channel = <fun>
# Unix.close_process_in ;;
- : in_channel -> Unix.process_status = <fun>
# Unix.open_process_out ;;
- : string -> out_channel = <fun>
# Unix.close_process_out ;;
- : out_channel -> Unix.process_status = <fun>
```

Here is a nice small example for the use of `open_process`: we start `ocaml` from `ocaml`!

```
# let n_print_string s = print_string s ; print_string "(* <-- *)" ;;
val n_print_string : string -> unit = <fun>
# let p () =
    let oc_in, oc_out = Unix.open_process "/usr/local/bin/ocaml"
    in n_print_string (input_line oc_in) ; print_newline() ;
       n_print_string (input_line oc_in) ; print_newline() ;
       print_char (input_char oc_in) ;
       print_char (input_char oc_in) ;
       flush stdout ;
       let s = input_line stdin
       in output_string oc_out s ;
```

```
          output_string oc_out "#quit\n" ;
          flush oc_out ;
          let r = String.create 250 in
          let n = input oc_in r 0 250
          in n_print_string (String.sub r 0 n) ;
             print_string "Thank you for your visit\n" ;
             flush stdout ;
             Unix.close_process (oc_in, oc_out) ;;
val p : unit -> Unix.process_status = <fun>
```

The call of the function `p` starts a *toplevel* of Objective Caml. We note that it is version 2.03 which is in directory `/usr/local/bin`. The first four read operations allow us to get the header, which is shown by *toplevel*. The line `let x = 1.2 +. 5.6;;` is read from the keybard, then sent to `oc_out` (the output channel bound to the standard input of the new process). This one evaluates the passed Objective Caml expression and writes the result to the standard output which is bound to the input channel `oc_in`. This result is read and written to the output by the function `input`. Also the string `"Thank you for your visit"` is written to the output. We send the command #*quit*;; to exit the new process.

```
# p();;
        Objective Caml version 2.03

# let x = 1.2 +. 5.6;;
val x : float = 6.8
Thank you for your visit
- : Unix.process_status = Unix.WSIGNALED 13
#
```

Signals under Unix

One possibility to communicate with a process is to send it a *signal*. A signal may be received at any moment during the execution of a program. Reception of a signal causes a logical interruption. The execution of a program is interrupted to treat the received signal. Then the execution continues at the point of interruption. The number of signals is quite restricted (32 under Linux). The information carried by a signal is quite rudimentary: it is only the identity (the number) of the signal. The processes have a predefined reaction to each signal. However, the reactions can be redefined for most of the signals.

The data and functions to handle signals are distributed between the modules `Sys` and `Unix`. The module `Sys` contains signals conforming to the POSIX norm (described in [Ste92]) as well as some functions to handle signals. The module `Unix` defines the function `kill` to send a signal. The use of signals under Windows is restricted to `sigint`.

A signal may have several sources: the keyboard, an illegal attempt to access memory, etc. A process may send a signal to another by calling the function

```
# Unix.kill ;;
```

```
- : int -> int -> unit = <fun>
```

Its first parameter is the PID of the receiver. The second is the signal which we want to send.

Handling Signals

There are three categories of reactions associated with a signal. For each category there is a constructor of type `signal_behavior`:

- `Signal_default`: the default behavior defined by the system. In most of the cases this is the termination of the process, with or without the creation of a file describing the process state (`core` file).
- `Signal_ignore`: the signal is ignored.
- `Signal_handle`: the behavior is redefined by an Objective Caml function of type *int -> unit* which is passed as an argument to the constructor. For the modified handling of the signal, the number of the signal is passed to the handling function.

On reception of a signal, the execution of the receiving process is diverted to the function handling the signal. The function allowing you to redefine the behavior associated with a signal is provided by the module `Sys`:

```
# Sys.set_signal;;
- : int -> Sys.signal_behavior -> unit = <fun>
```

The first argument is the signal to redefine. The second is the associated behavior.

The module `Sys` provides another modification function to handle signals:

```
# Sys.signal ;;
- : int -> Sys.signal_behavior -> Sys.signal_behavior = <fun>
```

It behaves like `set_signal`, except that it returns in addition the value associated with the signal before the modification. So we can write a function returning the behavioral value associated with a signal. This can be done even without changing this value:

```
# let signal_behavior s =
   let b = Sys.signal s Sys.Signal_default
   in Sys.set_signal s b ; b ;;
val signal_behavior : int -> Sys.signal_behavior = <fun>
# signal_behavior Sys.sigint;;
- : Sys.signal_behavior = Sys.Signal_handle <fun>
```

However, the behavior associated with some signals can not be changed. Therefore our function can not be used for all signals:

```
# signal_behavior Sys.sigkill ;;
Uncaught exception: Sys_error("Invalid argument")
```

Some Signals

We illustrate the use of some essential signals.

sigint. This signal is generally associated with the key combination CTRL-C. In the following small example we modify the reaction to this signal so that the receiving process is not interrupted until the third occurence of the signal.

We create the following file `ctrlc.ml`:

