

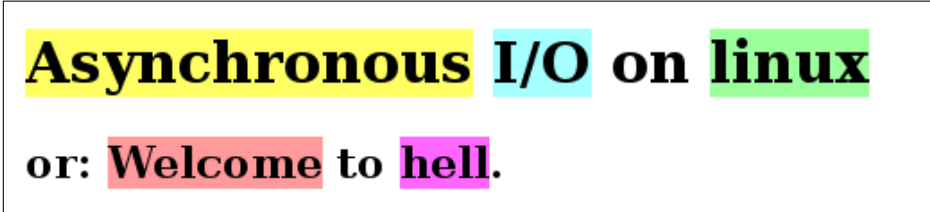
## Lecture 8 — Asynchronous I/O, epoll, select

Patrick Lam

2018-12-04

## Asynchronous/non-blocking I/O

Let's start with some juicy quotes.



**Asynchronous I/O on linux**  
**or: Welcome to hell.**

(mirrored at [compgeom.com/~piyush/teach/4531\\_06/project/hell.html](http://compgeom.com/~piyush/teach/4531_06/project/hell.html))

“Asynchronous I/O, for example, is often infuriating.”

— Robert Love. *Linux System Programming*, 2nd ed, page 215.

To motivate the need for non-blocking I/O, consider some standard I/O code:

```
fd = open(...);  
read(...);  
close(fd);
```

This isn't very performant. The problem is that the `read` call will *block*. So, your program doesn't get to use the zillions of CPU cycles that are happening while the I/O operation is occurring.

**As seen previously: threads.** That can be fine if you have some other code running to do work—for instance, other threads do a good job mitigating the I/O latency, perhaps doing I/O themselves. But maybe you would rather not use threads. Why not?

- potential race conditions;
- overhead due to per-thread stacks; or
- limitations due to maximum numbers of threads.

**Live coding example.** To illustrate the max-threads issue, let's write `threadbomb.c`, to explore how many simultaneous threads one could start on my computer.

**Non-blocking I/O.** The main point of this lecture, though, is non-blocking/asynchronous I/O. The simplest example:

```
fd = open(..., O_NONBLOCK);  
read(...); // returns instantly!  
close(fd);
```

In principle, the `read` call is supposed to return instantly, whether or not results are ready. That was easy!

Well, not so much. The `O_NONBLOCK` flag actually only has the desired behaviour on sockets. The semantics of `O_NONBLOCK` is for I/O calls to not block, in the sense that they should never wait for data while there is no data available.

Unfortunately, files always have data available. Under Linux, you'd have to use `aio` calls to be able to send requests to the I/O subsystem asynchronously and not, for instance, wait for the disk to spin up. We won't talk about them, but they operate along the same lines as what we will see. They just have a different API.

**Conceptual view: non-blocking I/O.** Fundamentally, there are two ways to find out whether I/O is ready to be queried: polling (under UNIX, implemented via `select`, `poll`, and `epoll`) and interrupts (under UNIX, signals).

We will describe `epoll`. It is the most modern and flexible interface. Unfortunately, I didn't realize that the obvious `curl` interface does not work with `epoll` but instead with `select`. There is different syntax but the ideas are the same.

The key idea is to give `epoll` a bunch of file descriptors and wait for events to happen. In particular:

- create an `epoll` instance (`epoll_create1`);
- populate it with file descriptors (`epoll_ctl`); and
- wait for events (`epoll_wait`).

Let's run through these steps in order.

**Creating an `epoll` instance.** Just use the API:

```
int epfd = epoll_create1(0);
```

The return value `epfd` is typed like a UNIX file descriptor—`int`—but doesn't represent any files; instead, use it as an identifier, to talk to `epoll`.

The parameter “0” represents the flags, but the only available flag is `EPOLL_CLOEXEC`. Not interesting to you.

**Populating the `epoll` instance.** Next, you'll want `epfd` to do something. The obvious thing is to add some `fd` to the set of descriptors watched by `epfd`:

```
struct epoll_event event;
int ret;
event.data.fd = fd;
event.events = EPOLLIN | EPOLLOUT;
ret = epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &event);
```

You can also use `epoll_ctl` to modify and delete descriptors from `epfd`; read the manpage to find out how.

