Lecture 28 — Event-Driven I/O with libevent

Prepared by Jeff Zarnett, taught by Seyed Majid Zahedi
jzarnett@uwaterloo.ca, smzahedi@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

Events!

Using libevent

The libevent library is meant for high performance applications and scalable network servers.

Instead of focusing now on whether an operation is blocking or not, we'll try to think about I/O as events: when something happens, take some action.

Let's Get Ready

The library supports a lot of different configuration options.

But the most important setup thing is that we need to configure it to use the pthreads library and functions.

```
int evthread_use_pthreads( )
```

Zero Wing...

Each event is associated with an `event_base` structure.

We might need multiple bases for multiple thread operations.

Building a Base

```
struct event_base* event_base_new( ); /* Create with default settings */
struct event_base* event_base_new_with_config(
    const struct event_config* cfg ); /* Create with configuration */
struct event_config* event_config_new( );
void event_config_free( struct event_config* cfg );
```

The use of a configuration is optional.

To deallocate an event base, the (self-explanatory) function for that is:

```
void event_base_free( struct event_base* base );
```

Find Your Team's Capabilities

```
#include <stdlib.h>
#include <stdio.h>
#include <event2/event.h>

int main( int argc, char** argv ) {

    int i;
    const char **methods = event_get_supported_methods();
    printf("Starting␣Libevent␣%s.␣␣␣Available␣methods␣are:\n",
           event_get_version());
    for (i=0; methods[i] != NULL; ++i) {
        printf("␣␣␣␣%s\n", methods[i]);
    }
    return 0;
}
```

Ask ecetesla0

```
jzarnett@ecetesla0:~/ece252$ gcc -std=c99 -g -levent -o le1 le1.c
jzarnett@ecetesla0:~/ece252$ ./le1
Starting Libevent 2.1.8-stable.  Available methods are:
    epoll
    poll
    select
```

What You Can Do vs What You Will Do

```
#include <stdlib.h>
#include <stdio.h>
#include <event2/event.h>

int main( int argc, char** argv ) {
    struct event_base *base;
    enum event_method_feature f;

    base = event_base_new();
    if (!base) {
        puts("Couldn't get an event_base!");
    } else {
        printf("Using Libevent with backend method %s.",
                event_base_get_method(base));
        f = event_base_get_features(base);
        if ((f & EV_FEATURE_ET))
            printf("  Edge-triggered events are supported.");
        if ((f & EV_FEATURE_O1))
            printf("  O(1) event notification is supported.");
        if ((f & EV_FEATURE_FDS))
            printf("  All FD types are supported.");
        puts("");
    }
    return 0;
}
```

Well, we'll... just go with that then.

```
jzarnett@ecetesla0:~/ece252$ ./le2
Using Libevent with backend method epoll.
 Edge-triggered events are supported.
 O(1) event notification is supported.
```

Wait, we didn't learn about epoll... Do we need to?

Cancel Red Alert

No.

The great thing about libevent is that we don’t have to think about the details of the backend.

Keep Your Eyes Open

The goal is to watch for some events; for that we need a definition of an event.

An event happens on a file descriptor (as usual).

Event Lifecycle:

- Created
- Initialized
- Pending
- Non-Pending

```
typedef void (*event_callback_fn)( evutil_socket_t fd, short what, void* arg )
struct event* event_new( struct event_base* base, evutil_socket_t fd,
    short what, event_callback_fn cb, void* arg )
```

`event_callback_fn`: function signature definition for callback.

`base`: event base to use.

`fd`: file descriptor.

`what`: what we want to be notified of.

`arg`: user-defined argument.

Professor: there is a use for the short type in C

Students:

The what Parameter

```
#define EV_TIMEOUT      0x01
#define EV_READ         0x02
#define EV_WRITE        0x04
#define EV_SIGNAL       0x08
#define EV_PERSIST      0x10
#define EV_ET           0x20
```

Some require some explanation…

They can be combined with the bitwise-OR operator again.

Call Yourself?

If you want the event itself to be the `void*` argument passed to the callback function, that can be done here as well.

Normally this would not work, because the event doesn't exist yet.

But there's a workaround for that, a function `event_self_cbarg()` that does a little magic for you.

Destroying an Event

```
void event_free( struct event* event )
```

It is okay to call this even on an event that is pending or active.

Ready to Watch? Done Watching?

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

But did we forget something?

Oh yeah… starting the events!

Dispatching Events

There are two ways to dispatch events, the simple way and the hard way:

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

The easy way is the same as the hard way with no flags set.

```
#define EVLOOP_ONCE                0x01
#define EVLOOP_NONBLOCK            0x02
#define EVLOOP_NO_EXIT_ON_EMPTY    0x04
```

No-exit means don't break the loop when no more events pending/active.

When I Say...


