ECE 252: Systems Programming and Concurrency Spring 2019

Lecture 32 — Event-Driven I/O with libevent

Jeff Zarnett *2019-02-28*

Event-Driven I/O with libevent

We've now looked at a few different kinds of asynchronous I/O, including select/poll, cURL, and POSIX AIO. All of these seem to have some drawbacks, though. 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

Getting ready. The library supports a lot of different configuration options. Most of them are too advanced for consideration in this class, such as telling it how you want to handle fatal errors or override default memory allocation and deallocation behaviour. But the most important setup thing is that we need to configure it to use the pthreads library and functions. For this, the function is [Mat12]:

```
int evthread_use_pthreads( )
```

It returns 0 on success and -1 on failure. There exists a matching Windows function, but obviously that doesn't work if running under Linux... This function is a composite of several configurations about locks and condition variables and threads, so there's no need to use the smaller ones. Just use this top level function and we're ready to go.

All your base... Our goal is to handle events of some sort. Each event is associated with an `event_base` structure, which is a container for a set of events and has some associated configuration. If locking is enabled, it's safe to use in multiple threads, though its loop can only be in one thread. So we might need multiple bases for multiple thread operations.

We can choose what method we would like the event base to use check what events are ready. There are many, but the ones we are most familiar with are already here: select and poll. Other options include epoll, kqueue, dnvpoll, evport, win32 [Mat12].

To create an event base, we can do it in one of two ways:

```
struct event_base* event_base_new( ); /* Create an event_base 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 );
```

Like with pthreads, the use of a configuration structure is optional. If we do want one, there are separate functions to allocate and to clean them up. The options allow us to say that we would like to avoid certain methods for checking on events and configure arcane options. We'll say that for the purposes of this course that default options are fine.

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

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

Let's consider a brief example, then, in which we can ask on one of our servers what is available [Mat12]:

```
#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;
}
```

This produces as output on `ecetesla0` (at the time of writing):

```
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
```

If you actually want to know what method will be used internally [Mat12]:

```
#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;
}
```

Running this produces:

```
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? Well, no – the great thing about libevent is that we don't have to think about the details of the backend – whatever implementation is used is not really our concern.

Event notification. The goal is to watch for some events, and for that we need a definition of an event. An event happens on a file descriptor, as with all the other I/O methods we have discussed.

An event has a lifecycle: it is created; when it becomes associated with an event base it is said to be initialized. When you add it, it is pending, which is to say that we are waiting for the event to happen. If the event that we're looking for does occur when the event is pending, then a user-defined callback runs. If we don't want to wait for an event anymore we can use delete to make an event non-pending; a non-pending event can be re-added if desired [Mat12].

To create and destroy an event:

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

The first line in there is the definition that the callback function must take: void return type with three parameters. We should be familiar enough with this sort of thing by now.

The event creation function tells you everything you need to provide when creating an event. The definition of the structure is kind of unimportant to us because we don't access its internals directly. That's fine with us. Anyway, the first argument is the base that this event will be associated with. The second argument of type `evutil_socket_t` is really an `int` (in non-Windows system anyway) and we can just put in our regular file descriptor here. The `what` parameter is how we specify the kind of thing we want to be notified of (see below). And finally, the last two parameters are the callback function and our user-defined argument to that callback function.

About that `what` parameter: hey look, we found a use for `short`! Only took half your degree... Anyway, we use this to say what we're looking for, and it's some combination of the following options (as defined by the library):

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

Most of them are self explanatory, but some warrant a bit. The timeout flag is ignored when setting what you are interested in, but can be returned as a result. The "ET" one indicates that the event should be edge-triggered, which matters for read and writes (i.e., a change in readiness is reported rather than readiness). The "persist" one means that when an event is triggered it's automatically ready to trigger again immediately.

They can be combined, as we've seen previously for say, an open system call. If you're interested in watching for a read and a write, combine them using `|` (bitwise OR) just as we've done for other scenarios: `EV_READ | EV_WRITE`.

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 [Mat12].

Deallocating an event is pretty self explanatory as well. It is okay to call this even on an event that is pending or active, according to the library docs, because it checks this and will deactivate or make non-pending any event that gets freed using the `event_free()` call.

Alright, so we have created an event. When we're ready to start watching we can add the event, and when we're done watching we can remove it, with the following functions:

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

We will see an example soon, but first we are missing one more important thing: dispatching the events!

Let's get started. 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, so the hard way actually isn't all that hard. The flags are [Mat12]:

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

Once means that we just wait for some events to become active, then turn active events until there are no more, then return. The non-blocking option means we won't wait for events to trigger: we'll check if they're ready, run them if so, and skip them if not. And normally the behaviour is to exit from the loop when there are no more events pending or active; you can turn that off with the no-exit flag. If you plan to use that, you can manually exit the loop with one of:

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

The exit function says to stop after some time has elapsed; if any events are active we process them before exiting. The break function says stop as soon as we've finished the currently-being-processed active event but not any other ones that are active. Typically this would occur in a callback function of some sort, when we know we are done.

Now, that code example that was delayed [Mat12]:

