NAME | SYNOPSIS | DESCRIPTION | EVENT PRIORITIES | THREAD SAFE EVENTS | BUFFERED EVENTS | NON-BLOCKING HTTP SUPPORT | ADDITIONAL NOTES | RETURN VALUES | SEE ALSO | HISTORY | AUTHORS | BUGS | COLOPHON

Search online pages

EVENT(3)   BSD Library Functions Manual   EVENT(3)

NAME top

event_init, event_dispatch, event_loop, event_loopexit, event_loopbreak, event_set, event_base_dispatch, event_base_loop, event_base_loopexit, event_base_loopbreak, event_base_set, event_base_free, event_add, event_del, event_once, event_base_once, event_pending, event_initialized, event_priority_init, event_priority_set, evtimer_set, evtimer_add, evtimer_del, evtimer_pending, evtimer_initialized, signal_set, signal_add, signal_del, signal_pending, signal_initialized, bufferevent_new, bufferevent_free, bufferevent_write, bufferevent_write_buffer, bufferevent_read, bufferevent_enable, bufferevent_disable, bufferevent_settimeout, bufferevent_base_set, evbuffer_new, evbuffer_free, evbuffer_add, evbuffer_add_buffer, evbuffer_add_printf, evbuffer_add_vprintf, evbuffer_drain, evbuffer_write, evbuffer_read, evbuffer_find, evbuffer_readline, evhttp_new, evhttp_bind_socket, evhttp_free — execute a function when a specific event occurs

SYNOPSIS top

```
#include <sys/time.h>
#include <event.h>

struct event_base *
event_init(void);

int
event_dispatch(void);

int
```

```
event_loop(int flags);

int
event_loopexit(struct timeval *tv);

int
event_loopbreak(void);

void
event_set(struct event *ev, int fd, short event,
    void (*fn)(int, short, void *), void *arg);

int
event_base_dispatch(struct event_base *base);

int
event_base_loop(struct event_base *base, int flags);

int
event_base_loopexit(struct event_base *base, struct timeval *tv);

int
event_base_loopbreak(struct event_base *base);

int
event_base_set(struct event_base *base, struct event *);

void
event_base_free(struct event_base *base);

int
event_add(struct event *ev, struct timeval *tv);

int
event_del(struct event *ev);

int
event_once(int fd, short event, void (*fn)(int, short, void *),
    void *arg, struct timeval *tv);

int
```

```
event_base_once(struct event_base *base, int fd, short event,
    void (*fn)(int, short, void *), void *arg, struct timeval *tv);

int
event_pending(struct event *ev, short event, struct timeval *tv);

int
event_initialized(struct event *ev);

int
event_priority_init(int npriorities);

int
event_priority_set(struct event *ev, int priority);

void
evtimer_set(struct event *ev, void (*fn)(int, short, void *),
    void *arg);

void
evtimer_add(struct event *ev, struct timeval *);

void
evtimer_del(struct event *ev);

int
evtimer_pending(struct event *ev, struct timeval *tv);

int
evtimer_initialized(struct event *ev);

void
signal_set(struct event *ev, int signal,
    void (*fn)(int, short, void *), void *arg);

void
signal_add(struct event *ev, struct timeval *);

void
signal_del(struct event *ev);
```

```
int
signal_pending(struct event *ev, struct timeval *tv);

int
signal_initialized(struct event *ev);

struct bufferevent *
bufferevent_new(int fd, evbuffercb readcb, evbuffercb writecb,
    everrorcb, void *cbarg);

void
bufferevent_free(struct bufferevent *bufev);

int
bufferevent_write(struct bufferevent *bufev, void *data, size_t size);

int
bufferevent_write_buffer(struct bufferevent *bufev,
    struct evbuffer *buf);

size_t
bufferevent_read(struct bufferevent *bufev, void *data, size_t size);

int
bufferevent_enable(struct bufferevent *bufev, short event);

int
bufferevent_disable(struct bufferevent *bufev, short event);

void
bufferevent_settimeout(struct bufferevent *bufev, int timeout_read,
    int timeout_write);

int
bufferevent_base_set(struct event_base *base,
    struct bufferevent *bufev);

struct evbuffer *
evbuffer_new(void);

void
```

```
evbuffer_free(struct evbuffer *buf);

int
evbuffer_add(struct evbuffer *buf, const void *data, size_t size);

int
evbuffer_add_buffer(struct evbuffer *dst, struct evbuffer *src);

int
evbuffer_add_printf(struct evbuffer *buf, const char *fmt, ...);

int
evbuffer_add_vprintf(struct evbuffer *buf, const char *fmt,
    va_list ap);

void
evbuffer_drain(struct evbuffer *buf, size_t size);

int
evbuffer_write(struct evbuffer *buf, int fd);

int
evbuffer_read(struct evbuffer *buf, int fd, int size);

unsigned char *
evbuffer_find(struct evbuffer *buf, const unsigned char *data,
    size_t size);

char *
evbuffer_readline(struct evbuffer *buf);

struct evhttp *
evhttp_new(struct event_base *base);

int
evhttp_bind_socket(struct evhttp *http, const char *address,
    unsigned short port);

void
evhttp_free(struct evhttp *http);
```

```
int (*event_sigcb)(void);

volatile sig_atomic_t event_gotsig;
```