**Waiting on an `epoll` instance.** Having completed the setup, we're ready to wait for events on any file descriptor in `epfd`.

```
#define MAX_EVENTS 64

struct epoll_event events[MAX_EVENTS];
int nr_events;

nr_events = epoll_wait(epfd, events, MAX_EVENTS, -1);
```

The given `-1` parameter means to wait potentially forever; otherwise, the parameter indicates the number of milliseconds to wait. (It is therefore “easy” to sleep for some number of milliseconds by starting an `epfd` and using `epoll_wait`; takes two function calls instead of one, but allows sub-second latency.)

Upon return from `epoll_wait`, we know that we have `nr_events` events ready.

## Level-Triggered and Edge-Triggered Events

One relevant concept for these polling APIs is the concept of *level-triggered* versus *edge-triggered*. The default `epoll` behaviour is level-triggered: it returns whenever data is ready. One can also specify (via `epoll_ctl`) edge-triggered behaviour: return whenever there is a change in readiness.

One would think that level-triggered mode would return from `read` whenever data was available, while edge-triggered mode would return from `read` whenever new data came in. Level-triggered does behave as one would guess: if there is data available, `read()` returns the data. However, edge-triggered mode returns whenever the state-of-readiness of the socket changes (from no-data-available to data-available). Play with it and get a sense for how it works.

Good question to think about: when is it appropriate to choose one or the other?

## Asynchronous I/O

As mentioned above, the POSIX standard defines `aio` calls. Unlike just giving the `O_NONBLOCK` flag, using `aio` works for disk as well as sockets.

**Key idea.** You specify the action to occur when I/O is ready:

- nothing;
- start a new thread; or
- raise a signal.

Your code submits the requests using e.g. `aio_read` and `aio_write`. If needed, wait for I/O to happen using `aio_suspend`.

## `curl_multi`

### Using cURL Asynchronously

We've already seen that network communication is a great example of a way that you could use asynchronous I/O. You can start a network request and move on to creating more without waiting for the results of the first one. For requests to different recipients, it certainly makes sense to do this.

The cURL multi interface has a lot of similarities with the regular cURL interface. It's been a little while since we went over it, so let's recap what we did before. Remember from earlier the example we did that was modified from <https://curl.haxx.se/libcurl/c/https.html> (i.e., the official docs):

```
#include <stdio.h>
#include <curl/curl.h>

int main( int argc, char** argv ) {
    CURL *curl;
    CURLcode res;

    curl_global_init(CURL_GLOBAL_DEFAULT);

    curl = curl_easy_init();
    if( curl ) {
        curl_easy_setopt(curl, CURLOPT_URL, "https://example.com/" );
        res = curl_easy_perform( curl );

        if( res != CURLE_OK ) {
            fprintf(stderr, "curl_easy_perform()_failed:_%s\n", curl_easy_strerror(res));
        }
    }
}
```

```

    }
    curl_easy_cleanup(curl);
}

curl_global_cleanup();
return 0;
}

```

In the previous example, the call to `curl_easy_perform()` is blocking and we wait for the curl execution to take place. We want to change that! The tool for this is the “multi handle” - this is a structure that lets us have more than one curl easy handle. And rather than waiting, we can start them and then check on their progress.

There are still the global initialization and cleanup functions. The structure for the new multi-handle type is `CURLM` (instead of `CURL`) and it is initialized with the `curl_multi_init()` function.

Once we have a multi handle, we can add easy handles – however many we need – to the multi handle. Creation of the easy handle is the same as it is when being used alone - use `curl_easy_init()` to create it and then we can set however many options on this we need. Then, we add the easy handle to the multi handle with `curl_multi_add_handle( CURLM* cm, CURL* eh )`.

Once we have finished putting all the easy handles into the multi handle, we can dispatch them all at once with `curl_multi_perform( CURLM* cm, int* still_running )`. The second parameter is a pointer to an integer that is updated with the number of the easy handles in that multi handle that are still running. If it's down to 0, then we know that they are all done. If it's nonzero it means that some of them are still in progress.