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

Start Events and Watch For Them

```
void cb_func( evutil_socket_t fd, short what, void *arg ) {
        const char *data = arg;
        printf("Got an event on socket %d:%s%s%s%s [%s]",
            (int) fd,
            (what&EV_TIMEOUT) ? " timeout" : "",
            (what&EV_READ)    ? " read" : "",
            (what&EV_WRITE)   ? " write" : "",
            (what&EV_SIGNAL)  ? " signal" : "",
            data);
}
```

Start Events and Watch For Them

```
void main_loop( evutil_socket_t fd1, evutil_socket_t fd2 ){
        struct event *ev1, *ev2;
        struct timeval five_seconds = {5,0};
        struct event_base *base = event_base_new();

        /* The caller has already set up fd1, fd2 somehow,
           and make them nonblocking. */

        ev1 = event_new(base, fd1, EV_TIMEOUT|EV_READ|EV_PERSIST, cb_func,
            (char*)"Reading event");
        ev2 = event_new(base, fd2, EV_WRITE|EV_PERSIST, cb_func,
            (char*)"Writing event");

        event_add(ev1, &five_seconds);
        event_add(ev2, NULL);
        event_base_dispatch(base);
}
```

It's Clean-Up Time

Finally, libevent has some global structures that are initialized once.

When we're all completely done with everything:

```
void libevent_global_shutdown( )
```

This does not deallocate anything that was the return value of a libevent function.

Buffering...

We might want to wait until we have a significant chunk of data before we're ready to process it.

Buffering...

It's better to have the event happen when a condition is fulfilled, such as having enough data available.

The library does support this: `bufferevents`!

A normal callback is triggered when the underlying transport (e.g., socket) is ready to be read or written.

A buffer event takes place when enough data has been read or written.

Buffer events really only work for TCP communication.

Always Two, There Are

Each buffer event has two buffers: the input and output buffer.

There are also two callbacks, a read and a write callback.

There are defaults; these can be overridden.

Watermarks

Every buffer event has four “watermarks”:

- Read low-water mark
- Read high-water mark
- Write low-water mark
- Write high-water mark

Make Me a Buffer Event

```
struct bufferevent* bufferevent_socket_new( struct event_base* base,
    evutil_socket_t fd, enum bufferevent_options options );
```

`base`: the event base the buffer event belongs to.

`fd`: file descriptor

`options`: there are several, but we care about `BEV_OPT_CLOSE_ON_FREE`, `BEV_OPT_THREADSAFE`

```
void bufferevent_free( struct bufferevent* bev );
```

Deallocation is straightforward…

Callbacks

```
typedef void (*bufferevent_data_cb)(struct bufferevent* bev, void* ctx);
typedef void (*bufferevent_event_cb)(struct bufferevent* bev,
     short events, void* ctx);
```

Return type is void.

`bev`: buffer event in question.

`ctx`: user-provided context.

`what`: same as before, what we are interested in,.

Set up Buffer Event Callbacks

```
void bufferevent_setcb(struct bufferevent* bufev,
    bufferevent_data_cb readcb, bufferevent_data_cb writecb,
    bufferevent_event_cb eventcb, void* cbarg);
```

`bufev`: buffer event in question

`readcb`: read callback, `NULL` if not desired.

`writecb`: write callback; `NULL` if not desired.

`eventcb`: event callback; `NULL` if not desired.

`cbarg`: the user-supplied argument.

Now We're Ready...

We're ready, but have not yet actually created the event.

```
int bufferevent_socket_connect( struct bufferevent* bev,
    struct sockaddr* address, int addrlen );
```

bev: buffer event in question.

address: address to connect to.

addrlen: size of the address structure.

Put It All Together: A Silly Callback

```c
#include <event2/event.h>
#include <event2/bufferevent.h>
#include <sys/socket.h>
#include <string.h>

void eventcb(struct bufferevent *bev, short events, void *ptr) {
    if (events & BEV_EVENT_CONNECTED) {
        /* We're connected to 127.0.0.1:8080.   Ordinarily we'd do
           something here, like start reading or writing. */
    } else if (events & BEV_EVENT_ERROR) {
        /* An error occured while connecting. */
    }
}
```

Put It All Together: Main Loop

```
int main_loop( ) {
    struct event_base *base;
    struct bufferevent *bev;
    struct sockaddr_in sin;

    base = event_base_new();

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl(0x7f000001); /* 127.0.0.1 */
    sin.sin_port = htons(8080); /* Port 8080 */

    bev = bufferevent_socket_new(base, -1, BEV_OPT_CLOSE_ON_FREE);

    bufferevent_setcb(bev, NULL, NULL, eventcb, NULL);

    if (bufferevent_socket_connect(bev,
        (struct sockaddr *)&sin, sizeof(sin)) < 0) {
        /* Error starting connection */
        bufferevent_free(bev);
        return -1;
    }

    event_base_dispatch(base);
    return 0;
}
```

This Is Just The Beginning

By no means have we covered every possible option or tool in the libevent library.

It’s just one of the many ways we have seen for how to do asynchronous I/O…

Sunset in Starling City...