DESCRIPTION

top

The event API provides a mechanism to execute a function when a specific event on a file descriptor occurs or after a given time has passed.

The event API needs to be initialized with event_init() before it can be used.

In order to process events, an application needs to call event_dispatch(). This function only returns on error, and should replace the event core of the application program.

The function event_set() prepares the event structure ev to be used in future calls to event_add() and event_del(). The event will be prepared to call the function specified by the fn argument with an int argument indicating the file descriptor, a short argument indicating the type of event, and a void * argument given in the arg argument. The fd indicates the file descriptor that should be monitored for events. The events can be either EV_READ, EV_WRITE, or both, indicating that an application can read or write from the file descriptor respectively without blocking.

The function fn will be called with the file descriptor that triggered the event and the type of event which will be either EV_TIMEOUT, EV_SIGNAL, EV_READ, or EV_WRITE. Additionally, an event which has registered interest in more than one of the preceeding events, via bitwise-OR to event_set(), can provide its callback function with a bitwise-OR of more than one triggered event. The additional flag EV_PERSIST makes an event_add() persistent until event_del() has been called.

Once initialized, the ev structure can be used repeatedly with event_add() and event_del() and does not need to be reinitialized unless the function called and/or the argument to it are to be changed. However, when an ev structure has been added to libevent using event_add() the structure must persist until the event occurs (assuming EV_PERSIST is not set) or is removed using event_del(). You may not reuse the same ev structure for multiple monitored descriptors; each descriptor needs its own ev.

The function event_add() schedules the execution of the ev event when the event specified in event_set() occurs or in at least the time specified in the tv. If tv is NULL, no timeout occurs and the function will only be called if a matching event occurs on the file descriptor. The event in the ev argument must be already initialized by event_set() and may not be used in calls to event_set() until it has timed out or been removed with event_del(). If the event in the ev argument already has a scheduled timeout, the old timeout will be replaced by the new one.

The function event_del() will cancel the event in the argument ev. If the event has already executed or has never been added the call will have no effect.

The functions evtimer_set(), evtimer_add(), evtimer_del(), evtimer_initialized(), and evtimer_pending() are abbreviations for common situations where only a timeout is required. The file descriptor passed will be -1, and the event type will be EV_TIMEOUT.

The functions signal_set(), signal_add(), signal_del(), signal_initialized(), and signal_pending() are abbreviations. The event type will be a persistent EV_SIGNAL. That means signal_set() adds EV_PERSIST.

In order to avoid races in signal handlers, the event API provides two variables: event_sigcb and event_gotsig. A signal handler sets event_gotsig to indicate that a signal has been received. The application sets event_sigcb to a callback function. After the signal handler sets event_gotsig, event_dispatch will execute the callback function to process received signals. The callback returns 1 when no events are registered any more. It can return -1 to indicate an error to the event library, causing event_dispatch() to terminate with errno set to EINTR.

The function event_once() is similar to event_set(). However, it schedules a callback to be called exactly once and does not require the caller to prepare an event structure. This function supports EV_TIMEOUT, EV_READ, and EV_WRITE.

The event_pending() function can be used to check if the event specified by event is pending to run. If EV_TIMEOUT was specified and tv is not NULL, the expiration time of the event will be returned in tv.

The event_initialized() macro can be used to check if an event has been initialized.

The event_loop function provides an interface for single pass execution of pending events. The flags EVLOOP_ONCE and EVLOOP_NONBLOCK are recognized. The event_loopexit function exits from the event loop. The next event_loop() iteration after the given timer expires will complete normally (handling all queued events) then exit without blocking for events again. Subsequent invocations of event_loop() will proceed normally. The event_loopbreak function exits from the event loop immediately. event_loop() will abort after the next event is completed; event_loopbreak() is typically invoked from this event's callback. This behavior is analogous to the "break;" statement. Subsequent invocations of event_loop() will proceed normally.

It is the responsibility of the caller to provide these functions with pre-allocated event structures.

