The Equeue user's guide

Gerd Stolpmann

The Equeue user's guide
by Gerd Stolpmann

Equeue offers general purpose event queues and event queues for system events.

Download Equeue:. The free Equeue library can be downloaded at http://www.ocaml-programming.de/packages/. The user's guide is included. Newest releases of Equeue will be announced in The OCaml Link Database (http://www.npc.de/ocaml/linkdb/).

License

Table of Contents

Chapter 1. Introduction into event-driven programming

Event-driven programming is an advanced way of organizing programs around I/O channels. This may be best explained by an example: Consider you want to read from a pipeline, convert all arriving lowercase letters to their corresponding uppercase letters, and finally write the result into a second pipeline.

A conventional solution works as follows: A number of bytes is read from the input pipeline into a buffer, converted, and then written into the output pipeline. Because we do not know at the beginning how many bytes will arrive, we do not know how big the buffer must be to store all bytes; so we simply decide to repeat the whole read/convert/write cycle until the end of input is signaled.

In O'Caml code:

```
let buffer_length = 1024 in
let buffer = String.create buffer_length in
try
  while true do
    (* Read up to buffer_length bytes into the buffer: *)
    let n = Unix.read Unix.stdin buffer 0 buffer_length in
    (* If n=0, the end of input is reached. Otherwise we have
     * read n bytes.
     *)
    if n=0 then
      raise End_of_file;
    (* Convert: *)
    let buffer' = String.uppercase (String.sub buffer 0 n) in
    (* Write the buffer' contents: *)
    let m = ref 0 in
    while !m < n do
      m := !m + Unix.write Unix.stdout buffer' !m (n - !m)
    done
  done
with
  End_of_file -> ()
```

The input and output pipelines may be connected with any other endpoint of pipelines, and may be arbitrary slow. Because of this, there are two interesting phenomenons. First, it is possible that the `Unix.read` system call returns less than `buffer_length` bytes, even if we are not almost at the end of the data stream. The reason might be that the pipeline works across a network connection, and that just a network packet arrived with less than `buffer_length` bytes. In this case, the operating system may decide to forward this packet to the application as soon as possible (but it is free not to decide so). The same may happen when `Unix.write` is called; because of this the inner `while` loop invokes `Unix.write` repeatedly until all bytes are actually written.

Nevertheless, `Unix.read` guarantees to read at least one byte (unless the end of the stream is reached), and `Unix.write` always writes at least one byte. But what happens if there is currently no byte to return? In this case, the second phenomenon happens: The program stops until at least one byte is available; this is called *blocking*.

Consider that the output pipeline is very fast, and that the input pipeline is rather slow. In this case, blocking slows down the program such that it is as slow as the input pipeline delivers data.

Consider that both pipelines are slow: Now, the program may block because it is waiting on input, but the output pipeline would accept data. Or, the program blocks because it waits until the output side is ready, but there have already input bytes arrived which cannot be read in because the program blocks. In these cases, the program runs much slower than it could do if it would react on I/O possibilities in an optimal way.

The operating systems indicates the I/O possibilities by the `Unix.select` system call. It works as follows: We pass lists of file descriptors on which we want to react. `Unix.select` also blocks, but the program continues to run already if *one* of the file descriptors is ready to perform I/O. Furthermore, we can pass a timeout value.

Here is the improved program:

```
let buffer_length = 1024 in
let in_buffer = String.create buffer_length in
let out_buffer = String.create buffer_length in
let out_buffer_length = ref 0 in
let end_of_stream = ref false in
let waiting_for_input = ref true in
let waiting_for_output = ref false in

while !waiting_for_input or !waiting_for_output do
  (* If !waiting_for_input, we are interested whether input arrives.
   * If !waiting_for_output, we are interested whether output is
   * possible.
   *)
  let (in_fd, out_fd, oob_fd) =
    Unix.select (if !waiting_for_input then [ Unix.stdin] else [])
                (if !waiting_for_output then [ Unix.stdout] else [])
                []
                (-.1.0) in

  (* If in_fd is non-empty, input is immediately possible and will
   * not block.
   *)
  if in_fd <> [] then begin
    (* How many bytes we can read in depends on the amount of
     * free space in the output buffer.
     *)
    let n = buffer_length - !out_buffer_length in
    assert(n > 0);
    let n' = Unix.read Unix.stdin in_buffer 0 n in
    end_of_stream := (n' = 0);
    (* Convert the bytes, and append them to the output buffer. *)
    let converted = String.uppercase (String.sub in_buffer 0 n') in
    String.blit converted 0 out_buffer !out_buffer_length n';
    out_buffer_length := !out_buffer_length + n';
  end;

  (* If out_fd is non-empty, output is immediately possible and
   * will not block.
   *)
  if out_fd <> [] then begin
    (* Try to write !out_buffer_length bytes. *)
    let n' = Unix.write Unix.stdout out_buffer 0 !out_buffer_length in
    (* Remove the written bytes from the out_buffer: *)
    String.blit out_buffer n' out_buffer 0 (!out_buffer_length - n');
    out_buffer_length := !out_buffer_length - n'
  end;

  (* Now find out which event is interesting next: *)
```

```
  waiting_for_input :=                    (* Input is interesting if...*)
    not !end_of_stream &&                 (* ...we are before the end *)
    !out_buffer_length < buffer_length;   (* ...there is space in the out buf *)

  waiting_for_output :=                   (* Output is interesting if... *)
    !out_buffer_length > 0;               (* ...there is material to output *)

done
```

Most important, we must now track the states of the I/O connections ourselves. The variable `end_of_stream` stores whether the end of the input stream has been reached. In `waiting_for_input` it is stored whether we are ready to accept input data. We can only accept input if there is space in the output buffer. The variable `waiting_for_output` indicates whether we have data to output or not. In the previous program, these states were implicitly encoded by the "program counter", i.e. which next statement was to be executed: After the `Unix.read` was done we *knew* that we had data to output; after the `Unix.write` we *knew* that there was again space in the buffer. Now, these states must be explicitly stored in variables because the structure of the program does not contain such information anymore.

