ECE459: Programming for Performance Winter 2018

Lecture 9 — Asynchronous I/O, epoll, select

Patrick Lam

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`)

"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` behavious 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`.

Nonblocking I/O with curl. The next lecture notes give more clue about nonblocking I/O with curl. Although it doesn't work with `epoll` but rather `select`, it uses the same ideas—we'll therefore see two (three, with aio) different implementations of the same idea. Briefly, you:

- build up a set of descriptors;
- invoke the transfers and wait for them to finish; and
- see how things went.

curl_multi

It's important to see at least one specific example of an idea. I talked about `epoll` and I meant that to be the specific example, but we can't quite use it without getting into socket programming, and I don't want to do that. Instead, we'll see non-blocking I/O in the specific example of the curl library, which is reasonably widely used in the Linux world.

Tragically, it's complicated to use `epoll` with `curl_multi`, and I couldn't quite figure it out. So I'll describe the `select`-based interface for `curl_multi`. A socket-based interface which works with `epoll` also exists. I won't talk about that.

The relevant steps, in any case, are:

- Create individual requests with `curl_easy_init`.

- Create a multi-handle with `curl_multi_init` and add the requests to it with `curl_multi_add_handle`.
- (for select-based interface:) put in requests & wait for results, using `curl_multi_perform`. That call generalizes `curl_easy_perform`.
- Handle completed transfers with `curl_multi_info_read`.

On the use of `curl_multi_perform`. The actual non-blocking read/write is done in `curl_multi_perform`, which returns the number of still-active handles through its parameter.

You call it in a loop, with a call to `select` above. Call `select` and then `curl_multi_perform` in a loop while there are still running transfers. You're also allowed to manipulate (delete/alter/re-add) a `curl_easy_handle` whenever a transfer finishes.

Setting up the `select`. Before you call `curl_multi_perform` and `select`, you need to set up the `select`. The curl call `curl_multi_fdset` sets up the parameters for the `select`, while `curl_multi_timeout` gives you the proper timeout to hand to `select`.

```
// zero the fd-sets
FD_ZERO(&fdread); FD_ZERO(&fdwrite); FD_ZERO(&fdexcep);
// retrieve the fds, check for error
curl_multi_fdset(multi_handle,
                 &fdread, &fdwrite, &fdexcep, &maxfd);
if (maxfd < -1) abort_("multi_fdset: couldn't wait for fds");
// retrieve the timeout
curl_multi_timeout(multi_handle, &curl_timeout);
```

In an API infelicity, you have to convert the `curl_timeout` into a `struct timeval` for use by `select`.

Calling `select`. The call itself is fairly straightforward:

```
rc = select(maxfd + 1, &fdread, &fdwrite, &fdexcep, &timeout);
if (rc == -1) abort_("[main] select error");
```

This waits for one of the file descriptors to become ready, or for the timeout to elapse (whichever happens first).

Of course, once `select` returns, you only know that something happened, but you haven't done the work yet. So you then need to call `curl_multi_perform` again to do the work.

Finally, you get the results of `curl_multi_perform` by calling `curl_multi_info_read`. It also tells you how many messages are left.

```
msg = curl_multi_info_read(multi_handle, &msgs_left);
```

The return value `msg->msg` can be either `CURLMSG_DONE` or an error. The handle `msg->easy_handle` tells you which handle finished. You may have to look that up in your collection of handles.

Some gotchas (thanks Desiye Collier):

- Checking `msg->msg == CURLMSG_DONE` is not sufficient to ensure that a curl request actually happened. You also need to check `data.result`.
- (A1 hint:) To reset an individual handle in the `multi_handle`, you need to "replace" it. But you shouldn't use `curl_easy_init()` to replace the handle. In fact, you don't need a new handle at all.

Cleanup. Always clean up after yourself! Use `curl_multi_cleanup` to destroy the multi-handle and `curl_easy_cleanup` to destroy each individual handle. If you replace the `curl_easy_init` call by `curl_global_init` for the multi-threaded case (which is a good idea), then you should use `curl_global_cleanup` to clean up.

Example. There is a not-great example at

`http://curl.haxx.se/libcurl/c/multi-app.html`

but I'm not even sure it works verbatim. You could use it as a solution template, but you'll need to add more code—I asked you to replace completed transfers in the `curl_multi`.

A better way to use sockets. Late-breaking news: instead of that `select()` crud, use `curl_multi_wait()`, which is just better, and easy to use. An example: `https://gist.github.com/clemensg/4960504`

About that socket-based alternative. There is yet another interface which would allow you to use `epoll`, but I couldn't figure it out. Sorry. The advantage, beyond using `epoll`, is that `libcurl` doesn't need to scan over all of the transfers when it receives notice that a transfer is ready. This can help when there are lots of sockets open.

From the manpage:

- Create a multi handle
- Set the socket callback with `CURLMOPT_SOCKETFUNCTION`
- Set the timeout callback with `CURLMOPT_TIMERFUNCTION`, to get to know what timeout value to use when waiting for socket activities.
- Add easy handles with `curl_multi_add_handle()`
- Provide some means to manage the sockets libcurl is using, so you can check them for activity. This can be done through your application code, or by way of an external library such as libevent or glib.
- Call `curl_multi_socket_action(..., CURL_SOCKET_TIMEOUT, 0, ...)` to kickstart everything. To get one or more callbacks called.
- Wait for activity on any of libcurl's sockets, use the timeout value your callback has been told.
- When activity is detected, call `curl_multi_socket_action()` for the socket(s) that got action. If no activity is detected and the timeout expires, call `curl_multi_socket_action(3)` with `CURL_SOCKET_TIMEOUT`.

There's an example, which has too many moving parts, here:

`http://curl.haxx.se/libcurl/c/hiperfifo.html`

It uses `libevent`, which I totally don't want to talk about in this class.

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 undertand and easy to program."

Disadvantage:

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

This method can handle $\sim$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: 2×7200 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[1], and here's an 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`.

[1] `http://expressjs.com`

[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/`.