```
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);
}

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);
}
```

Alright, this demonstrates how to create an event base, create some events, add them to the pending list, and begin dispatching (waiting for events to occur). Only one thread can be dispatching a given event base at a time, but this is why we can have multiple bases.

Cleaning up. Finally, libevent has some global structures that are initialized once. When we're all completely done with everything, there is a global cleanup function available [Mat12]:

```
void libevent_global_shutdown( )
```

This is not meant to deallocate anything that was the return value of a libevent function; those should be cleaned up normally. This should be the last function called, once all libevent stuff is done, otherwise unpredictable behaviour may occur.

Buffered events. For a slightly more advanced scenario, we might want to wait until we have a significant chunk of data before we're ready to process it. If we're receiving some data one character at a time, it's very wasteful to fire an event every time a whole character is received. Instead, 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, so we'll be back to sockets. This is, to some extent, expected, because our general view of files is that they're always "ready", even if a read takes time to be carried out.

Each buffer event has two buffers: the input and output buffer. There are also two callbacks, a read and a write callback. By default, the read callback happens whenever any data is received, and the write callback is called whenever enough data from the output buffer is sent [Mat12]. These can be overridden. But you can see why this would make some sense: whenever a chunk of data is received we put it in a buffer until all the data is there; we can send data when enough data has accumulated to be sent.

Every buffer event has four "watermarks", which are the levels that trigger some sort of happening [Mat12]:

- Read low-water mark: when a read occurs that results in the buffer being at least this full, the read callback triggers. By default, it is 0, so every chunk of data received starts the callback.
- Read high-water mark: when a read occurs that leaves the buffer at this level, no more data will be copied into the buffer until data has been taken out of the buffer to get it below the level. By default, this is unlimited so libevent will try to fill up the buffer as much as possible.
- Write low-water mark: when a write occurs to get to this level, the write callback occurs. This defaults to 0, so the write callback only runs if the output buffer is emptied.
- Write high-water mark: strangely, does not do what you expect. It can be used for some special meaning situations, though.

We'll take a look at a buffer events example for socket communication. This means we need to know how to create a buffer event, and how to use it on a socket.

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

typedef void (*bufferevent_data_cb)(struct bufferevent* bev, void* ctx);
typedef void (*bufferevent_event_cb)(struct bufferevent* bev, short events, void* ctx);
void bufferevent_setcb(struct bufferevent* bufev, bufferevent_data_cb readcb, bufferevent_data_cb writecb,
     bufferevent_event_cb eventcb, void* cbarg);

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

Creating a buffer event structure is fairly straightforward: specify which event base it should belong to, the file descriptor, and then you get to choose some options. There are multiple options, but the two we will consider are as follows. The first is `BEV_OPT_CLOSE_ON_FREE`: close the socket when the buffer event is deallocated. The other is `BEV_OPT_THREADSAFE`: configure the buffer event to use locks, so it can be accessed concurrently from multiple threads.

The deallocation function is totally self-explanatory.

The definitions of data callbacks and event callbacks are a bit more complex. The return type is void, and the arguments are the buffer event structure – sensible, and then there's a divergence. The data callback takes only some argument; it's the one provided by the user when setting up the callback (see below). The event callback has the extra parameter of events which works as it did earlier (where you say what events you are interested in).

Setting the callback is relatively self explanatory: specify the buffer event, the read callback (NULL if you don't want one, or to clear one), the write callback (again, NULL if you don't want one), and the event callback. Finally, there is the callback argument (of any type) and this argument is shared between all callbacks.

Then there is the connect function. This function takes the buffer event that you want to configure, and the address to connect to, as well as the size of the address. So that part is easy.

Let's consider an example from [Mat12] that shows how to use the buffer events:

```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. */
    }
}

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 sets up a callback that doesn't do much, but that's fine for the purpose of the example. We set up the socket connection stuff - the address and port. Then we create the buffer event, with the file descriptor as -1. This placeholder value means that at the time the buffer event is created, the file descriptor is not yet set. That makes sense, because the socket hasn't been opened yet. There's no file descriptor to set yet. Then we'll configure the callback, and launch the connect call. When the connection is established, the callback runs.

By no means have we covered every possible option or tool in the libevent library. It is quite powerful and quite effective at its job. But now we have an introduction to it, and we've seen a lot of ways to do asynchronous I/O.

References

[Mat12] Nick Mathewson. Fast portable non-blocking network programming with libevent, 2012. Online; accessed 22-February-2019. URL: `http://www.wangafu.net/~nickm/libevent-book/`.