This program is already an example of event-driven programming. We have two possible *events*: "Input arrived", and "output is possible". The `Unix.select` statement is the *event source*, it produces a sequence of events. There are two *resources* which cause the events, namely the two file descriptors. We have two *event handlers*: The statements after `if in_fd <> [] then` form the input event handler, and the statements after `if out_fd <> [] then` are the output event handler.

The `Equeue` module now provides these concepts as abstractions you can program with. It is a general-purpose event queue, allowing to specify an arbitrary event source, to manage event handlers, and offering a system how the events are sent to the event handlers that can process them. The `Unixqueue` module is a layer above `Equeue` and deals with file descriptor events. It has already an event source generating file descriptor events using the `Unix.select` system call, and it provides a way to manage file descriptor resources.

Especially the `Unixqueue` abstraction is an interesting link between the operating system and components offering services on file descriptors. For example, it is possible to create one event queue, and to attach several, independent components to this queue, and to invoke these components in parallel. For instance, consider a HTTP proxy. Such proxies accept connections and forward them to the service that can best deal with the requests arriving. These services are typically a disk cache, a HTTP client, and an FTP client. Using the `Unixqueue` model, you can realize this constellation by creating one event queue, and by attaching the services to it which can be independently programmed and tested; finally these components communicate either directly with the outer world or with other components only by putting events onto the queue and receiving events from this queue.

Chapter 2. The Equeue module

2.1. Description

The interface of the `Equeue` module.

```
type 'a t              (* Event systems over events of type 'a *)

exception Reject       (* Possible reaction of an event handler *)
exception Terminate    (* Possible reaction of an event handler *)

exception Out_of_handlers     (* Error condition *)

val create      : ('a t -> unit) -> 'a t
val add_event   : 'a t -> 'a -> unit
val add_handler : 'a t -> ('a t -> 'a -> unit) -> unit
val run         : 'a t -> unit
```

The values of type `Equeue.t` are called *event systems*, and contain:

- An event source, which is simply a function that gets the event system as argument and that may add further events to the system by invoking `Equeue.add_event`. The event source must be passed to `Equeue.create` as argument; it is not possible to change the source later.
- A list of event handlers. Handlers are added to the system by calling `Equeue.add_handler`.
- A queue of events waiting to be delivered to one of the handlers. You can add an event to the queue by invoking `Equeue.add_event`.

The module is intended to be used as follows: First, an event system is created, and initialized with an event source. Some event handlers are added:

```
let some_source esys = ... in

let handler1 esys e = ... in
let handler2 esys e = ... in
... (* more handlers *)

let esys = Equeue.create some_source in
Equeue.add_handler esys handler1;
Equeue.add_handler esys handler2;
... (* more handlers *)
```

It is necessary that at least one handler is added. In the second step, the event system can be started:

```
Equeue.run esys
```

This means the following:

- At the beginning, the function realizing the event source is called once. The function has the chance to add the first event(s) to the event queue by calling `Equeue.add_event`.

- If the event queue is not empty: It is iterated over all events currently in the queue. Every event is tried to be delivered to a handler by the simplest possible algorithm: The handlers are tried in turn, and the first handler that wants to consume the event gets the event.
- After one round of iteration over all events, it is possible that the handlers did already add further events to the queue, or it is possible that the queue is now empty. In the first case, the iteration is simply repeated with the newly added events. In the second case, the event source is called. If there are now events, they are iterated.
- Otherwise, the event system terminates.

A handler can indicate either that it wants to consume the event, or that it rejects the event, or that it wants to be removed from the list of handlers. Consumption is indicated by returning normally. Rejection is indicated by raising the `Equeue.Reject` exception. If the handler raises the `Equeue.Terminate` exception, the event is consumed and the handler is removed from the list of handlers.

Other exceptions, either raised within the event source function or within a handler function, simply fall through the event loop; they are not caught. However, the event system is restartable, which means:

- If the exception happened within the event source, the source is called again.
- If the exception happened within a handler function, the current event is scheduled again.

The event source is called when there are no events in the equeue. Note that the event source may not only add events, but also event handlers. It is an error if after the invocation of the event source there are events in the queue, but no handlers are defined. In this case, the exception `Out_of_handlers` is raised.

2.2. A silly example

Two kinds of events:

```
type event =
  A of int
| B
```

This event source produces ten events from `A 1` to `A 10`:

```
let n = ref 1
let source esys =
  if !n <= 10 then begin
    Equeue.add_event esys (A !n);
    incr n
  end
```

The handler for type A events puts as many type B events on the queue as the argument counts.

```
let handler_a esys e =
  match e with
    A n ->
      for i = 1 to n do
        Equeue.add_event esys B
      done
  | _ ->
      raise Equeue.Reject
```

The handler for type B events simply prints the events:

```
let handler_b esys e =
  match e with
    B ->
      print_endline "B"
  | _ ->
      raise Equeue.Reject
```

Finally, we set up the event system and start it:

```
let esys = Equeue.create source in
Equeue.add_handler esys handler_a;
Equeue.add_handler esys handler_b;
Equeue.run esys;
```

As result, the program prints 55 Bs.

Chapter 3. The Unixqueue module

3.1. Description

The interface of the `Unixqueue` module.

```
open Unix
open Sys

type group                        (* Groups of events *)
type wait_id                      (* Wait ticket *)

type operation =
    Wait_in  of file_descr         (* wait for input data *)
  | Wait_out of file_descr         (* wait until output can be written *)
  | Wait_oob of file_descr         (* wait for out-of-band data *)
  | Wait of wait_id                (* wait only for timeout *)

type event =
    Input_arrived of (group * file_descr)
  | Output_readiness of (group * file_descr)
  | Out_of_band of (group * file_descr)
  | Timeout of (group * operation)
  | Signal
  | Extra of exn

type event_system

exception Abort of (group * exn);;

val create_unix_event_system : unit -> event_system
val new_group : event_system -> group
val new_wait_id : event_system -> wait_id
val exists_resource : event_system -> operation  -> bool
val add_resource : event_system -> group -> (operation * float) -> unit
val add_close_action :
      event_system -> group -> (file_descr * (file_descr -> unit))
        -> unit
val add_abort_action :
      event_system -> group -> (group -> exn -> unit)
        -> unit
val remove_resource : event_system -> group -> operation -> unit
val add_handler :
      event_system -> group ->
      (event_system -> event Equeue.t -> event -> unit)
         -> unit
val add_event : event_system -> event -> unit
val clear : event_system -> group -> unit
val run : event_system -> unit
val once : event_system -> group -> float -> (unit -> unit)
               -> unit
```

Theme of this module are four types of operations: Waiting for input data, waiting for output readiness, waiting for out-of-band data, and waiting for a period of time. You can associate resources which simply means that it is waited until one of the operations becomes possible or is timed out. Resources are the combination of an operation and a time-out value.

