19

Concurrent Programming

Concurrency is the word used to describe causal independence between a number of actions, such as the execution of a number of instructions "at the same time". This is also the definition which we give of the term "parallel" in the introduction of this fourth part. The processes of the `Unix` library presented in the preceding chapter could be considered as concurrent to the extent that the `Unix` system provides the appearance of their simultaneous execution on a uniprocessor machine. But the notion of process and concurrency does not apply only to those obtained by the `fork` system call.

The Objective Caml language possesses a library for lightweight processes (threads.) The principle difference between a thread and a process is in the sharing or non-sharing of memory between the different child processes of the same program. Only the context of execution differs between two threads: the code and memory sections of the data are shared. Threads do not improve the execution time of an application. Their principal attraction is to make it possible to express the programming of concurrent algorithms within a language.

The nature of the chosen language, imperative or functional, affects the model of concurrency. For an imperative program, as every thread can modify the communal/shared memory, we are in a shared memory model. Communication between processes can be achieved by values written and read in this memory. For a purely functional program, that is to say, without side effects, even though the memory is shared, the calculations which each process executes do not act on this shared memory. In this case, the model used is that of separate memory and interaction between processes must be achieved by communication of values though channels.

The Objective Caml language implements both models in its thread library. The `Thread` module makes it possible to start new processes corresponding to a function call with its argument. Modules `Mutex` and `Condition` provide the synchronization tools for mutual exclusion and waiting on a condition. The `Event` model implements a means of communication of language values by events. These values can themselves be

functional, thus making it possible to exchange calculations to be carried out between threads. As always in Objective Caml it is possible to mix the two models.

This library is portable to the different systems where OCAML runs. Unlike the `Unix` module, the `Thread` library facilitates the use of processes on machines that are not running Unix.

Plan of the Chapter

The first section details the possible interactions between threads, and proceeds with describing module `Thread`, and showing how to execute many processes in the same application.

The second part deals with the synchronization between threads by mutual exclusion (`Mutex` module), and with waiting for conditions (`Condition` module). Two complete examples show the difficulties inherent to this module.

The third section explains the mode of communication by events provided by the `Event` module and the new possibilities which is provides.

The fourth section concludes the chapter with the implementation of a shared queue for the different counters at a post office.

Concurrent Processes

With an application composed of many concurrent processes, we lose the convenience offered by the determinism of sequential programs. For processes sharing the same zone of memory, the result of the following program cannot be deduced from reading it.

<table>
<tr><td colspan="2">main program</td></tr>
<tr><td colspan="2"><code>let x = ref 1;;</code></td></tr>
<tr><td>process P</td><td>process Q</td></tr>
<tr><td><code>x := !x + 1;;</code></td><td><code>x := !x * 2;;</code></td></tr>
</table>

At the end of the execution of P and Q, the reference `x` can point to 2, 3 or 4, depending on the order of execution of each process.

This indeterminism applies also to terminations. When the memory state depends on the execution of each parallel process, an application can fail to terminate on a particular execution, and terminate on another. To provide some control over the execution, the processes must be synchronized.

For processes using distinct memory areas, but communicating between each other, their interaction depends on the type of communication. We introduce for the following example two communication primitives: `send` which sends a value, showing the

destination, and `receive` which receives a value from a process. Let P and Q be two communicating processes:

process P	process Q
`let x = ref 1;;`	`let y = ref 1;;`
`send(Q,!x);`	`y := !y + 3;`
`x := !x * 2;`	`y := !y + receive(P);`
`send(Q,!x);`	`send(P,!y);`
`x := !x + receive(Q);`	`y := !y + receive(P);`

In the case of a transient communication, process Q can miss the messages of P. We fall back into the non-determinism of the preceding model.

For an asynchronous communication, the medium of the communication channel stores the different values that have been transmitted. Only reception is blocking. Process P can be waiting for Q, even if the latter has not yet read the two messages from P. However, this does not prevent it from transmitting.

We can classify concurrent applications into five categories according to the program units that compose them:

1. unrelated;
2. related, but without synchronization;
3. related, with mutual exclusion;
4. related, with mutual exclusion and communication;
5. related, without mutual exclusion, and with synchronous communication.

The difficulty of implementation comes principally from these last categories. Now we will see how to resolve these difficulties by using the Objective Caml libraries.

Compilation with Threads

The Objective Caml thread library is divided into five modules, of which the first four each define an abstract type:

- module `Thread`: creation and execution of threads. (type *`Thread.t`*);
- module `Mutex`: creation, locking and release of mutexes. (type *`Mutex.t`*);
- module `Condition`: creation of conditions (signals), waiting and waking up on a condition (type *`Condition.t`*);
- module `Event`: creation of communication channels (type *`'a Event.channel`*), the values which they carry (type *`'a Event.event`*), and communication functions.
- module `ThreadUnix`: redefinitions of I/O functions of module `Unix` so that they are not blocking.

