LibEvent Programmers Manual

David F. Skoll
Roaring Penguin Software Inc.

September 30, 2002

1 Introduction

Many UNIX programs are event-driven. They spend most of their time waiting for an event, such as input from a file descriptor, expiration of a timer, or a signal, and then react to that event.

The standard UNIX mechanisms for writing event-driven programs are the **select** and **poll** system calls, which wait for input on a set of file descriptors, optionally with a timeout.

While **select** and **poll** can be used to write event-driven programs, their calling interface is awkward and their level of abstraction too low. LibEvent is built around **select**, but provides a more pleasant interface for programmers.

LibEvent provides the following mechanisms:

- *Events*, which trigger under user-specified conditions, such as readability/writability of a file descriptor or expiration of a timer.
- *Synchronous signal-handling*, which is the ability to defer signal-handling to a safe point in the event-handling loop.
- *Syncronous child cleanup*, which lets you defer calls to **wait** or **waitpid** to a safe point in the event-handling loop.

2 Overview

Figure 1 indicates the overall flow of programs using LibEvent.

1. Call **Event_CreateSelector** once to create an *Event Selector*. This is an object which manages event dispatch.
2. Open file descriptors as required, and call **Event_CreateHandler** to create *Event Handlers* for each descriptor of interest. You can call **Event_CreateTimerHandler** to create timers which are not associated with file descriptors.

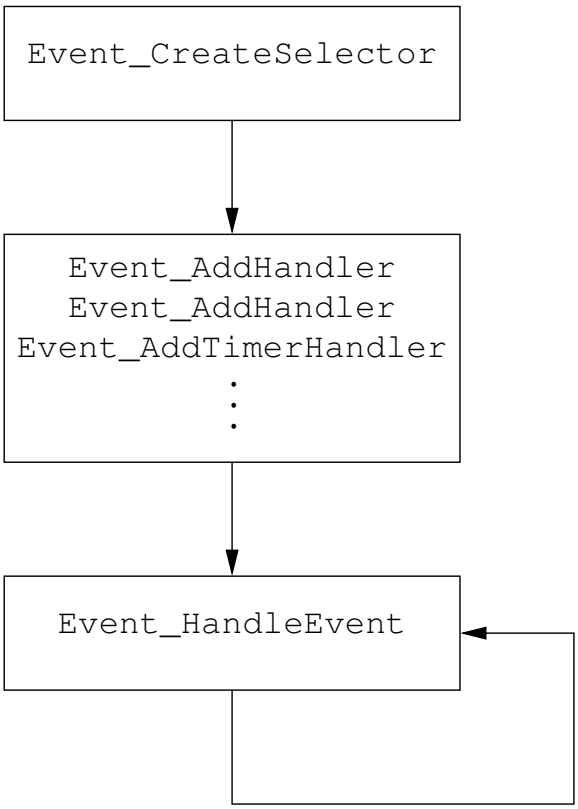

Figure 1: LibEvent Flow

3. Call **Event_HandleEvent** in a loop. Presumably, some event will cause the program to exit out of the infinite loop (unless the program is designed never to exit.)

To use LibEvent, you should `#include` the file `libevent/event.h`

3 Types

LibEvent defines the following types:

- *EventSelector* – a container object which manages event handlers.
- *EventHandler* – an object which triggers a callback function when an event occurs.
- *EventCallbackFunc* – a prototype for the callback function called by an *EventHandler*.

4 Basic Functions

This section describes the basic LibEvent functions. Each function is described in the following format:

---

type **name**(*type1* `arg1`, *type2* `arg2`)

Description: A brief description of the function. *type* is the type of the return value and **name** is the name of the function.

Returns: What the function returns

Arguments:

- `arg1` – A description of the first argument.
- `arg2` – A description of the second argument, etc.

4.1 Event Selector Creation and Destruction

EventSelector * **Event_CreateSelector**(*void*)

Description: Creates an *EventSelector* object and returns a pointer to it. An *EventSelector* is an object which keeps track of event handlers. You should treat it as an opaque type.

Returns: A pointer to the *EventSelector*, or NULL if out of memory.