This does mean that we're going to call `curl_multi_perform()` more than once. Doing so doesn't restart or interfere with anything that was already in progress – it just gives us an update on the status of what's going on. We can check as often as we'd like, but the intention is of course to do something useful while the asynchronous I/O request(s) are going on. Otherwise, why not make it synchronous?

Suppose we've run out of things to do though. What then? Well, we can wait, if we want, using `curl_multi_wait( CURLM *multi_handle, struct curl_waitfd extra_fds[], unsigned int extra_nfds, int timeout_ms, int *numfds )`. This function will block the current thread until something happens (some event occurs).

The first parameter is the multi handle, which makes sense. The second parameter is a structure of extra file descriptors you can wait on (but we will always want this to be `NULL` in this course) and the third parameter is the count (the size of the provided array) which would also be zero here. Then the second-last parameter is a maximum time to wait. The last parameter is a pointer that will be updated with the actual number of “interesting” events that occurred (interesting is the word used in the specifications, and what it means is mysterious). For a simple use case you can ignore most of the parameters and just wait for something to happen and go from there.

In the meantime though, the perform operations are happening, and so are whatever callbacks we have set up (if any). And as the I/O operation moves through its life cycle, the state of the easy handle is updated appropriately. Each easy handle has an associated status message as well as a return code.

Why both? Well - one is about what the status of the request is. The message could be, for example “done”, but does that mean finished with success or finished with an error? For the second one tells us about that. We can ask about the status of the request using `curl_multi_info_read( CURLM* cm, int* msgs_left )`. This returns a pointer to information “next” easy handle, if there is one. The return value is a pointer to a struct of type `CURLMsg`. Along side this, the parameter `msgs_left` is updated to say how many messages remain (so you don't have to remember or know in advance, really).

We will therefore check the `CURLMsg` message to see what happened and make sure all is well. If our message that we got back with the info read is called `m`, What we are looking for is that the `m->msg` is equal to `CURLMSG_DONE` – request completed. If not, this request is still in progress and we aren't ready to evaluate whether it was successful or not. If there are more handles to look at, we should go on to the next. If it is done, we should look at the return code in and the result, in `m->data.result`. If it is `CURLE_OK` then everything succeeded. If it's anything else, it indicates an error.

When a handle has finished, you need to remove it from the multi handle. A pointer to it is inside the CURLMsg under `m->easy_handle`. It is removed with `curl_multi_remove_handle( CURLM* cm, CURL eh )`. Once removed, it should be cleaned up like normal with `curl_easy_cleanup( CURL* eh )`.

There is of course the corresponding cleanup function `curl_multi_cleanup( CURLM * cm )` for the multi handle when we are done with all the easy handles inside. The last step, as before, is to use the global cleanup function. After that we are done.

Let's consider the following code example by Clemens Gruber [Gru13], with slight modifications for compactness, formatting, and to remember the cleanup. This example puts together all the things we talked about in one compact code segment. Here, the callback does nothing, but that's okay – it's just to show what you could do with it.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <curl/multi.h>

#define MAX_WAIT_MSECS 30*1000 /* Wait max. 30 seconds */

const char *urls[] = {
    "http://www.microsoft.com",
    "http://www.yahoo.com",
    "http://www.wikipedia.org",
    "http://slashdot.org"
};

#define CNT 4

size_t cb(char *d, size_t n, size_t l, void *p) {
    /* take care of the data here, ignored in this example */
    return n*l;
}

void init( CURLM *cm, int i ) {
    CURL *eh = curl_easy_init();
    curl_easy_setopt( eh, CURLOPT_WRITEFUNCTION, cb );
    curl_easy_setopt( eh, CURLOPT_HEADER, 0L );
    curl_easy_setopt( eh, CURLOPT_URL, urls[i] );
    curl_easy_setopt( eh, CURLOPT_PRIVATE, urls[i] );
    curl_easy_setopt( eh, CURLOPT_VERBOSE, 0L );
    curl_multi_add_handle( cm, eh );
}

int main( int argc, char** argv ) {
    CURLM *cm = NULL;
    CURL *eh = NULL;
    CURLMsg *msg = NULL;
    CURLcode return_code = 0;
    int still_running = 0;
    int msgs_left = 0;
    int http_status_code;
    const char *szUrl;

    curl_global_init( CURL_GLOBAL_ALL );
    cm = curl_multi_init( );

    for ( int i = 0; i < CNT; ++i ) {
        init( cm, i );
    }

    curl_multi_perform( cm, &still_running );

    do {
        int numfds = 0;
        int res = curl_multi_wait( cm, NULL, 0, MAX_WAIT_MSECS, &numfds );
        if( res != CURLM_OK ) {
            fprintf( stderr, "error:_curl_multi_wait()_returned_%d\n", res );
            return EXIT_FAILURE;
        }
    }
```