This module already implements an event source which checks whether the operations are possible or timed-out, and which generates events describing what has happended. As with `Equeue` you can add events yourself, and you can add handlers which perform actions on certain events. As `Unixqueue` is based on `Equeue`, the queue model is simply the same.

Resources, handlers and events are grouped, i.e. you can reference to a bundle of resources/events by specifying the group they belong to. Group are created by `Unixqueue.new_group`, and every resource must belong to a group. The events caused by a resource belong to the same group as the resource. Handlers have a group, too, and the handlers only get events of the same group (*this has been changed between Equeue 1.1 and 1.2*). Furthermore, the groups simplify clean-up actions.

You can associate a close action with a resource. The close action is automatically called if the resource is removed from the queue. The close action should then close the underlying file descriptor.

You can remove all handlers and resouces belonging to a group with only one function call (`clear`).

There is special support for aborting groups. Aborting means that an exception is to be raised, but there needs to be some cleanup before the event queue is in a sane state. Because of this, it is possible to asscociate an abort action with a group. The abort action is automatically performed if the special Abort exception is raised within a handler.

Details:

- `Unixqueue.create_unix_event_system()`: Returns a fresh event system with empty queue, no handlers, but already including the built-in event source.
- `new_group ues`: Returns a new group descriptor for the event system `ues`.
- `new_wait_id ues`: Returns a new wait-only identifier for the event system `ues`. You need such an ID to form a `Wait` operation. In constrast to the operations for file I/O, the various wait-only operations would otherwise not be distinguishable.
- `add_resource ues g (op,t)`: Adds a new resource to the event system `ues`. The resource belongs to the group `g`, and watches the operation `op` with the timeout value `t` (in seconds). A negative timeout value means no timeout. If you try to add two resources for the same operation, the secondly added resource is silently ignored.
- `exists_resource ues op`: Returns true iff the event system `ues` contains a resource for operation `op`.
- `remove_resource ues g op`: Removes the resource from the event system `ues` which belongs to group `g` and watches the operation `op`. If there is no such resource, `Not_found` is raised. If a file descriptor is no longer contained in any resource because of this removal, the associated close action is performed.
- `add_close_action ues g (fd, action)`: Adds the close action `action` to the file descriptor `fd` in the event system `ues`. There must already be some resource watching the file descriptor `fd`. The group `g` associates the close action to a group; see `clear`. The close action is called if the last resource for the specified file descriptor is removed. - If several close actions are added for the same file descriptor, the last added action hides the previous ones.
- `add_handler ues g handler`: Adds the `handler` to the event system `ues`, and associates it to the group `g`.
- `add_abort_action ues g action`: Adds the abort action `action` for group `g` to the event system `ues`. The abort action is invoked if the exception `Abort(g,x)` is raised within a handler. After the action has been performed, the exception `x` is raised. - If several abort actions are added for the same group, the last added action hides the previous ones.
- `clear ues g`: Removes all entities associated with group `g` in the event system `ues`. This means that all resources, handlers, close actions, and abort actions belonging to the group are deleted. Note that the removal of resources may still trigger the close action if the last resource for a certain file descriptor is deleted. Furthermore, the cleared group can no longer be used for regular operations (in Equeue 1.2, this is enforced:

handlers of cleared groups do not get any events; and it is not possible to add new handlers for a cleared group).

- `add_event ues e`: Adds the event `e` to the event system `ues`.
- `once ues g t action`: Adds a resource and a handler such that the `action` will be invoked after `t` seconds. The new entities are associated with the group `g`. The resource and the handler are removed once the action is over.
- `run ues`: Starts the event system.

Object-oriented interface. In addition to the functional interface, there is also an object-oriented interface. Instead of calling one of the above functions *`f`*, one can also invoke the method with the same name. For example, the call

```
add_resource ues g (op,t)
```

can also be written as

```
ues # add_resource g (op,t)
```

Both styles can be used in the same program, and there is absolutely no difference (actually, the object-oriented interface is even the fundamental interface, and the functions are just wrappers for the method calls).

Instead of creating the event system with

```
let ues = create_unix_event_system()
```

one can also use

```
let ues = new unix_event_system()
```

Again, both calls do exactly the same.

The object-oriented interface has been introduced to support other implementations of file descriptor polling than [Unix.select]. The integration into the Tcl and Glib event systems has been implemented by defining additional classes that are compatible with `Unixqueue.unix_event_system`, but internally base on different polling mechanisms.

3.2. Example: Copying several files in parallel

We present here a function which adds a file copy engine to an event system. It is simple to add the engine several times to the event system to copy several files in parallel.

```
open Unixqueue

type copy_state =
    { copy_ues    : Unixqueue.event_system;
      copy_group  : Unixqueue.group;
      copy_infd   : Unix.file_descr;
      copy_outfd  : Unix.file_descr;
      copy_size   : int;
      copy_inbuf  : string;
      copy_outbuf : string;
      mutable copy_outlen      : int;
      mutable copy_eof         : bool;
      mutable copy_have_inres  : bool;
```

```
    mutable copy_have_outres : bool;
    mutable copy_cleared     : bool;
  }
```

This record type contains the state of the engine.