Arguments:

None.

---

void **Event_DestroySelector**(*EventSelector* ***es**)

Description: Destroys an *EventSelector* and all associated event handlers.

Returns: Nothing.

Arguments:

- `es` – the *EventSelector* to destroy.

4.2 Event Handler Creation and Destruction

An *EventHandler* is an opaque object which contains information about an event. An event may be *triggered* by one or more of three things:

1. A file descriptor becomes readable. That is, **select** for readability would return.
2. A file descriptor becomes writeable.
3. A timeout elapses.

When an event triggers, it calls an event callback function. An event callback function looks like this:

---

void **functionName**(*EventSelector* *`es`,
int `fd`,
unsigned int `flags`,
void *`data`)

Description: Called when an event handler triggers.

Returns: Nothing

Arguments:

- `es` – the *EventSelector* to which the event handler belongs.
- `fd` – the file descriptor (if any) associated with the event.
- `flags` – a bitmask of one or more of the following values:
  - `EVENT_FLAG_READABLE` – the descriptor is readable.

  - `EVENT_FLAG_WRITEABLE` – the descriptor is writeable.
  - `EVENT_FLAG_TIMEOUT` – a timeout triggered.
- `data` – an opaque pointer which was passed into **Event_AddHandler**.

---

*EventHandler ** **Event_AddHandler**(*EventSelector **`es`,
int `fd`,
unsigned int `flags`,
EventCallbackFunc `fn`,
*void **`data`)

Description: Creates an *EventHandler* to handle an event.

Returns: An allocated *EventHandler*, or NULL if out of memory.

Arguments:

- `es` – the event selector.
- `fd` – the file descriptor to watch. `fd` must be a legal file descriptor for use inside **select**.
- `flags` – a bitmask whose value is one of `EVENT_FLAG_READABLE`, `EVENT_FLAG_WRITEABLE` or `EVENT_FLAG_READABLE | EVENT_FLAG_WRITEABLE`. `flags` specifies the condition(s) under which to trigger the event.
- `fn` – the callback function to invoke when the event triggers.
- `data` – a pointer which is passed unchanged as the last parameter of `fn` when the event triggers.

---

*EventHandler ** **Event_AddTimerHandler**(*EventSelector **`es`,
struct timeval `t`,
EventCallbackFunc `fn`,
*void **`data`)

Description: Creates an *EventHandler* to handle a timeout. After the timeout elapses, the callback function is called once only, and then the *EventHandler* is automatically destroyed.

Returns: An allocated *EventHandler*, or NULL if out of memory.

Arguments:

- `es` – the event selector.
- `t` – the time after which to trigger the event. `t` specifies how long *after* the current time to trigger the event.

- `fn` – the callback function to invoke when the event triggers. A timer handler function is always called with its `flags` set to `EVENT_FLAG_TIMER | EVENT_FLAG_TIMEOUT`.
- `data` – a pointer which is passed unchanged as the last parameter of `fn` when the event triggers.

---

*EventHandler ** **Event_AddHandlerWithTimeout**(*EventSelector **`es`,
int `fd`,
unsigned int `flags`,
struct timeval `t`,
EventCallbackFunc `fn`,
*void **`data`)

Description: Creates an *EventHandler* to handle an event. The event is called when a file descriptor is ready or a timeout elapses. This function may be viewed as a combination of **Event_AddHandler** and **Event_AddTimerHandler**.

Returns: An allocated *EventHandler*, or NULL if out of memory.

Arguments:

- `es` – the event selector.
- `fd` – the file descriptor to watch. `fd` must be a legal file descriptor for use inside **select**.
- `flags` – a bitmask whose value is one of `EVENT_FLAG_READABLE`, `EVENT_FLAG_WRITEABLE` or `EVENT_FLAG_READABLE | EVENT_FLAG_WRITEABLE`. `flags` specifies the condition(s) under which to trigger the event.
- `t` – the time after which to trigger the event. If the event is triggered because of a timeout, the callback function's `flags` has the `EVENT_FLAG_TIMEOUT` bit set.
- `fn` – the callback function to invoke when the event triggers.
- `data` – a pointer which is passed unchanged as the last parameter of `fn` when the event triggers.