This library is not part of the execution library of Objective Caml. Its use requires the option `-custom` both for compiling programs and for constructing a new toplevel by using the commands:

```
$ ocamlc -thread -custom threads.cma  files.ml -cclib -lthreads
$ ocamlmktop -tread -custom -o threadtop thread.cma -cclib -lthreads
```

The `Threads` library is not usable with the native compiler unless the platform implements threads conforming to the POSIX 1003[1]. Thus we compile executables by adding the libraries `unix.a` and `pthread.a`:

```
$ ocamlc -thread -custom threads.cma files.ml -cclib -lthreads \
  -cclib -lunix -cclib -lpthread
$ ocamltop -thread -custom threads.cma files.ml -cclib -lthreads \
  -cclib -lunix -cclib -lpthread
$ ocamlcopt -thread threads.cmxa files.ml -cclib -lthreads \
  -cclib -lunix -cclib -lpthread
```

Module Thread

The Objective Caml `Thread` module contains the primitives for creation and management of threads. We will not make an exhaustive presentation, for instance the operations of file I/O have been described in the preceding chapter.

A thread is created through a call to:

```
# Thread.create ;;
- : ('a -> 'b) -> 'a -> Thread.t = <fun>
```

The first argument, of type *'a -> 'b*, corresponds to the function executed by the created process; the second argument, of type *'a*, is the argument required by the executed function; the result of the call is the descriptor associated with the process. The process thus created is automatically destroyed when the associated function terminates.

Knowing its descriptor, we can ask for the execution of a process and wait for it to finish by using the function `join`. Here is a usage example:

```
# let f_proc1 () = for i=0 to 10 do Printf.printf "(%d)" i; flush stdout done;
                   print_newline() ;;
val f_proc1 : unit -> unit = <fun>
# let t1 = Thread.create f_proc1 () ;;
val t1 : Thread.t = <abstr>
# Thread.join t1 ;;
(0)(1)(2)(3)(4)(5)(6)(7)(8)(9)(10)
- : unit = <unknown constructor>
```

1. In this case, the Objective Caml compilers should have been constructed to indicate that they used the library furnished by the platform, and not the one provided by the distribution.

Warning The result of the execution of a process is not recovered by the parent process, but lost when the child process terminates.

We can also brutally interrupt the execution of a process of which we know the descriptor with the function `kill`. For instance, we create a process which is immediately interrupted:

```
# let n = ref 0 ;;
val n : int ref = {contents=0}
# let f_proc1 () = while true do incr n done ;;
val f_proc1 : unit -> unit = <fun>
# let go () = n := 0 ;
              let t1 = Thread.create f_proc1 ()
              in Thread.kill t1 ;
                 Printf.printf "n = %d\n" !n ;;
val go : unit -> unit = <fun>
# go () ;;
n = 0
- : unit = ()
```

A process can put an end to its own activity by the function:

```
# Thread.exit ;;
- : unit -> unit = <fun>
```

It can suspend its activity for a given time by a call to:

```
# Thread.delay ;;
- : float -> unit = <fun>
```

The argument stands for the number of seconds to wait.

Let us consider the previous example, and add timing. We create a first process `t1` of which the associated function `f_proc2` creates in its turn a process `t2` which executes `f_proc1`, then `f_proc2` delays for `d` seconds, and then terminates `t2`. On termination of `t1`, we print the contents of `n`.

```
# let f_proc2 d =
     n := 0 ;
     let t2 = Thread.create f_proc1 ()
     in Thread.delay d ;
        Thread.kill t2 ;;
val f_proc2 : float -> unit = <fun>
# let t1 = Thread.create f_proc2 0.25
  in Thread.join t1 ; Printf.printf "n = %d\n" !n ;;
n = 132862
- : unit = ()
```

Synchronization of Processes

In the setting of processes sharing a common zone of memory, the word "concurrency" carries its full meaning: the various processes involved are compete for access to the unique resource of the memory[2]. To the problem of division of resources, is added that of the lack of control of the alternation and of the execution times of the concurrent processes.

The system which manages the collection of processes can at any moment interrupt a calculation in progress. Thus when two processes cooperate, they must be able to guarantee the integrity of the manipulations of certain shared data. For this, a process should be able to remain owner of these data as long as it has not completed a calculation or any other operation (for example, an acquisition of data from a peripheral). To guarantee the exclusivity of access to the data to a single process, we set up a mechanism called *mutual exclusion*.

Critical Section and Mutual Exclusion

The mechanisms of mutual exclusion are implemented with the help of particular data structures called *mutexes*. The operations on mutexes are limited to their creation, their setting, and their disposal. A mutex is the smallest item of data shared by a collection of concurrent processes. Its manipulation is always exclusive. To the notion of exclusivity of manipulation of a mutex is added that of exclusivity of *possession*: only the process which has taken a mutex can free it; if other processes wish to use the mutex, then they must wait for it to be released by the process that is holding it.