- copy_ues: The event system to which the engine is attached
- copy_group: The group to which all the entities belong
- copy_infd: The file descriptor of the source file
- copy_outfd: The file descriptor of the copy file
- copy_size: The size of copy_inbuf and copy_outbuf
- copy_inbuf: The string buffer used to read the bytes of the source file
- copy_outbuf: The string buffer used to write the bytes to the copy file
- copy_outlen: The portion of copy_outbuf that is actually used
- copy_eof: Whether the EOF marker has been read or not
- copy_have_inres: Whether there is currently an input resource for the input file
- copy_have_outres: Whether there is currently an output resource for the output file
- copy_cleared: Whether the copy is over or not

Now the core function begins:

```
let copy_file ues old_name new_name =
  (* Adds the necessary handlers and actions to the Unixqueue.event_system
   * ues that copy the file 'old_name' to 'new_name'.
   *)
```

Several inner functions are defined now. First, update_resources adds or removes the resources involved into copying. The record components copy_have_inres and copy_have_outres store whether there is currently a resource for input and for output, respectively. It is computed whether a input or output resource is wanted; and then the resource is added or removed as needed. If both resources are deleted, the file descriptors are closed, and the event system is cleaned.

We want input if there is space in the output buffer, and the end of the input file has not yet been reached. If this is true, it is ensured that an input resource is defined for the input file such that input events are generated.

We want output if there is something in the output buffer. In the same manner it is ensured that an output resource is defined for the output file.

Note that normally the input and output resources are added and removed several times until the complete file is copied.

```
  let update_resources state ues =
    let want_input_resource =
      not state.copy_eof && state.copy_outlen < state.copy_size in
    let want_output_resource =
      state.copy_outlen > 0 in
    if want_input_resource && not state.copy_have_inres then
      add_resource ues state.copy_group (Wait_in state.copy_infd, -.1.0);
    if not want_input_resource && state.copy_have_inres then
      remove_resource ues state.copy_group (Wait_in state.copy_infd);
    if want_output_resource && not state.copy_have_outres then
      add_resource ues state.copy_group (Wait_out state.copy_outfd, -.1.0);
    if not want_output_resource && state.copy_have_outres then
```

```
      remove_resource ues state.copy_group (Wait_out state.copy_outfd);
    state.copy_have_inres <- want_input_resource;
    state.copy_have_outres <- want_output_resource;
    if not want_input_resource && not want_output_resource &&
      not state.copy_cleared
    then begin
      (* Close file descriptors at end: *)
      Unix.close state.copy_infd;
      Unix.close state.copy_outfd;
      (* Remove everything: *)
      clear ues state.copy_group;
      state.copy_cleared <- true;    (* avoid to call 'clear' twice *)
    end
  in
```

The input handler is called only for input events belonging to our own group. It is very similar to the example in the introductory chapter.

The input handler calls `update_resource` after the work is done. It is now possible that the output buffer contentains data after it was previously empty, and `update_resource` will then add the output resource. Or, it is possible that the output buffer is now full, and `update_resource` will then remove the input resource such that no more input data will be accepted. Of course, both conditions can happen at the same time.

```
  let handle_input state ues esys e =
    (* There is data on the input file descriptor. *)
    (* Calculate the available space in the output buffer: *)
    let n = state.copy_size - state.copy_outlen in
    assert(n > 0);
    (* Read the data: *)
    let n' = Unix.read state.copy_infd state.copy_inbuf 0 n in
    (* End of stream reached? *)
    state.copy_eof <- n' = 0;
    (* Append the read data to the output buffer: *)
    String.blit state.copy_inbuf 0 state.copy_outbuf state.copy_outlen n';
    state.copy_outlen <- state.copy_outlen + n';
    (* Add or remove resources: *)
    update_resources state ues
  in
```

The output handler is called only for output events of our own group, too.

The output handler calls `update_resource` after the work is done. It is now possible that the output buffer has space again, and `update_resource` will add the input resource again. Or, th output buffer is even empty, and `update_resource` will also remove the output resource.

```
  let handle_output state ues esys e =
    (* The file descriptor is ready to output data. *)
    (* Write as much as possible: *)
    let n' = Unix.write state.copy_outfd state.copy_outbuf 0 state.copy_outlen
    in
    (* Remove the written bytes from the output buffer: *)
    String.blit
      state.copy_outbuf n' state.copy_outbuf 0 (state.copy_outlen - n');
    state.copy_outlen <- state.copy_outlen - n';
    (* Add or remove resources: *)
```

```
    update_resources state ues
  in
```

This is the main event handler. It accepts only `Input_arrived` and `Output_readiness` events belonging to our own group. All other events are rejected.

```
  let handle state ues esys e =
    (* Only accept events associated with our own group. *)
    match e with
Input_arrived (g,fd) ->
  handle_input state ues esys e
      | Output_readiness (g,fd) ->
  handle_output state ues esys e
      | _ ->
  raise Equeue.Reject
  in
```

Now the body of the `copy_file` function follows. It contains only initializations.

```
  let g = new_group ues in
  let infd = Unix.openfile
       old_name
       [ Unix.O_RDONLY; Unix.O_NONBLOCK ]
       0 in
  let outfd = Unix.openfile
new_name
[ Unix.O_WRONLY; Unix.O_NONBLOCK; Unix.O_CREAT; Unix.O_TRUNC ]
0o666 in
  Unix.clear_nonblock infd;
  Unix.clear_nonblock outfd;
  let size = 1024 in

  let state =
    { copy_ues = ues;
      copy_group = g;
      copy_infd = infd;
      copy_outfd = outfd;
      copy_size = size;
      copy_inbuf = String.create size;
      copy_outbuf = String.create size;
      copy_outlen = 0;
      copy_eof = false;
      copy_have_inres = false;
      copy_have_outres = false;
      copy_cleared = false;
    } in

  update_resources state ues;
  add_handler ues g (handle state);
;;
```

Note that the files are opened in "non-blocking" mode. This ensures that the `Unix.openfile` system call does not block itself. After the files have been opened, the non-blocking flag is reset; the event system already guarantees that I/O will not block.