```

    curl_multi_perform( cm, &still_running );

} while( still_running );

while ( ( msg = curl_multi_info_read( cm, &msgs_left ) ) ) {
    if ( msg->msg == CURLMSG_DONE ) {
        eh = msg->easy_handle;

        return_code = msg->data.result;
        if ( return_code != CURLE_OK ) {
            fprintf( stderr, "CURL_error_code:_%d\n", msg->data.result );
            continue;
        }

        // Get HTTP status code
        http_status_code = 0;
        szUrl = NULL;

        curl_easy_getinfo( eh, CURLINFO_RESPONSE_CODE, &http_status_code );
        curl_easy_getinfo( eh, CURLINFO_PRIVATE, &szUrl );

        if( http_status_code == 200 ) {
            printf( "200_OK_for_%s\n", szUrl );
        } else {
            fprintf( stderr, "GET_of_%s_returned_http_status_code_%d\n", szUrl, http_status_code );
        }

        curl_multi_remove_handle( cm, eh );
        curl_easy_cleanup( eh );
    } else {
        fprintf( stderr, "error:_after_curl_multi_info_read(),_CURLMsg=%d\n", msg->msg );
    }
}
curl_multi_cleanup( cm );
curl_global_cleanup();
return 0;
}

```

You may wonder about re-using an easy handle rather than removing and destroying it and making a new one. The official docs say that you can re-use one, but you have to remove it from the multi handle and then re-add it, presumably after having changed anything that you want to change about that handle.

In this example all requests had the same (useless) callback, but of course you could have different callbacks for different easy handles if you wanted them to do different things.

## Building Servers: Concurrent Socket I/O

Switching gears, we'll talk about building software that handles tons of connections. Let's go back to that initial question:

What is the ideal design for server process in Linux that handles concurrent socket I/O?

So far in this class, we've seen:

- processes;
- threads;
- thread pools; and
- non-blocking/async I/O.

We'll analyze the answer by Robert Love, Linux kernel hacker [Lov13]:

## The Real Question.

How do you want to do I/O?

The question is not really “how many threads should I use?”.

**Your Choices.** The first two both use blocking I/O, while the second two use non-blocking I/O.

- Blocking I/O; 1 process per request.
- Blocking I/O; 1 thread per request.
- Asynchronous I/O, pool of threads, callbacks, each thread handles multiple connections.
- Nonblocking I/O, pool of threads, multiplexed with select/poll, event-driven, each thread handles multiple connections.

**Blocking I/O; 1 process per request.** This is the old Apache model.

- The main thread waits for connections.
- Upon connect, the main thread forks off a new process, which completely handles the connection.
- Each I/O request is blocking, e.g., reads wait until more data arrives.

Advantage:

- “Simple to understand and easy to program.”

Disadvantage:

- High overhead from starting 1000s of processes. (We can somewhat mitigate this using process pools).

This method can handle ~10 000 processes, but doesn’t generally scale beyond that, and uses many more resources than the alternatives.

**Blocking I/O; 1 thread per request.** We know that threads are more lightweight than processes. So let’s use threads instead of processes. Otherwise, this is the same as 1 process per request, but with less overhead. I/O is the same—it is still blocking.

Advantage:

- Still simple to understand and easy to program.

Disadvantages:

- Overhead still piles up, although less than processes.
- New complication: race conditions on shared data.

**Asynchronous I/O.** The other two choices don't assign one thread or process per connection, but instead multiplex the threads to connections. We'll first talk about using asynchronous I/O with select or poll.

Here are (from 2006) some performance benefits of using asynchronous I/O on lighttpd [Tea06].

version		fetches/sec	bytes/sec	CPU idle
1.4.13	sendfile	36.45	3.73e+06	16.43%
1.5.0	sendfile	40.51	4.14e+06	12.77%
1.5.0	linux-aio-sendfile	72.70	7.44e+06	46.11%

(Workload:  $2 \times 7200$  RPM in RAID1, 1GB RAM, transferring 10GBytes on a 100MBit network).

The basic workflow is as follows:

1. enqueue a request;
2. ... do something else;
3. (if needed) periodically check whether request is done; and
4. read the return value.

Some code which uses the Linux asynchronous I/O model is:

```
#include <aio.h>

int main() {
    // so far, just like normal
    int file = open("blah.txt", O_RDONLY, 0);

    // create buffer and control block
    char* buffer = new char[SIZE_TO_READ];
    aiocb cb;

    memset(&cb, 0, sizeof(aiocb));
    cb.aio_nbytes = SIZE_TO_READ;
    cb.aio_fildes = file;
    cb.aio_offset = 0;
    cb.aio_buf = buffer;

    // enqueue the read
    if (aio_read(&cb) == -1) { /* error handling */ }

    do {
        // ... do something else ...
        while (aio_error(&cb) == EINPROGRESS); // poll

        // inspect the return value
        int numBytes = aio_return(&cb);
        if (numBytes == -1) { /* error handling */ }

        // clean up
        delete[] buffer;
        close(file);
    }
```

**Using Select/Poll.** The idea is to improve performance by letting each thread handle multiple connections. When a thread is ready, it uses select/poll to find work:

- perhaps it needs to read from disk into a mmap'd tempfile;
- perhaps it needs to copy the tempfile to the network.

In either case, the thread does work and updates the request state.



## Callback-Based Asynchronous I/O Model

Finally, we'll talk about a not-very-popular programming model for non-blocking I/O (at least for C programs; it's the only game in town for JavaScript and a contender for Go). Instead of select/poll, you pass a callback to the I/O routine, which is to be executed upon success or failure.

```
void new_connection_cb (int cfd)
{
    if (cfd < 0) {
        fprintf (stderr, "error_in_accepting_connection!\n");
        exit (1);
    }

    ref<connection_state> c =
        new refcounted<connection_state>(cfd);

    // what to do in case of failure: clean up.
    c->killing_task = delaycb(10, 0, wrap(&clean_up, c));

    // link to the next task: got the input from the connection
    fdcb (cfd, selread, wrap (&read_http_cb, cfd, c, true,
        wrap(&read_req_complete_cb)));
}
```

**node.js: A Superficial View.** We'll wrap up today by talking about the callback-based node.js model. node.js is another event-based nonblocking I/O model. Given that JavaScript doesn't have threads, the only way to write servers is using non-blocking I/O.

The canonical example from the node.js homepage:

```
var http = require('http');
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
console.log('Server_running_at_http://127.0.0.1:1337/');
```

Note the use of the callback—it's called upon each connection.

However, usually we don't want to handle the fields in the request manually. We'd prefer a higher-level view. One option is expressjs<sup>1</sup>, and here's an example from the Internet [Gli11]:

```
app.post('/nod', function(req, res) {
    loadAccount(req,function(account) {
        if(account && account.username) {
            var n = new Nod();
            n.username = account.username;
            n.text = req.body.nod;
            n.date = new Date();
            n.save(function(err){
                res.redirect('/');
            });
        }
    });
});
```

## References

[Gli11] Travis Glines. nod.js, 2011. Online; accessed 23-November-2015. URL: <https://github.com/tglines/nodrr/blob/master/controllers/nod.js>.

---

<sup>1</sup><http://expressjs.com>

- [Gru13] Clemens Gruber. libcurl multi interface example, 2013. Online; accessed 30-October-2018. URL: <https://gist.github.com/clemensg/4960504>.
- [Lov13] Robert Love. What is the ideal design for a server process in linux that handles concurrent socket i/o, 2013. Online; accessed 23-November-2015. URL: <https://plus.google.com/+RobertLove/posts/VPMT8ucAcFH>.
- [Tea06] Lighty Team. Lighty 1.5.0 and Linux-aio, 2006. Online; accessed 23-November-2015. URL: <http://blog.lighttpd.net/articles/2006/11/12/lighty-1-5-0-and-linux-aio/>.