`Mutex` *Module*

Module `Mutex` is used to create mutexes between processes related by mutual exclusion on an area of memory. We will illustrate their use with two small classic examples of concurrency.

The functions of creation, locking, and unlocking of mutexes are:

```
# Mutex.create ;;
- : unit -> Mutex.t = <fun>
# Mutex.lock ;;
- : Mutex.t -> unit = <fun>
# Mutex.unlock ;;
- : Mutex.t -> unit = <fun>
```

There exists a variant of mutex locking that is non-blocking:

```
# Mutex.try_lock;;
- : Mutex.t -> bool = <fun>
```

2. In a more general sense, we can be in contention for other resources such as I/O peripherals

If the mutex is already locked, the function returns **false**. Otherwise, the function locks the mutex and returns **true**.

The Dining Philosophers

This little story, due to Dijkstra, illustrates a pure problem of resource allocation. It goes as follows:

"Five oriental philosophers divide their time between study and coming to the refectory to eat a bowl of rice. The room devoted to feeding the philosophers contains nothing but a single round table on which there is a large dish of rice (always full), five bowls, and five chopsticks."

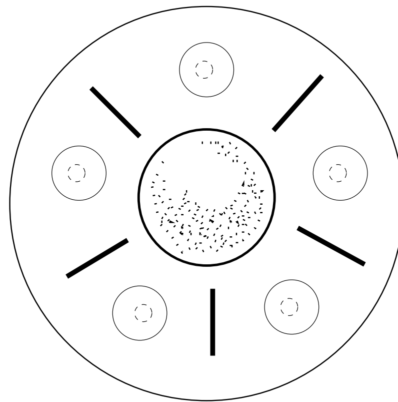

Figure 19.1: The Table of the Dining Philosophers

As we can see in the figure 19.1, a philosopher who takes his two chopsticks beside his bowl stops his neighbours from doing the same. When he puts down one of his chopsticks, his neighbour, famished, can grab it. If needs be, this latter should wait until the other chopstick is available. Here the chopsticks are the resources to be allocated.

To simplify things, we suppose that each philosopher habitually comes to the same place at the table. We model the five chopsticks as five mutexes stored in a vector `b`.

```
# let b =
    let b0 = Array.create 5 (Mutex.create()) in
     for i=1 to 4 do b0.(i) <- Mutex.create() done;
     b0 ;;
val b : Mutex.t array = [|<abstr>; <abstr>; <abstr>; <abstr>; <abstr>|]
```

Eating and meditation are simulated by a suspension of processes.

```
# let meditation = Thread.delay
  and eating = Thread.delay ;;
val meditation : float -> unit = <fun>
val eating : float -> unit = <fun>
```

We model a philosopher by a function which executes an infinite sequence of actions from Dijsktra's story. Taking a chopstick is simulated by the acquisition of a mutex, thus a single philosopher can hold a given chopstick at a time. We introduce a little time of reflection between taking and dropping of each of the two chopsticks while a number of output commands track the activity of the philosopher.

```
# let philosopher i =
    let ii = (i+1) mod 5
    in while true do
         meditation 3. ;
         Mutex.lock b.(i);
         Printf.printf "Philosopher (%d) takes his left-hand chopstick" i ;
         Printf.printf " and meditates a little while more\n";
         meditation 0.2;
         Mutex.lock b.(ii);
         Printf.printf "Philosopher (%d) takes his right-hand chopstick\n" i;
         eating 0.5;
         Mutex.unlock b.(i);
         Printf.printf "Philosopher (%d) puts down his left-hand chopstick" i;
         Printf.printf " and goes back to meditating\n";
         meditation 0.15;
         Mutex.unlock b.(ii);
         Printf.printf "Philosopher (%d) puts down his right-hand chopstick\n" i
       done ;;
val philosopher : int -> unit = <fun>
```

We can test this little program by executing:

```
for i=0 to 4 do ignore (Thread.create philosopher i) done ;
while true do Thread.delay 5. done ;;
```

We suspend, in the infinite loop **while**, the main process in order to increase the chances of the philosopher processes to run. We use randomly chosen delays in the activity loop with the aim of creating some disparity in the parallel execution of the processes.

Problems of the naïve solution. A terrible thing can happen to our philosophers: they all arrive at the same time and seize the chopstick on their left. In this case we are in a situation of *dead-lock*. None of the philosophers can eat! We are in a situation of *starvation*.

To avoid this, the philosophers can put down a chopstick if they do not manage to take the second one. This is highly courteous, but still allows two philosophers to gang up against a third to stop him from eating, by not letting go of their chopsticks, except the ones that their other neighbour has given them. There exist numerous solutions to this problem. One of them is the object of the exercise on page 619.