Now we can add our copy engine to an event system, e.g.

```
let ues = create_unix_event_system() in
copy_file ues "a.old" "a.new";
copy_file ues "b.old" "b.new";
run ues
;;
```

This piece of code will copy both files in parallel. Note that the concept of "groups" is very helpful to avoid that several instances of the same engine interfer with each other.

Chapter 4. Engines

Programming directly with Unixqueues can be quite ineffective. One needs a lot of code to perform even simple problems. The question arises whether there is a way to construct event-driven code from larger units that do more complicated tasks than just looking at the possible I/O operations of file descriptors. Ideally, there would be a construction principle that scales with the problems the programmer wants to solve.

An *engine* is an object bound to an event system that performs a task in an autonomous way. After the engine has started, the user of the engine can leave it alone, and let it do what it has been designed for, and simply wait until the engine has completed its task. The user can start several engines at once, and all run in parallel. It is also possible to construct larger engines from more primitive ones: One can run engines in sequence (the output of the first engine is the input of the next), one can run synchronize engines (when two engines are done the results of both engines are combined into a single result), and map the results of engines to different values.

4.1. Modelling the abstract properties of engines

The formalization of engines assumes that there are four major states (see the module `Uq_engines`):

```
type 't engine_state =
  [ `Working of int
  | `Done of 't
  | `Error of exn
  | `Aborted
  ]
```

A `` `Working `` engine is actively performing its task. The number argument counts the events that are processed while progressing. The state `` `Done `` indicates that the task is completed. The argument of `` `Done `` is the result value of the engine. The state `` `Error `` means that the engine ran into a problem, and cannot continue. Usually an exception was raised, and in order to be able to pass the exception to the outside world, it becomes the argument of `` `Error ``. Finally, an engine can be explictly `` `Aborted `` by calling the `abort` method. This forces that the engine stops and releases the resources it has allocated.

The last three states are called *final states* because they indicate that the engine has stopped. Once it is in a final state, the engine will never go back to `` `Working ``, and will also not transition into another final state.

There is no state for the situation that the engine has not yet begun operation. It is assumed that an engine starts performing its task right when it has been created, so the initial state is usually `` `Working 0 ``.

Engines are objects that implement this class type:

```
class type [ 't ] engine = object
  method state : 't engine_state
  method abort : unit -> unit
  method request_notification : (unit -> bool) -> unit
  method event_system : Unixqueue.event_system
end
```

The method `state` reports the state the engine currently has. By calling `abort` the engine is aborted. The method `request_notification` will be explained later. Finally, `event_system` reports the Unixqueue event system the engine is attached to.

4.2. Examples for engine primitives and engine construction

Fortunately, there are already some primitive engines we can just instantiate, and see what they are doing. The function `connector` creates an engine that connects to a TCP service in the network, and returns the connected socket as result:

```
val connector : ?proxy:#client_socket_connector ->
connect_address ->
Unixqueue.event_system ->
  connect_status engine
```

To create and setup the engine, just call this function, as in:

```
let ues = Unixqueue.create_unix_event_system() in
let addr = `Socket(`Sock_inet_byname(Unix.SOCK_STREAM, "www.npc.de", 80)) in
let eng = connector addr ues in
...
```

The engine will connect to the web server (port 80) on www.npc.de. It has added handlers and resources to the event system `ues` such that the action of connecting will be triggered when `Unixqueue.run` becomes active. To see the effect, just activate the event system:

```
Unixqueue.run ues
```

When the connection is established, `eng#state` changes to `` `Done(`Socket(fd,addr)) `` where `fd` is the socket, and `addr` is the logical address of the client socket (which may be different than the physical address because `connect` supports network proxies). It is also possible that the state changes to `` `Error e `` where `e` is the problematic exception. Note that there is no timeout value; to limit the time of engine actions one has to attach a watchdog to the engine.

This is not yet very impressive, because we have only a single engine. As mentioned, engines run in parallel, so we can connect to several web services in parallel by just creating several engines:

```
let ues = Unixqueue.create_unix_event_system() in
let addr1 = `Socket(`Sock_inet_byname(Unix.SOCK_STREAM, "www.npc.de", 80)) in
let addr2 = `Socket(`Sock_inet_byname(Unix.SOCK_STREAM, "caml.inria.fr", 80)) in
let addr3 = `Socket(`Sock_inet_byname(Unix.SOCK_STREAM, "ocaml-programming.de", 80)) in

let eng1 = connector addr1 ues in
let eng2 = connector addr2 ues in
let eng3 = connector addr3 ues in
Unixqueue.run ues
```

Note that the resolution of DNS names is not done in the background, and may block the whole event system for a moment.

As a variant, we can also connect to one service after the other:

```
let eng1 = connector addr1 ues in
let eng123 = new seq_engine
               eng1
               (fun result1 ->
          let eng2 = connector addr2 ues in
                  new seq_engine
            eng2
                   (fun result2 ->
                      let eng3 = connector addr3 ues in
                      eng3)))
```

The constructor for sequential engine execution, `seq_engine`, expects one engine and a function as arguments. When the engine is done, the function is invoked with the result of the engine, and the function must return a second engine. The result of `seq_engine` is the result of the second engine.

In these examples, we have called `Unixqueue.run` to start the event system. This function returns when all actions are completed; this implies that finally all engines are synchronized again (i.e. in a final state). We can also synchronize in the middle of the execution by using `sync_engine`. In the following code snipped, two services are connected in parallel, and when both connections have been established, a third connection is started:

```
let eng1 = connector addr1 ues in
let eng2 = connector addr2 ues in
let eng12 = new sync_engine eng1 eng2 in
let eng123 = new seq_engine
               eng12
               (fun result12 ->
                  let eng3 = connector addr3 ues in
                  eng3)
```

4.3. The notification mechanism

Often, one just wants to watch an engine, and to perform a special action when it reaches a final state. There is a simple way to configure a callback:

```
val when_state : ?is_done:('a -> unit) ->
 ?is_error:(exn -> unit) ->
 ?is_aborted:(unit -> unit) ->
 'a #engine ->
   unit
```

For example, to output a message when `eng1` is connected:

```
when_state ~is_done:(fun _ -> prerr_endline "eng1 connected") eng1
```

The argument of `is_done` is the result of the engine (not needed in this example).