---

int **Event_DelHandler**(*EventSelector **`es`,
*EventHandler **`eh`)

Description: Deletes an *EventHandler* and frees its memory. A handler may be deleted from inside a handler callback; LibEvent defers the actual deallocation of resources to a safe time.

Returns: 0 if the handler was found and deleted, non-zero otherwise. A non-zero return value indicates a critical internal error.

Arguments:

- `es` – the event selector which contains `eh`.
- `eh` – the event handler to delete.

4.3 Event Handler Access Functions

The functions in this section access or modify fields in the *EventHandler* structure. You should *never* access or modify fields in an *EventHandler* except with these functions.

---

void **Event_ChangeTimeout**(*EventHandler* *`eh`,
struct timeval `t`)

Description: Changes the timeout of `eh` to be `t` seconds from now. If `eh` was not created with **Event_AddTimerHandler** or **Event_AddHandlerWithTimeout**, then this function has no effect.

Returns: Nothing

Arguments:

- `eh` – the *EventHandler* whose timeout is to be modified.
- `t` – new value of timeout, relative to current time.

EventCallbackFunc **Event_GetCallback**(*EventHandler* *`eh`)

Description: Returns the callback function associated with `eh`.

Returns: A pointer to the callback function associated with `eh`.

Arguments:

- `eh` – the *EventHandler* whose callback pointer is desired.

---

void * **Event_GetData**(*EventHandler* *`eh`)

Description: Returns the data associated with `eh` (the `data` argument to the ...AddHandler... function.)

Returns: The data pointer associated with `eh`.

Arguments:

- `eh` – the *EventHandler* whose data pointer is desired.

---

void **Event_SetCallbackAndData**(*EventHandler* *`eh`,
EventCallbackFunc `fn`,
void *`data`)

Description: Sets the callback function and data associated with `eh`.

Returns: Nothing.

Arguments:

- `eh` – the *EventHandler* whose callback function and data pointer are to be set.
- `fn` – the new value for the callback function.
- `data` – the new value for the data pointer.

5 Signal Handling

In UNIX, signals can arrive asynchronously, and a signal-handler function may be called at an unsafe time, leading to race conditions. LibEvent has a mechanism to call a handler function during **Event_HandleEvent** so that the handler is dispatched just like any other event handler. In this way, the signal handler knows that it is safe to access shared data without interference from another thread of control.

LibEvent implements this *synchronous signal handling* by setting up a UNIX pipe, and writing to the write-end inside the asynchronous handler. The read end then becomes ready for reading, and triggers a normal event. LibEvent encapsulates all the details for you in two functions.

---

int **Event_HandleSignal**(*EventSelector* *`es`,
int `sig`,
*void (**`handler`*)(int* `sig`*)*)

Description: Arranges for the function `handler` to be called when signal `sig` is received. `sig` is typically a constant from `signal.h`, such as `SIGHUP`, `SIGINT`, etc. The `handler` function is not called in the context of a UNIX signal handler; rather, it is called soon after the signal has been received as part of the normal **Event_HandleEvent** loop.

As a side-effect of calling this function, a UNIX signal handler is established for `sig`. Any existing signal disposition is forgotten. If `sig` is `SIGCHLD`, then the `SA_NOCLDSTOP` flag is set in the `struct sigaction` passed to the low-level **sigaction** function.

Returns: 0 on success; -1 on failure. Failure is usually due to a UNIX system call failing or a lack of memory.

Arguments:

- `es` – the event selector.
- `sig` – the signal we wish to handle.
- `handler` – the function to call. It is passed a single argument—the signal which is being handled.

---

int **Event_HandleChildExit**(*EventSelector* *`es`,
pid_t `pid`,
*void (**`handler`*)(pid_t* `pid`, *int* `status`, *void **`data`*)*,
*void **`data`)

Description: Arranges for `handler` to be called when the child process with process-ID `pid` exits. `pid` must be the return value of a successful call to **fork**.