Producers and Consumers I

The pair of *producers-consumers* is a classic example of concurrent programming. A group of processes, designated the producers, are in charge of storing data in a queue: a second group, the consumers, is in charge of removing it. Each intervening party excludes the others.

We implement this scheme using a queue shared between the producers and the consumers. To guarantee the proper operation of the system, the queue is manipulated in mutual exclusion in order to guarantee the integrity of the operations of addition and removal.

`f` is the shared queue, and `m` is the mutex.

```
# let f = Queue.create ()  and  m = Mutex.create () ;;
val f : '_a Queue.t = <abstr>
val m : Mutex.t = <abstr>
```

We divide the activity of a producer into two parts: creating a product (function `produce`) and storing a product (fonction `store`). Only the operation of storage needs the mutex.

```
# let produce i p d =
     incr p ;
     Thread.delay d ;
     Printf.printf "Producer (%d) has produced %d\n" i !p ;
     flush stdout ;;
val produce : int -> int ref -> float -> unit = <fun>

# let store i p =
     Mutex.lock m ;
     Queue.add (i,!p) f ;
     Printf.printf "Producer (%d) has added its %dth product\n" i !p ;
     flush stdout ;
     Mutex.unlock m ;;
val store : int -> int ref -> unit = <fun>
```

The code of the producer is an endless loop of creation and storage. We introduce a random delay at the end of each iteration in order to desynchronize the execution.

```
# let producer i =
     let p = ref 0 and d = Random.float 2.
     in while true do
          produce i p d ;
          store i p ;
          Thread.delay (Random.float 2.5)
        done ;;
val producer : int -> unit = <fun>
```

The only operation of the consumer is the retrieval of an element of the queue, taking care that the product is actually there.

```
# let consumer i =
```

```
    while true do
      Mutex.lock m ;
      ( try
          let ip, p = Queue.take f
          in Printf.printf "The consumer(%d) " i ;
             Printf.printf "has taken product (%d,%d)\n" ip p ;
             flush stdout ;
        with
           Queue.Empty →
             Printf.printf "The consumer(%d) " i ;
             print_string "has returned empty-handed\n" ) ;
      Mutex.unlock m ;
      Thread.delay (Random.float 2.5)
    done ;;
val consumer : int -> unit = <fun>
```

The following test program creates four producers and four consumers.

```
for i = 0 to 3 do
 ignore (Thread.create producer i);
 ignore (Thread.create consumer i)
done ;
while true do Thread.delay 5. done ;;
```

Waiting and Synchronization

The relation of mutual exclusion is not "fine" enough to describe synchronization between processes. It is not rare that the work of a process depends on the completion of an action by another process, thus modifying a certain condition. It is therefore desirable that the processes should be able to communicate the fact that this condition might have changed, indicating to the waiting processes to test it again. The different processes are thus in a relation of mutual exclusion with communication.

In the preceding example, a consumer, rather than returning empty-handed, could wait until a producer came to resupply the stock. This last could signal to the waiting consumer that there is something to take. The model of waiting on a condition to take a mutex is known as *semaphore*.

Semaphores. A semaphore is an integral variable s which can only take non negative values. Once s is initialised, the only operations allowed are: $wait(s)$ and $signal(s)$, written $P(s)$ and $V(s)$, respectively. They are defined thus, s corresponding to the number of resources of a given type.

- $wait(s)$: if $s > 0$ then $s := s - 1$, otherwise the process, having called $wait(s)$, is suspended.
- $signal(s)$: if a process has been suspended after a prior invocation of $wait(s)$, then wake it up, otherwise $s := s + 1$.

A semaphore which only takes the values 0 or 1 is called a *binary semaphore*.

`Condition` *Module*

The functions of the module `Condition` implement the primitives of putting to sleep and waking up processes on a signal. A signal, in this case, is a variable shared by a collection of processes. Its type is abstract and the manipulation functions are:

`create` : *`unit -> Condition.t`* which creates a new signal.

`signal` : *`Condition.t -> unit`* which wakes up one of the processes waiting on a signal.

`broadcast` : *`Condition.t -> unit`* which wakes up all of the processes waiting on a signal.

`wait` : *`Condition.t -> Mutex.t -> unit`* which suspends the calling process on the signal passed as the first argument. The second argument is a mutex used to protect the manipulation of the signal. It is released, and then reset at each execution of the function.

Producers and Consumers (2)

We revisit the example of producers and consumeres by using the mechanism of condition variables to put to sleep a consumer arriving when the storehouse is empty.