The function `when_state` is implemented with the notification mechanism all engines must support. The method `request_notification` can be used to request a callback whenever the state of the engine changes:

```
method request_notification : (unit -> bool) -> unit
```

The callback function returns whether it is still interested in being called (`true`) or not (`false`). In the latter case, the engine must not call the function again.

For example, the connection message can also be output by:

```
eng1 # request_notification
         (fun () ->
    match eng1#state with
      `Done _ -> prerr_endline "eng1 connected"; false
           | `Error _
           | `Aborted -> false
           | `Working _ -> true
         )
```

Some more details: The callback function must be even called when only minor state changes occur, e.g. when `` `Working n `` changes to `` `Working (n+1) ``. The engine is free to invoke the callback function even more frequently.

Another detail: It is allowed that more callbacks are requested when a callback function is running.

Note that all engine construction classes base on the notification mechanism, so it is absolutely required that it is implemented error-free.

4.4. Asynchronous channels

Because engines are based on Unixqueues, one can imagine that complex operations on file descriptors are executed by engines. Actually, there is a primitive that copies the whole byte stream arriving at one descriptor to another descriptor: The class `copier`. We do not discuss this class in detail, it is explained in the reference manual. From the outside it works like every engine: One specifies the task, creates the engine, and waits until it is finished. Internally, the class has to watch both file descriptors, check when data can be read and written, and to actually copy chunk by chunk.

Now imagine we do not only want to copy from descriptor to descriptor, but to copy from a descriptor into a data object. Of course, we have the phenomenon that the descriptor sometimes has data to be read and sometimes not, this is well-known and can be effectively handled by Unixqueue means. In addition to this, we assume that there is only limited processing capacity in the data object, so it can sometimes accept data and sometimes not. This sounds the same, but it is not, *because there is no descriptor to which this phenomenon is bound*. We have to develop our own interface to mimick this behaviour on a higher programming level: The asynchronous output channel.

The term *channel* is used by the O'Caml runtime system to refer to buffered I/O descriptors. The Ocamlnet library has extended the meaning of the term to objects that handle I/O in a configurable way. As this is what we are going to do, we adopt this meaning.

An asynchronous output channel is a class with the type:

```
class type async_out_channel = object
  method output : string -> int -> int -> int
  method close_out : unit -> unit
  method pos_out : int
  method flush : unit -> unit
  method can_output : bool
  method request_notification : (unit -> bool) -> unit
```

The first four methods are borrowed from Ocamlnet's class type `raw_out_channel`:

- `output s k n` prints into the channel n bytes that can be found at position k of string s. The method returns the number of bytes that have been accepted
- `close_out()` closes the channel
- `flush()` causes that bytes found in internal buffers are immediately processed. Note that it is questionable what this means in an asynchronous programming environment, and because of this, we ignore this method.
- `pos_out` returns the number of bytes that have been written into the channel since its creation (as object)

Originally, these methods have been specified for synchronous channels. These are allowed to wait until a needed resource is again available - this is not possible for an asynchronous channel. For example, `output` ensures to accept at least one byte in the original specification. An implementation is free to wait until this is possible. Here, we should not do so because this would block the whole event system. Instead, there are two additional methods helping to cope with these difficulties:

- `can_output` returns true when `output` accepts at least one byte, and false otherwise
- `request_notification f` requests that the function f is called back whenever `can_output` changes its value

The point is that now the *user* of an asynchronous channel is able to defer the output operation into the future when it is currently not possible. Of course, it is required that the user knows this - using an asynchronous channel is not as easy as using a synchronous channel.

We show now two examples: The first always accepts output and appends it to a buffer. Of course, the two methods `can_output` and `request_notification` are trivial in this case. The second example illustrates these methods: The channel pauses for one second after one kilobyte of data have been accepted. This is of little practical use, but quite simple to implement, and has the right niveau for an example.

Example 1: We just inherit from an Ocamlnet class that implements the buffer:

```
class async_buffer b =
object (self)
  inherit Netchannels.output_buffer b
  method can_output = true
  method request_notification (f : unit->bool) = ()
end
```

I insist that this is a good example because it demonstrates why the class type `async_out_channel` bases on an Ocamlnet class type. (Note that `async_buffer` defines more methods than necessary. It might be necessary to coerce objects of this class to `async_out_channel` if required by typing.)

Example 2: Again we use an Ocamlnet class to implement the buffer, but we do not directly inherit from this class. Instead we instantiate it as an instance variable `real_buf`. The variable `barrier_enabled` is true as long as no more than 1024 bytes have been written into the buffer, *and* the sleep second is not yet over. The variable `barrier_reached` is true if at least 1024 bytes have been written into the buffer.

```
   class funny_async_buffer b ues =
   object (self)
     val real_buf = new Netchannels.output_buffer b
     val mutable barrier_enabled = true
     val mutable barrier_reached = false
     val mutable notify_list = []
     val mutable notify_list_new = []

     method output s k n =
       if barrier_enabled then (
 let m = 1024 - real_buf#pos_out in
         let r = real_buf # output s k (min n m) in
         if m > 0 && real_buf#pos_out = 1024 then (
           barrier_reached <- true;
           self # configure_sleep_second();
           self # notify()
         );
         r
       )
       else
         real_buf # output s k n

     method flush() = ()
```

```
    method pos_out = real_buf#pos_out

    method close_out() = real_buf#close_out()

    method can_output =
      if barrier_enabled then
        not barrier_reached
      else
        true

    method request_notification f =
      notify_list_new <- f :: notify_list_new

    method private notify() =
      notify_list <- notify_list @ notify_list_new;
      notify_list_new <- [];
      notify_list <- List.filter (fun f -> f()) notify_list

    method private configure_sleep_second() =
      let g = Unixqueue.new_group ues in
      Unixqueue.once ues g 1.0 self#wake_up

    method private wake_up() =
      barrier_enabled <- false;
      self # notify()
  end
```

Initially, the barrier is enabled, and `can_output` returns `true`. The logic in `output` ensures that no more than 1024 bytes are added to the buffer. When the 1024th byte is printed, the barrier is reached, and the sleep second begins. `can_output` changes to `false`, and because of this, we must `notify` the functions that have requested that. The timer is implemented by a call of `Unixqueue.once`; this function performs a callback after a period of time has elapsed. Here, `wake_up` is called back. It disables the barrier, and because `can_output` is now again `true`, the notifications have to be done again.