When the process with process-ID `pid` exits, LibEvent catches the `SIGCHILD` signal and at some point in the event-handling loop, calls `handler` with three arguments: `pid` is the process-ID of the process which terminated. `status` is the exit status as returned by the **waitpid** system call. And `data` is passed unchanged from the call to **Event_HandleChildExit**.

Returns: 0 on success; -1 on failure. Failure is the result of lack of memory or the failure of a UNIX system call.

Arguments:

- `es` – the event selector.
- `pid` – process-ID of the child process.
- `handler` – the function to call when the process exits.
- `data` – a pointer which is passed unchanged to `handler` when the process exits.

6 Stream-Oriented Functions

The functions presented in the previous sections are appropriate for simple events, especially those associated with datagram sockets. A higher level of abstraction is required for stream-oriented descriptors. It would be nice for LibEvent to invoke a callback function when a certain number of bytes or a

specific delimiter have been read from a stream, or when an entire buffer's worth of data has been written to a stream.

The functions in this section all (unfortunately) have the string `Tcp` in their names, because they were originally used with TCP sockets. However, they may be used with any stream-oriented sockets, including UNIX-domain sockets.

All of the stream-oriented functions are built on the simpler event functions described previously. They simply add an extra layer of convenience. To use the stream-oriented functions, `#include` the file `libevent/event_tcp.h`.

7 Stream-Oriented Data Types

The stream-oriented functions use the following publicly-accessible type:

- *EventTcpState* – an opaque object which records the state of stream-oriented event handlers.

8 Stream-Oriented Functions

The stream-oriented functions may be broken into two main groups: Connection establishment, and data transfer.

8.1 Connection Establishment

EventHandler * **EventTcp_CreateAcceptor**(*EventSelector* *`es`,
int `fd`,
EventTcpAcceptFunc `f`)

Description: Creates an event handler to accept incoming connections on the listening descriptor `fd`. Each time an incomming connection is accepted, the function `f` is called.

Returns: An *EventHandler* on success; NULL on failure.

Arguments:

- `es` – the event selector.
- `fd` – a listening socket (i.e., one for which the **listen**(2) system call has been called.)
- `f` – a function which is called each time an incoming connection is accepted. The function `f` should look like this:

  void **f**(*EventSelector* *`es`, *int* `fd`)

  In this case, `es` is the *EventSelector*, and `fd` is the new file descriptor returned by **accept**(2).

---

void **EventTcp_Connect**(*EventSelector* *`es`,
int `fd`,
struct sockaddr const *`addr`,
socklen_t `addrlen`,
EventTcpConnectFunc `f`,
int `timeout`,
void *`data`)

Description: Attempts to connect the socket `fd` to `addr` using the **connect**(2) system call.

Returns: Nothing. See below for error-handling notes.

Arguments:

- `es` – the event selector.
- `fd` – a socket which is suitable for passing to **connect**(2).
- `addr` – the server address to connect to.
- `addrlen` – the length of the server address. The three parameters `fd`, `addr` and `addrlen` are passed directly to **connect**(2).
- `f` – A function which is called when the connection succeeds (or if an error occurs.) The function `f` looks like this:

  void **f**(*EventSelector* *`es`, *int* `fd`, *int* `flag`, *void* *`data`)

  The parameters of `f` have the following meaning:

  - `es` – the event selector.
  - `fd` – the descriptor.
  - `flag` – a flag indicating what happened. It may contain one of the following values:
    - `EVENT_TCP_FLAG_IOERROR` – the **connect** system call failed.
    - `EVENT_TCP_FLAG_COMPLETE` – the **connect** system call succeeded and the descriptor is now connected.
    - `EVENT_TCP_FLAG_TIMEOUT` – the **connect** system call did not complete within the specified timeout.
  - `data` – a copy of the `data` given to **EventTcp_Connect**.
- `timeout` – a timeout value in seconds. If **connect** does not complete within `timeout` seconds, the `f` is called with a flag of `EVENT_TCP_FLAG_TIMEOUT`.
- `data` – an opaque pointer passed unchanged to `f`.