To implement synchronization between waiting consumers and production, we declare:

```
# let c = Condition.create () ;;
val c : Condition.t = <abstr>
```

We modify the storage function of the producer by adding to it the sending of a signal:

```
# let store2 i p =
    Mutex.lock m ;
    Queue.add (i,!p) f ;
    Printf.printf "Producer (%d) has added its %dth product\n" i !p ;
    flush stdout ;
    Condition.signal c ;
    Mutex.unlock m ;;
val store2 : int -> int ref -> unit = <fun>
# let producer2 i =
    let p = ref 0 in
    let d = Random.float 2.
    in while true do
         produce i p d;
         store2 i p;
```

```
      Thread.delay (Random.float 2.5)
    done ;;
val producer2 : int -> unit = <fun>
```

The activity of the consumer takes place in two phases: waiting until a product is available, then taking the product. The mutex is taken when the wait is finished and it is released when the consumer has taken its product. The wait takes place on the variable `c`.

```
# let wait2 i =
    Mutex.lock m ;
    while Queue.length f = 0 do
      Printf.printf "Consumer (%d) is waiting\n" i ;
      Condition.wait c m
    done ;;
val wait2 : int -> unit = <fun>
# let take2 i =
    let ip, p = Queue.take f in
      Printf.printf "Consumer (%d) " i ;
      Printf.printf "takes product (%d, %d)\n" ip p ;
      flush stdout ;
      Mutex.unlock m ;;
val take2 : int -> unit = <fun>
# let consumer2 i =
    while true do
      wait2 i;
      take2 i;
      Thread.delay (Random.float 2.5)
   done ;;
val consumer2 : int -> unit = <fun>
```

We note that it is no longer necessary, once a consumer has begun to wait in the queue, to check for the existence of a product. Since the end of its wait corresponds to the locking of the mutex, it does not run the risk of having the new product stolen before it takes it.

Readers and Writers

Here is another classic example of concurrent processes in which the agents do not have the same behaviour with respect to the shared data.

A writer and some readers operate on some shared data. The action of the first may cause the data to be momentarily inconsistent, while the second group only have a passive action. The difficulty arises from the fact that we do not wish to prohibit multiple readers from examining the data simultaneously. One solution to this problem is to keep a counter of the number of readers in the processes of accessing the data. Writing is not allowed except if the number of readers is 0.

The data is symbolized by the integer `data` which takes the value `0` or `1`. The value `0` indicates that the data is ready for reading:

```
# let data = ref 0 ;;
val data : int ref = {contents=0}
```

Operations on the counter n are protected by the mutex m:

```
# let n = ref 0 ;;
val n : int ref = {contents=0}
# let m = Mutex.create () ;;
val m : Mutex.t = <abstr>
# let cpt_incr () = Mutex.lock m ; incr n ; Mutex.unlock m ;;
val cpt_incr : unit -> unit = <fun>
# let cpt_decr () = Mutex.lock m ; decr n ; Mutex.unlock m ;;
val cpt_decr : unit -> unit = <fun>
# let cpt_signal () = Mutex.lock m ;
                      if !n=0 then Condition.signal c ;
                      Mutex.unlock m ;;
val cpt_signal : unit -> unit = <fun>
```

The readers update the counter and emit the signal c when no more readers are present. This is how they indicate to the writer that it may come into action.

```
# let c = Condition.create () ;;
val c : Condition.t = <abstr>
# let read i =
    cpt_incr () ;
    Printf.printf "Reader (%d) read (data=%d)\n" i !data ;
    Thread.delay (Random.float 1.5) ;
    Printf.printf "Reader (%d) has finished reading\n" i ;
    cpt_decr () ;
    cpt_signal () ;;
val read : int -> unit = <fun>

# let reader i = while true do read i ; Thread.delay (Random.float 1.5) done ;;
val reader : int -> unit = <fun>
```

The writer needs to block the counter to prevent the readers from accessing the shared data. But it can only do so if the counter is 0, otherwise it waits for the signal indicating that this is the case.

```
# let write () =
    Mutex.lock m ;
    while !n<>0 do Condition.wait c m done ;
    print_string  "The writer is writing\n" ; flush stdout ;
    data := 1 ; Thread.delay (Random.float 1.) ; data := 0 ;
    Mutex.unlock m ;;
val write : unit -> unit = <fun>

# let writer () =
    while true do write () ; Thread.delay (Random.float 1.5) done ;;
val writer : unit -> unit = <fun>
```

We create a reader and six writers to test these functions.

```
ignore (Thread.create writer ());
for i=0 to 5 do ignore(Thread.create reader i) done;
while true do Thread.delay 5. done ;;
```

This solution guarantees that the writer and the readers cannot have access to the data at the same time. On the contrary, nothing guarantees that the writer could ever "fufill his officé', there we are confronted again with a case of starvation.

Synchronous Communication

Module `Event` from the thread library implements the communication of assorted values between two processes through particular "communication channels". The effective communication of the value is synchronized through send and receive events.

This model of communication synchronized by events allows the transfer through typed channels of the values of the language, including closures, objects, and events.