The complete example can be found in the "examples/engines" directory of the equeue distribution.

An implementation of a useful asynchronous channel is `output_async_descr` that outputs the channel data to a file descriptor. This class is also an engine. See the reference manual for a description.

4.5. Receivers

The question is what one can do with asynchronous channels. We have mentioned that these objects were designed with copy tasks in mind that transfer data from file descriptors into data objects. Of course, the asynchronous channels play the role of these data objects. In addition to these, we need an engine that actually performs this kind of data transfer: The receiver engine.

The receiver class has this signature:

```
class receiver : src:Unix.file_descr ->
 dst:#async_out_channel ->
 ?close_src:bool ->
 ?close_dst:bool ->
 Unixqueue.event_system ->
   [unit] engine
```

Obviously, `src` is the descriptor to get the data from, and `dst` is the asynchronous channel to write the data into. After the receiver has been created, it copies the data stream from `src` to `dst` until EOF is found.

The receiver is an engine, and this means that it reports its state to the outer world. When the copy task has been completed, it transitions into the state `` `Done()``.

In the next section we present a real example that also uses the receiver class.

4.6. Example: A simple HTTP client

The HTTP protocol is used to get web pages from web servers. Its principle is very simple: A request is sent to the server, and the server replies with the document (well, actually HTTP can be very complicated, but it can also still be used in this simple way). For example, the request could be

```
GET / HTTP/1.0
--empty line--
```

Note there is a second line which is empty. The server responds with a header, an empty line, and the document. In HTTP/1.0 we can assume that the server sends EOF after the document.

The first part of our client connects to the web server. This is not new:

```
let ues = Unixqueue.create_unix_event_system();;
let c = connector (`Socket(`Sock_inet_byname(Unix.SOCK_STREAM,
     "www.npc.de", 80),
   default_connect_options
  )) ues;;
```

Furthermore, we need an asynchronous output channel that stores the incoming server reply. This is also a known code snippet:

```
 class async_buffer b =
 object (self)
   inherit Netchannels.output_buffer b
   method can_output = true
   method request_notification (f : unit->bool) = ()
 end
```

We also create a buffer:

```
let b = Buffer.create 10000;;
```

Now we are interested in the moment when the connection is established. In this moment, we set up an `output_async_descr` object that copies its contents to the connection, so we can asynchronously send our HTTP request. Furthermore, we create an `async_buffer` object that collects the HTTP response, which can arrive at any time from now on.

```
  when_state
    ~is_done:(fun connstat ->
match connstat with
    `Socket(fd, _) ->
      prerr_endline "CONNECTED";
      let printer = new output_async_descr ~dst:fd ues in
      let buffer = new async_buffer b in
      let receiver = new receiver ~src:fd ~dst:buffer ues in
      let s = "GET / HTTP/1.0\n\n" in
```

```
    ignore(printer # output s 0 (String.length s));
            when_state
                    ~is_done:(fun _ ->
                                prerr_endline "HTTP RESPONSE RECEIVED!")
              ~is_error:(fun _ ->
                                prerr_endline "ERROR!")
                    receiver
| _ -> assert false
  )
  c
;;
```

Some details: We can ignore the result of `printer#output` because the `printer` has unlimited capacity (the default of `output_async_descr` channels). Because `printer` is not closed, this channel does not close the destination descriptor `fd` (which would be fatal). The `receiver`, however, closes the file descriptor when it finds the end of the input stream.

One important line is missing: Up to now we have only set up the client, but it is not yet running. To invoke it we need:

```
Unixqueue.run ues;;
```

This client is not perfect, not only, because it is restricted to the most basic form of the HTTP protocol. The error handling could be better: In the case that `printer` transitions to error state, it will close `fd`. But the file descriptor is in use by `receiver` at the same time, causing a followup error that is finally reported. A better solution would use a duplicate of the file descriptor for `receiver`, so both engines can independently close their descriptors. Furthermore, the error state of `printer` would be trapped.

Chapter 5. Event-driven programming vs. multi-threaded programming

One of the tasks of event-driven programming is to avoid blocking situations, another is to schedule the processor activities. Another approach to achieve these goals is multi-threaded programming.

The fundamental difference between both approaches is that in the case of event-driven programming the application controls itself, while in the case of multi-threaded programming additional features of the operating system are applied. The latter seems to have major advantages, for example blocking is impossible at all (if one thread blocks, the other threads may continue running), and scheduling is one of the native tasks of an operating system.

This is not the whole truth. First of all, multi-threaded programming has the disadvantage that every line of the program must follow certain programming guidelines, especially shared storage must be protected by mutexes such that everything is "reentrant". This is not very simple. On the contrary, event-driven programs can be "plugged together" from a set of basic components, and you do not need to know how the components are programmed.

Scheduling: Multi-threaded programs sometimes lead to situations where there are many runnable threads. Despite the capabilities of the operating system, every modern hardware has the restriction that it performs badly if the code to execute is wide-spread over the whole memory. This is mainly caused by limited cache memory. Many operating systems are not well enough designed to efficiently get around this bottleneck.

Furthermore, I think that scheduling controlled by the application that knows best its own requirements cannot be worse than scheduling controlled by the operating system. (But this may be wrong in special situations.)

Avoid blocking: Of course, an event-driven program blocks if it gets into an endless loop. A multi-threaded application does not block in this case, but it wastes CPU time. It is normally not possible to kill single wild-running threads because most programs are not "cancellation-safe" (a very high requirement). In O'Caml, the latter is only possible for the bytecode thread emulation.

Of course, if you must combine some non-blocking I/O with time-consuming computations, the multi-threaded program will block "less" (it becomes only slower) than the event-driven program, which is unavailable for a period of time.