EVENT PRIORITIES

top

By default libevent schedules all active events with the same priority. However, sometimes it is desirable to process some events with a higher priority than others. For that reason, libevent supports strict priority queues. Active events with a lower priority are always processed before events with a higher priority.

The number of different priorities can be set initially with the event_priority_init() function. This function should be called before the first call to event_dispatch(). The event_priority_set() function can be used to assign a priority to an event. By default, libevent assigns the middle priority to all events unless their priority is explicitly set.

THREAD SAFE EVENTS

top

Libevent has experimental support for thread-safe events. When initializing the library via event_init(), an event base is returned. This event base can be used in conjunction with calls to event_base_set(), event_base_dispatch(), event_base_loop(), event_base_loopexit(), bufferevent_base_set() and event_base_free(). event_base_set() should be called after preparing an event with event_set(), as event_set() assigns the provided event to the most recently created event base. bufferevent_base_set() should be called after preparing a bufferevent with bufferevent_new(). event_base_free() should be used to free memory associated with the event base when it is no longer needed.

BUFFERED EVENTS

top

libevent provides an abstraction on top of the regular event callbacks. This abstraction is called a buffered event. A buffered event provides input and output buffers that get filled and drained automatically. The user of a buffered event no longer deals directly with the IO, but instead is reading from input and writing to output buffers.

A new bufferevent is created by bufferevent_new(). The parameter fd specifies the file descriptor from which data is read and written to. This file descriptor is not allowed to be a pipe(2). The next three parameters are callbacks. The read and write callback have the following form: void (*cb)(struct bufferevent *bufev, void *arg). The error callback has the following form: void (*cb)(struct bufferevent *bufev, short what, void *arg). The argument is specified by the fourth parameter cbarg. A bufferevent struct pointer is returned on success, NULL on error. Both the read and the write callback may be NULL. The error callback has to be always provided.

Once initialized, the bufferevent structure can be used repeatedly with bufferevent_enable() and bufferevent_disable(). The flags parameter can be a combination of EV_READ and EV_WRITE. When read enabled the bufferevent will try to read from the file descriptor and call the read callback. The write callback is executed whenever the output buffer is drained below the write low watermark, which is 0 by default.

The bufferevent_write() function can be used to write data to the file descriptor. The data is appended to the output buffer and written to the descriptor automatically as it becomes available for writing. bufferevent_write() returns 0 on success or -1 on failure. The bufferevent_read() function is used to read data from the input buffer, returning the amount of data read.

If multiple bases are in use, bufferevent_base_set() must be called before enabling the bufferevent for the first time.

NON-BLOCKING HTTP SUPPORT

top

libevent provides a very thin HTTP layer that can be used both to host an HTTP server and also to make HTTP requests. An HTTP server can be created by calling evhttp_new(). It can be bound to any port and address with the evhttp_bind_socket() function. When the HTTP server is no longer used, it can be freed via evhttp_free().

To be notified of HTTP requests, a user needs to register callbacks with the HTTP server. This can be done by calling evhttp_set_cb(). The second argument is the URI for which a callback is being registered. The corresponding callback will receive an struct evhttp_request object that contains all information about the request.

This section does not document all the possible function calls; please check event.h for the public interfaces.

ADDITIONAL NOTES

top

It is possible to disable support for epoll, kqueue, devpoll, poll or select by setting the environment variable EVENT_NOEPOLL, EVENT_NOKQUEUE, EVENT_NODEVPOLL, EVENT_NOPOLL or EVENT_NOSELECT, respectively. By setting the environment variable EVENT_SHOW_METHOD, libevent displays the kernel notification method that it uses.

RETURN VALUES

top

Upon successful completion event_add() and event_del() return 0. Otherwise, -1 is returned and the global variable errno is set to indicate the error.

SEE ALSO top

kqueue(2), poll(2), select(2), evdns(3), timeout(9)

HISTORY top

The event API manpage is based on the timeout(9) manpage by Artur Grabowski. The port of libevent to Windows is due to Michael A. Davis. Support for real-time signals is due to Taral.

AUTHORS top

The event library was written by Niels Provos.

BUGS top

This documentation is neither complete nor authoritative. If you are in doubt about the usage of this API then check the source code to find out how it works, write up the missing piece of documentation and send it to me for inclusion in this man page.

COLOPHON top

This page is part of the libevent (an event notification library) project. Information about the project can be found at http://libevent.org/. If you have a bug report for this manual page,

see ⟨http://sourceforge.net/p/levent/bugs/⟩. This page was obtained

from the project's upstream Git repository ⟨https://github.com/libevent/libevent.git⟩ on 2018-10-29. (At that time, the date of the most recent commit that was found in the repository was 2018-10-28.) If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is not part of the original manual page), send a mail to man-pages@man7.org