It is described in [Rep99].

Synchronization using Communication Events

The primitive communication events are:

- `send c v` sends a value `v` on the channel `c`;
- `receive c` receives a value on the channel `c`

So as to implement the physical action with which they are associated, two events should be synchronized. For this purpose, we introduce an operation of synchronization (`sync`) on events. The sending and receiving of a value are not effective unless the two communicating processes are in phase. If a single process wishes to synchronize itself, the operation gets blocked, waiting for the second process to perform its synchronization. This implies that a sender wishing to synchronize the sending of a value (`sync (send c v)`) can find itself blocked waiting for a synchronization from a receiver (`sync (receive c)`).

Transmitted Values

The communication channels through which the exchanged values travel are typed: Nothing prevents us from creating multiple channels for communicating each type of value. As this communication takes place between Objective Caml threads, any value of the language can be sent on a channel of the same type. This is useful for closures, objects, and also events, for a "relayed" synchronization request.

Module **Event**

The values encapsulated in communication events travel through communication channels of the abstract data type *'a channel*. The creation function for channels is:

```
# Event.new_channel ;;
- : unit -> 'a Event.channel = <fun>
```

Send and receive events are created by a function call:

```
# Event.send ;;
- : 'a Event.channel -> 'a -> unit Event.event = <fun>
# Event.receive ;;
- : 'a Event.channel -> 'a Event.event = <fun>
```

We can consider the functions `send` and `receive` as constructors of the abstract type *'a event*. The event constructed by send does not preserve the information about the type of the value to transmit (type *unit Event.event*). On the other hand, the receive event takes account of it to recover the value during a synchronization. These functions are non-blocking in the sense that the transmission of a value does not take place until the time of the synchronization of two processes by the function:

```
# Event.sync ;;
- : 'a Event.event -> 'a = <fun>
```

This function may be blocking for the sender and the receiver.

There is a non-blocking version:

```
# Event.poll ;;
- : 'a Event.event -> 'a option = <fun>
```

This function verifies that another process is waiting for synchronization.

If this is the case, it performs the transmissions, and returns the value `Some v`, if `v` is the value associated with the event, and `None` otherwise. The received message, extracted by the function `sync`, can be the result of a more or less complicated process, triggering other exchanges of messages.

Example of synchronization. We define three threads. The first, `t1`, sends a chain of characters on channel `c` (function `g`) shared by all the processes. The two others `t2` and `t3` wait for a value on the same channel. Here are the functions executed by the different processes:

```
# let c = Event.new_channel ();;
val c : '_a Event.channel = <abstr>
# let f () =
    let ids = string_of_int (Thread.id (Thread.self ()))
    in print_string ("-------- before  -------" ^ ids) ; print_newline() ;
       let e = Event.receive c
       in print_string ("-------- during  -------" ^ ids) ; print_newline() ;
          let v = Event.sync e
```

```
          in print_string (v ^ " " ^ ids ^ " ") ;
             print_string ("-------- after  -------" ^ ids) ; print_newline() ;;
val f : unit -> unit = <fun>
# let g () =
     let ids = string_of_int (Thread.id (Thread.self ()))
     in print_string ("Start of " ^ ids ^ "\n");
        let e2 = Event.send c "hello"
        in Event.sync e2 ;
           print_string ("End of " ^ ids) ;
           print_newline () ;;
val g : unit -> unit = <fun>
```

The three processes are created and executed:

```
# let t1,t2,t3 = Thread.create f (), Thread.create f (), Thread.create g ();;
val t1 : Thread.t = <abstr>
val t2 : Thread.t = <abstr>
val t3 : Thread.t = <abstr>
# Thread.delay 1.0;;
Start of 5
-------- before  -------6
-------- during  -------6
hello 6 -------- after  -------6
-------- before  -------7
-------- during  -------7
End of 5
- : unit = <unknown constructor>
```

The transmission may block. The trace of `t1` is displayed after the synchronization traces of `t2` and `t3`. Only one of the two processes `t1` or `t2` is really terminated, as the following calls show:

```
# Thread.kill t1;;
- : unit = ()
# Thread.kill t2;;
Uncaught exception: Failure("Thread.kill: killed thread")
```

Example: Post Office

We present, to end this chapter, a slightly more complete example of a concurrent program: modelling a common queue at a number of counters at a post office.

As always in concurrent programming the problems are posed metaphorically, but replace the counters of the post office by a collection of printers and you have the solution to a genuine problem in computing.

Here the policy of service that we propose; it is well tried and tested, rather than original: each client takes a number when he arrives; when a clerk has finished serving

a client, he calls for a number. When his number is called, the client goes to the corresponding counter.