To come to an end, I think that there are many tasks where event-driven programs perform as well as multi-threaded programs, but where the first style has fewer requirements on the quality of the code.

5.1. Combining both styles

Since Equeue 1.2, it is possible to use Equeue in a multi-threaded environment. The fundamental Equeue module is reentrant, and the Unixqueue module even serializes the execution of functions if necessary, such that the same event system may be used from different threads.

One idea is to program a hybrid server in the following way: One thread does all network I/O (using event systems), and the other threads execute the operations the server provides. For example, consider a server doing remote procedures (as most servers do). Such a server receives requests, and every request is responded. When the server starts up, the networking thread begins to wait for requests. When a complete request has been received, a new thread is started performing the requested operation. The network thread continues immediately, normally doing other network I/O. When the operation is over, an artificial event is generated indicating this situation (see below on artificial events). The artificial event carries the result of the operation, and is added to the event system directly from the thread that executed the operation. This thread can now stop working. The network thread receives this artificial event like every other event, and can start sending the result over the network back to the client.

Artificial events are new in Equeue 1.2, too. The idea is to use O'Caml exceptions as dynamically extensible sum type. For example:

```
exception Result of result_type ;;
...
add_event esys (Extra (Result r))
```

The `Extra` event constructor can carry every exception value.

Caveat. The `Extra` events are not associated to any group. Every event handler will get them.

Chapter 6. Pitfalls

There are some situations where the program may still block, if it is not programmed very carefully.

- Despite the `open` system call, the `connect` system call may also block. To avoid blocking, you must first set the socket to non-blocking mode. E.g.

  ```
  let s = Unix.socket Unix.PF_INET Unix.SOCK_STREAM 0 in
  Unix.set_nonblock s;
  Unix.connect s some_address;
  ```

- Other blocking candidates are the name-server functions, in O'Caml `Unix.inet_addr_of_string`. This is very hard to solve because the underlying C library performs the DNS lookup. The POSIX thread implemention does not help, because special DNS functions needs to be called to avoid blocking (these functions have a reentrant function interface), and the O'Caml `Unix` module does not use them. A possible solution is to fork a new process, and let the new process perform the DNS lookup.

Chapter 7. Using Unixqueue together with Tcl (labltk) and Glib (lablgtk)

The Tcl programming language has already an event queue implementation, and the Tk toolkit applies it to realize event queues for graphical user interfaces (GUIs). In the O'Caml world, Tcl/Tk is available through the packages camltk and labltk.

The same holds for the Glib library which is used by the gtk GUI toolkit to implement event queues. In the O'Caml world, gtk bindings are provided by lablgtk (and lablgtk2).

While the GUI queues mainly process GUI events (e.g. mouse and keyboard events), they can also watch files in the same way as Unixqueue does it. It is, however, not possible to run both types of queues in parallel, because there is the problem that when one type of queue blocks, the other is implicitly also blocked, even when there would be events to process. The solution is to *integrate* both queues, and because the GUI queues can subsume the functionality of Unixqueue, the GUI queues are the more fundamental ones, and Unixqueue must integrate its event processing into the GUI queues.

This type of integration into the GUI queues is implemented by defining the alternate classes `Uq_tcl.tcl_event_system` and `Uq_gtk.gtk_event_system`. These classes can be used in the same way as `Unixqueue.unix_event_system`, but automatically arrange the event queue integration.

For example, a labltk program uses

```
let ues = new Uq_tcl.tcl_event_system()
```

to create the event system object, which can be used in the same way as event systems created with `create_unix_event_system` or `new unix_event_system`. There is one important difference, however. One must no longer call `Unixqueue.run` to start the processing. The reason is that the TCL queue is already started, and remains active during the runtime of the program. Remember that when the GUI function

```
Tk.mainLoop()
```

is entered, the TCL queue becomes active, and all subsequent execution of O'Caml code is triggered by callback functions. The integrated queue now behaves as follows: When handlers, resources, or events are added to `ues`, they are automatically considered for processing when the current callback function returns. For example, this might look as follows:

```
let b1 = Button.create
           ~text:"Start function"
           ~command:(fun () ->
                      Unixqueue.add_handler ues ...; ...) widget in
let b2 = Button.create
           ~text:"Stop function"
           ~command:(fun () ->
                      Unixqueue.remove_handler ues ...; ...) widget in
...
```

When the button is pressed, the function is triggered, and the callback function passed as `command` starts executing. This adds handlers, resources, and what ever is needed to start the activated function. The callback function returns immediately, and the processing of the event queue is performed by the regular GUI event system. Of course, it is still possible to press other buttons etc., because GUI and Unixqueue events are processed in an interweaved way. So the user is able to press the "Stop" button to stop the further execution of the activated function.

API change. In Equeue-2.1, the interface for the Tcl queue integration was changed. It works now as described above; the function `Unixqueue.attach_to_tcl_queue` no longer exists. The new scheme has the advantage that the Glib-type queues (and probably any other event queue implementation) can be also easily supported.

Sample code. The example discussed before, copying files in an event-driven way, has been extended to show how Unixqueue and Tcl can cooperate. While the file is being copied, a window informs about the progress and offers a "Stop" button which immediately aborts the copy procedure. See the directory "filecopy_labltk" in the distributed tarball. There is also a variant that works with lablgtk or lablgtk2, see the directory "filecopy_lablgtk".

Pitfalls. If you call Unixqueue functions from Unixqueue event handlers, the functions behave exactly as described in the previous chapters. However, it is also possible to call Unixqueue functions from TCL/Glib event handlers. In this case, not all change requests will be immediately honoured. Especially, `add_event` does not immediately invoke the appropriate event handler; the event is just recorded, and the handler will be called when the next *system* event happens (either a GUI event, a file descriptor event, or a timeout event). You can force to respect the new event as soon as possible by adding an empty handler using `Unixqueue.once` with a timeout of 0 seconds. - The other Unixqueue functions should not behave differently (although the actually performed operations are very different). Especially you can call `add_resource` and `remove_resource` and the change will be respected immediately.