8.2 Data Transfer

There are two stream-oriented functions for data transfer: One for reading and one for writing.

EventTcpState * **EventTcp_ReadBuf**(*EventSelector* *`es`,
int `fd`,
int `len`,
int `delim`,
EventTcpIOFinishedFunc `f`,
int `timeout`,
void *`data`)

Description: Arranges events to read up to `len` characters from the file descriptor `fd`. If `delim` is non-negative, reading stops when the characters `delim` is encountered. After `len` characters have been read (or `delim` has been encountered), or after `timeout` seconds have elapsed, the function `f` is called.

Returns: An *EventTcpState* object on success; NULL on failure. Failure is usually due to failure of a UNIX system call or lack of memory.

Arguments:

- `es` – the event selector.
- `fd` – the descriptor to read from.
- `len` – the maximum number of bytes to read.
- `delim` – if negative, reading continues until exactly `len` bytes have been read or the operation times out. If non-negative, reading stops when `len` bytes have been read or the characters `delim` is encountered, whichever comes first. Note that supplying a non-negative `delim` causes LibEvent to invoke the **read**(2) system call for *each character*; if you are expecting large amounts of data before the delimiter, this could be inefficient.
- `f` – a function which is called when reading has finished, an error occurs, or the operation times out. The function `f` looks like this:

  void **f**(*EventSelector* *`es`, *int* `fd`, *char* *`int buf`, *int* `len`, *int* `flag`, *void* *`data`)

  The arguments passed to `f` are:

  - `es` – the event selector.
  - `fd` – the file descriptor that was passed to **EventTcp_ReadBuf**. If no more activity on `fd` is required, then you should **close** it inside `f`.

  - `buf` – a dynamically-allocated buffer holding the data which were read from `fd`. *Do not* free this buffer; LibEvent will take care of it. *Do not* store the pointer value; if you need a copy of the data, you must copy the whole buffer.
  - `len` – the number of bytes actually read from `fd`.
  - `flag` – a flag indicating what happened. It can have one of four values:
    * `EVENT_TCP_FLAG_COMPLETE` – the operation completed successfully.
    * `EVENT_TCP_FLAG_IOERROR` – an error occurred during a **read**(2) or some other system call.
    * `EVENT_TCP_FLAG_EOF` – EOF was detected before all bytes were read. Nevertheless, `len` and `buf` have valid contents.
    * `EVENT_TCP_FLAG_TIMEOUT` – the operation timed out before all bytes were read. Nevertheless, `len` and `buf` have valid contents.
  - `data` – a copy of the `data` pointer passed to **EventTcp_ReadBuf**.

- `timeout` – if positive, LibEvent times the operation out after `timeout` seconds.

- `data` – an opaque pointer which is passed as-is to `f`.

EventTcpState * **EventTcp_WriteBuf**(*EventSelector* *`es`,
int `fd`,
char *`buf`,
int `len`,
EventTcpIOFinishedFunc `f`,
int `timeout`,
void *`data`)

Description: Arranges events to write `len` characters from the buffer `buf` to the file descriptor `fd`. After `len` characters have been written, an error occurs, or `timeout` seconds have elapsed, the function `f` is called.

Returns: An *EventTcpState* object on success; NULL on failure. Failure is usually due to failure of a UNIX system call or lack of memory.

Arguments:

- `es` – the event selector.

- `fd` – the descriptor to write to.

- `buf` – buffer containing characters to write. **EventTcp_WriteBuf** allocates its own private copy of the buffer; you may free or reuse the buffer once **EventTcp_WriteBuf** returns.

- `len` – the number of bytes to write.
- `f` – a function which is called when reading has finished, an error occurs, or the operation times out. The function `f` is as described in **EventTcp_ReadBuf**. As a special case, you may supply NULL as the value for `f`. In this case, **EventTcp_WriteBuf** calls **close**(2) on the descriptor `fd` once writing has finished or timed out, or if an error occurs.
- `timeout` – if positive, LibEvent times the operation out after `timeout` seconds.
- `data` – an opaque pointer which is passed as-is to `f`.