Organization of development. We distinguish in our development *resources*, and *agents*. The former are: the number dispenser, the number announcer, and the windows. The latter are: the clerks and the clients. The resources are modeled by objects which manage their own mutual exclusion mechanisms. The agents are modelled by functions executed by a thread. When an agent wishes to modify or examine the state of an object, it does not itself have to know about or manipulate mutexes, which allows a simplified organization for access to sensitive data, and avoids oversights in the coding of the agents.

The Components

The Dispenser. The number dispenser contains two fields: a counter and a mutex. The only method provided by the distributor is the taking of a new number.

```
# class dispenser () =
    object
      val mutable n = 0
      val m = Mutex.create()
      method take () = let r = Mutex.lock m ; n <- n+1 ; n
                         in Mutex.unlock m ; r
    end ;;
class dispenser :
  unit ->
  object val m : Mutex.t val mutable n : int method take : unit -> int end
```

The mutex prevents two clients from taking a number at the same time. Note the way in which we use an intermediate variable (`r`) to guarantee that the number calculated in the critical section is the same as the one return by the method call.

The Announcer. The announcer contains three fields: an integer (the client number being called); a mutex and a condition variable. The two methods are: (`wait`) which reads the number, and (`call`), which modifies it.

```
# class announcer () =
    object
      val mutable nclient = 0
      val m = Mutex.create()
      val c = Condition.create()

      method wait n =
        Mutex.lock m;
        while n > nclient do Condition.wait c m done;
        Mutex.unlock m;
```

```
  method call () =
    let r = Mutex.lock m ;
            nclient <- nclient+1 ;
            nclient
    in Condition.broadcast c ;
       Mutex.unlock m ;
       r
 end ;;
```

The condition variable is used to put the clients to sleep, waiting for their number. They are all woken up when the method `call` is invoked. Reading or writing access to the called number is protected by the mutex.

The window. The window consists of five fields: a fixed window number (variable `ncounter`); the number of the client being waited for (variable `nclient`); a boolean (variable `available`); a mutex, and a condition variable.

It offers eight methods, of which two are private: two simple access methods (methods `get_ncounter` and `get_nclient`): a group of three methods simulating the waiting period of the clerk between two clients (private method `wait` and public methods `await_arrival`, `await_departure`); a group of three methods simulate the occupation of the window (private method `set_available` and methods `arrive`, `depart`).

```
# class counter (i:int) =
    object(self)
      val ncounter = i
      val mutable nclient = 0
      val mutable available = true
      val m = Mutex.create()
      val c = Condition.create()

      method get_ncounter = ncounter
      method get_nclient  = nclient

      method private wait f =
        Mutex.lock m ;
        while f () do Condition.wait c m done ;
        Mutex.unlock m

      method wait_arrival n = nclient <- n ; self#wait (fun () → available)
      method wait_departure () =  self#wait (fun () → not available)

      method private set_available b =
        Mutex.lock m ;
        available <- b ;
        Condition.signal c ;
        Mutex.unlock m
      method arrive () = self#set_available false
      method leave () = self#set_available true
```

```
    end ;;
```

A post office. We collect these three resources in a record type:

```
# type office = { d : dispenser ; a : announcer ;  cs : counter array } ;;
```

Clients and Clerks

The behaviour of the system as a whole will depend on the three following parameters:

```
# let service_delay = 1.7 ;;
# let arrival_delay = 1.7 ;;
# let counter_delay = 0.5 ;;
```

Each represents the maximum value of the range from which each effective value will be randomly chosen. The first parameter models the time taken to serve a client; the second, the delay between the arrival of clients in the post office; the last, the time it takes a clerk to call a new client after the last one has left.

The Clerk. The work of a clerk consists of looping indefinitely over the following sequence:

1. Call for a number.
2. Wait for the arrival of a client holding the called number.
3. Wait for the departure of the client occupying his counter.

Adding some output, we get the function:

```
# let clerk ((a:announcer), (c:counter)) =
    while true do
      let n = a#call ()
      in Printf.printf "Counter %d calls %d\n" c#get_ncounter n ;
         c#wait_arrival n ;
         c#wait_departure () ;
         Thread.delay (Random.float counter_delay)
    done ;;
val clerk : announcer * counter -> unit = <fun>
```

The Client. A client executes the following sequence:

1. Take a waiting number.

2. Wait until his number is called.
3. Go to the window having called for the number to obtain service.

The only slightly complex activity of the client is to find the counter where they are expected.