```
let sigint_handle =
 let n = ref 0
 in function _ → incr n ;
                  match !n with
                      1 → print_string "You just pushed CTRL-C\n"
                    | 2 → print_string "You pushed CTRL-C a second time\n"
                    | 3 → print_string "If you insist ...\n" ; exit 1
                    | _ → () ;;
Sys.set_signal Sys.sigint (Sys.Signal_handle sigint_handle) ;;
match Unix.fork () with
    0 → while true do () done
  | pid → Unix.sleep 1 ; Unix.kill pid Sys.sigint  ;
          Unix.sleep 1 ; Unix.kill pid Sys.sigint  ;
          Unix.sleep 1 ; Unix.kill pid Sys.sigint  ;;
```

This program simulates the push of the key combination CTRL-C by sending the signal `sigint`. We obtain the following execution trace:

```
$ ocamlc -i -o ctrlc ctrlc.ml
val sigint_handle : int -> unit
$ ctrlc
You just pushed CTRL-C
You pushed CTRL-C a second time
If you insist ...
```

sigalrm. Another frequently used signal is `sigalrm`, which is associated with the system clock. It can be sent by the function

```
# Unix.alarm ;;
- : int -> int = <fun>
```

The argument specifies the number of seconds to wait before the sending of the signal `sigalrm`. The return value indicates the number of remaining seconds before the sending of a second signal, or if there is no alarm set.

We use this function and the associated signal to define the function `timeout`, which starts the execution of another function and interrupts it if neccessary, when the indicated time is elapsed. More precisely, the function `timeout` takes as arguments a function `f`, the argument `arg` expected by `f`, the duration (`time`) of the "*timeout*" and the value (`default_value`) to be returned when the duration time has elapsed.

A `timeout` is handled as follows:

1. We modify the behavior associated with the signal `sigalrm` so that a `Timeout` exception is thrown.

2. We take care to remember the behavior associated originally with `sigalrm`, so that it can be restored.
3. We start the clock.
4. We distinguish two cases:
   (a) If everything goes well, we restore the original state of `sigalrm` and return the value of the calculation.
   (b) If not, we restore `sigalrm`, and if the duration has elapsed, we return the default value.

Here are the corresponding definitions and a small example:

```
# exception Timeout ;;
exception Timeout
# let sigalrm_handler = Sys.Signal_handle (fun _ → raise Timeout) ;;
val sigalrm_handler : Sys.signal_behavior = Sys.Signal_handle <fun>
# let timeout f arg time default_value =
    let old_behavior = Sys.signal Sys.sigalrm sigalrm_handler in
    let reset_sigalrm () = Sys.set_signal Sys.sigalrm old_behavior
    in ignore (Unix.alarm time) ;
       try  let res = f arg in reset_sigalrm () ; res
       with exc → reset_sigalrm () ;
                   if exc=Timeout then default_value else raise exc ;;
val timeout : ('a -> 'b) -> 'a -> int -> 'b -> 'b = <fun>
# let iterate n = for i = 1 to n do () done ; n ;;
val iterate : int -> int = <fun>

Printf.printf "1st execution : %d\n" (timeout iterate 10 1 (-1));
Printf.printf "2nd execution : %d\n" (timeout iterate 100000000 1 (-1)) ;;
```

```
1st execution : 10
2nd execution : -1
- : unit = ()
```

sigusr1 and sigusr2. These two signals are provided only for the programer. They are not used by the operating system.

In this example, reception of the signal `sigusr1` by the child triggers the output of the content of variable `i`.

```
let i = ref 0  ;;
let write_i s = Printf.printf "signal received (%d) -- i=%d\n" s !i ;
                flush stdout ;;
Sys.set_signal Sys.sigusr1 (Sys.Signal_handle write_i) ;;

match Unix.fork () with
    0 → while true do incr i done
  | pid → Unix.sleep 0  ; Unix.kill pid Sys.sigusr1  ;
          Unix.sleep 3 ; Unix.kill pid Sys.sigusr1  ;
          Unix.sleep 1  ; Unix.kill pid Sys.sigkill
```

Here is the trace of a program execution:

```
signal received (10) -- i=0
signal received (10) -- i=167722808
```

When we examine the trace, we can see that after having executed the code associated with signal `sigusr1` the first time, the child process continues to execute the loop and to increment `i`.

sigchld. This signal is sent to a parent on termination of a process. We will use it to make a parent more attentive to the evolution of its children. Here's how:

1. We define a function handling the signal `sigchld`. It handles all terminated children on reception of this signal[5] and terminates the parent when he does not have any more children (exception `Unix_error`). In order not to block the parent if not all his children are dead, we use `waitpid` instead of `wait`.
2. The main program, after having redefined the reaction associated with `sigchld`, loops to create five children. After this, the parent does something else (loop **while** `true`) until his children have terminated.