We give, for this, the auxiliary function:

```
# let find_counter n cs =
    let i = ref 0 in while cs.(!i)#get_ncounter <> n do incr i done ; !i ;;
val find_counter : 'a -> < get_ncounter : 'a; .. > array -> int = <fun>
```

Adding some output, the principal function of the client is:

```
# let client o =
    let n = o.d#take()
    in Printf.printf "Arrival of client %d\n" n ; flush stdout ;
       o.a#wait n ;
       let ic = find_counter n o.cs
       in o.cs.(ic)#arrive () ;
          Printf.printf "Client %d occupies window %d\n" n ic ;
          flush stdout ;
          Thread.delay (Random.float service_delay) ;
          o.cs.(ic)#leave () ;
          Printf.printf "Client %d leaves\n" n ; flush stdout ;;
val client : office -> unit = <fun>
```

The System

The main programme of the application creates a post office and its clerks (each clerk is a process) then launches a process which creates an infinite stream of clients (each client is also a process).

```
# let main () =
   let o =
    { d = new dispenser();
      a = new announcer();
      cs = (let cs0 = Array.create 5 (new counter 0) in
             for i=0 to 4 do cs0.(i) <- new counter i done;
             cs0)
    }
   in for i=0 to 4 do ignore (Thread.create clerk (o.a, o.cs.(i))) done ;
      let create_clients o = while true do
                                ignore (Thread.create client o) ;
                                Thread.delay (Random.float arrival_delay)
                              done
      in ignore (Thread.create create_clients o) ;
         Thread.sleep () ;;
val main : unit -> unit = <fun>
```

The last instruction puts the process associated with the program to sleep in order to pass control immediately to the other active processes of the application.

Exercises

The Philosophers Disentangled

To solve the possible deadlock of the dining philosophers, it suffices to limit access to the table to four at once. Implement this solution.

More of the Post Office

We suggest the following modification to the post office described on page 614: some impatient clients may leave before there number has been called.

1. Add a method `wait` (with type *int -> unit*) to the class `dispenser` which causes the caller to wait while the last number distributed is less than or equal to the parameter of the method (it is necessary to modify `take` so that it emits a signal).
2. Modify the method `await_arrival` of class `counter`, so that it returns the boolean value `true` if the expected client arrives, and `false` if the client has not arrived at the end of a certain time.
3. Modify the class `announcer` by passing it a number dispenser as a parameter and:
   (a) adding a method `wait_until` which returns `true` if the expected number has been called during a given waiting period, and `false` otherwise;
   (b) modifying the method `call` to take a counter as parameter and update the field `nclient` of this counter (it is necessary to add an update method in the `counter` class).
4. Modify the function `clerk` to take fruitless waits into account.
5. Write a function `impatient_client` which simulates the behaviour of an impatient client.

Object Producers and Consumers

This exercise revisits the producer-consumer algorithm with the following variation: the storage warehouse is of finite size (*i.e.* a table rather than a list managed as a FIFO). Also, we propose to make an implementation that uses objects to model resources, like the post office.

1. Define a class `product` with signature:
```
class product : string →
  object
```

```
    val name : string
    method name : string
  end
```

2. Define a class shop such that:

```
class show : int →
  object
    val mutable buffer : product array
    val c : Condition.t
    val mutable ic : int
    val mutable ip : int
    val m : Mutex.t
    val mutable np : int
    val size : int
    method dispose : product → unit
    method acquire : unit → product
  end
```

   The indexes `ic` and `ip` are manipulated by the producers and the consumers, respectively. The index `ic` holds the index of the last product taken and `ip` that of the last product stored. The counter `np` gives the number of products in stock. Mutual exclusion and control of the waiting of producers and consumers will be managed by the methods of this class.

3. Define a function `consumer`: *shop → string → unit*.

4. Define a function `create_product` of type *string -> product*. The name given to a product will be composed of the string passed as an argument concatenated with a product number incremented at every invocation of the function.
   Use this function to define `producer`: *shop → string → unit*.

Summary

This chapter tackled the topic of concurrent programming in which a number of processes interact, either through shared memory, or by synchronous communication. The first case represents concurrency for imperative programming. In particular, we have detailed the mechanisms of mutual exclusion whose use permits the synchronization of processes for access to shared memory. Synchronous communication offers a model for concurrency in functional programming. In particular, the possibility of sending closures and synchronization events on communication channels facilitates the composition of calculations carried out in different processes.

The processes used in this chapter are the threads of the Objective Caml `Thread` module.

To Learn More

The first requirements for concurrent algorithms arose from systems programming. For this application, the imperative model of shared memory is the most widely used. For example, the relation of mutual exclusion and semaphores are used to manage shared resources. The different low-level mechanisms of managing processes accessing shared memory are described in [Ari90].

Nonetheless, the possibility of expressing concurrent algorithms in one's favorite languages makes it possible to investigate this kind of algorithm, as presented in [And91]. It may be noted that while the concepts of such algorithms can simplify the solution of certain problems, the production of the corresponding programs is quite hard work.

The model of synchronous communication presented by CML, and followed by the `Event` module, is fully described in [Rep99]. The online version is at the following address:

Link: http://cm.bell-labs.com/cm/cs/who/jhr/index.html

An interesting example is the threaded graphical library `EXene`, implemented in CML under X-Windows. The preceding link contains a pointer to this library.