```
let rec sigchld_handle s =
  try  let pid, _ = Unix.waitpid [Unix.WNOHANG] 0
       in if pid <> 0
          then ( Printf.printf "%d is dead and buried at signal %d\n" pid s ;
                 flush stdout ;
                 sigchld_handle s )
  with Unix.Unix_error(_, "waitpid", _) → exit 0 ;;

let i = ref 0
in Sys.set_signal Sys.sigchld (Sys.Signal_handle sigchld_handle) ;
   while true do
     match Unix.fork() with
         0 → let pid = Unix.getpid ()
              in Printf.printf "Creation of %d\n" pid ; flush stdout ;
                 Unix.sleep (Random.int (5+ !i)) ;
                 Printf.printf "Termination of %d\n" pid ; flush stdout ;
                 exit 0
       | _ → incr i ; if !i = 5 then while true do () done
   done ;;
```

We obtain the trace:

```
Creation of 10658
Creation of 10659
Creation of 10662
Creation of 10661
```

5. We recall that the signals are handled in an asynchronous way. So, if two children die one after the other, it is possible that the signal of the first has not been handled.

```
Creation of 10660
Termination of 10662
10662 is dead and buried at signal 17
Termination of 10658
10658 is dead and buried at signal 17
Termination of 10660
Termination of 10659
10660 is dead and buried at signal 17
10659 is dead and buried at signal 17
Termination of 10661
10661 is dead and buried at signal 17
```

Exercises

The three proposed exercises manipulate file descriptors, processes, respectively pipes and signals. The first two exercises stem from Unix system programming. The Objective Caml code can be compared with the C code in the Unix or Linux distributions.

Counting Words: the `wc` Command

We want to (re)program the Unix `wc` command, which counts the number of lines, words or characters contained in a text file. Words are separated by a space character, a tab, or a carriage return. We do not count the separators.

1. Write a first version (`wc1`) of the command, which only handles a single file. The name of the file is passed as an argument on the command line.
2. Write a more elaborated version (`wc2`), which can handle the three options `-l`, `-c`, `-w` as well as several file names. The options indicate if we want to count the number of lines, characters or words. The output of each result shall be preceded by the name of the file.

Pipes for Spell Checking

This exercise uses pipes to concatenate a suite of actions. Each action takes the result of the preceding action as argument. The communication is realized by pipes, connecting the output of one process to the input of the following, in the style of the Unix command line symbol `|` .

1. Write a function `pipe_two_progs` of type *`string * string list -> string * string list -> unit`* such that `pipe_two_progs (p1,[a1; ...; an]) (p2,[b1; ...; bp])` starts the programs `p1 a1 ... an` and `p2 b1 ... bp`, redirecting the standard output of `p1` to the standard input of `p2`. *`ai`* and *`bi`* are the command line arguments of each program.
2. We revisit the spell checker function from the exercise on page 115 to write a first program. Modify it so that the list of faulty words is sent without treatment in the form of one line per word to the standard output.

3. The second program takes a sequence of character strings from its standard input and sorts it in lexicographical order. The function `Sort.list` can be used, which sorts a list in an order defined by a given predicate. The sorted list is written to the standard output.
4. Test the function `pipe_two_progs` with the two programs.
5. Write a function `pipe_n_progs` to connect a list of programs.
6. Write a program to suppress multiple occurrences of elements in a list.
7. Test the function `pipe_n_progs` with these three programs.

Interactive Trace

In a complex calculation it may be useful to interact with the program to verify the progression. For this purpose we revisit the exercise on page 244 on the computation of prime numbers contained in an interval.

1. Modify the program so that a global variable `result` always contains the last prime number found.
2. Write a function `sigint_handle` which handles the signal `sigint` and writes the content of `result` to the output.
3. Modify the default signal handling of `sigint` by associating with it the preceding function `sigint_handle`.
4. Compile the program, then start the executable with an upper bound for the computation time. During the computation, send the signal `sigint` to the process, by the Unix `kill` command as well as by the key combination `CTRL-C`.

Summary

This chapter presented the main system interface functions provided by the `Unix` module. Despite of its name, the module offers a large number of functions which can be used under Windows as well (see figure 18.1).
In the area of process creation, we did concentrate on the possibilities of communication between several Objective Caml programs running at the same time on the same machine. Operations handling lower level file access, signals and communication pipes have been discussed in detail.

To Learn More

The `Unix` module provides functions of the Unix system library. Most of the underlying programming paradigms are not described in Objective Caml. The reader may refer to standard books about system programming. We cite [Ste92], or [CDM98], more specific to Linux.

Further, the excellent lecture notes from Xavier Leroy [Ler92] have for subject system programming in Caml-Light. They can be accessed under the following address:

Link: http://pauillac.inria.fr/~xleroy/publi/unix-in-caml.ps.gz

The implementation of the `Unix` module is a good example for the cooperation between C and Objective Caml. A large number of functions are just calls to C system functions, with the additional type transcription of the data. The implementation sources are good examples of how to interface an Objective Caml program with a C library. The programs can be found in the directories `otherlibs/unix` and `otherlibs/win32unix` of the Objective Caml distribution.

The chapter did present several functionalities of the `Unix` module. Some more points will be approached in chapter 20 about communication sockets and Internet addresses. Other notions, like that of terminals, of file systems, etc. are not discussed in this book. They can be explored in one of the books mentioned above.