Lecture 8 — Asynchronous I/O, epoll, select

Patrick Lam
patrick.lam@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 4, 2018

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.

Why non-blocking I/O?

Consider some I/O:

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

Not very performant—under what conditions do we lose out?

Mitigating I/O impact

So far: can use threads to mitigate latency.
What are the disadvantages?

Mitigating I/O impact

So far: can use threads to mitigate latency.
What are the disadvantages?

- race conditions
- overhead/max # of thread limitations

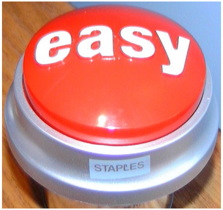
Live coding: Fork Bomb!

(well, threadbomb anyway)

Asynchronous/nonblocking I/O.

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

...



(credit: Yskyflyer, Wikimedia Commons)

Not Quite So Easy: Live Demo

Doesn't work on files—they're always ready. Only e.g. sockets.

Other Outstanding Problem with Nonblocking I/O

How do you know when I/O is ready to be queried?

Other Outstanding Problem with Nonblocking I/O

How do you know when I/O is ready to be queried?

- polling (`select`, `poll`, `epoll`)
- interrupts (signals)

Using `epoll`

Key idea: give `epoll` a bunch of file descriptors;
wait for events to happen.

Steps:

1. create an instance (`epoll_create1`);
2. populate it with file descriptors (`epoll_ctl`);
3. wait for events (`epoll_wait`).

Creating an `epoll` instance

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

`efpd` doesn't represent any files; use it to talk to `epoll`.

0 represents the flags (only flag: `EPOLL_CLOEXEC`).

Populating the `epoll` instance

To add `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);
```

Can also modify and delete descriptors from `epfd`.

Waiting on an epoll instance

Now 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);
```

-1: wait potentially forever; otherwise, milliseconds to wait.

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

Level-Triggered and Edge-Triggered Events

Default `epoll` behaviour is level-triggered:
return whenever data is ready.

Can also specify (via `epoll_ctl`) edge-triggered behaviour:
return whenever there is a change in readiness.

Asynchronous I/O

POSIX standard defines aio calls.

These work for disk as well as sockets.

Key idea: you specify the action to occur when I/O is ready:

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

Submit the requests using e.g. `aio_read` and `aio_write`.

Can wait for I/O to happen using `aio_suspend`.

Nonblocking I/O with curl

Similar idea to `epoll`:

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

curl_multi initialization

curl_multi: work with multiple resources at once.

How? Similar idea to `epoll`:

1. To use `curl_multi`, first create the individual requests (`curl_easy_init`).
   (Set options as needed on each handle).

2. Then, combine them with:

- `curl_multi_init();`
- `curl_multi_add_handle()`.

curl_multi_perform: option 1, select-based interface

Main idea: put in requests and wait for results.

`curl_multi_perform` is a generalization of `curl_easy_perform` to multiple resources.

Handle completed transfers with `curl_multi_info_read`.

calling curl_multi_perform

perform interface requires use of `select` (not `epoll`).

usage (once you've `curl_multi_add_handle`'d):

```
curl_multi_perform(multi_handle, &still_running)
```

performs a non-blocking read/write, and
returns the number of still-active handles
(with more data to come).

Next steps after curl_multi_perform

do

- organize a call to `select`; and
- call `curl_multi_perform` again

while there are still running transfers.

After the `curl_multi_perform`, you can also delete, alter, and re-add an `curl_easy_handle` when a transfer finishes.

Before calling `select`

`select` needs a timeout and an `fdset`.
    (`curl` provides both.)

Initializing the `fdset` from the `multi_handle`:

```
// 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");
```

Retrieving the proper timeout:

```
curl_multi_timeout(multi_handle, &curl_timeout);
```

(and then convert the `long` to a `struct timeval`).

The call to `select`

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

Wait for one of the fds to become ready,
or for timeout to elapse.

What next?

The call to select

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

Wait for one of the fds to become ready,
or for timeout to elapse.

What next?
Call `curl_multi_perform` again to do the work.

Knowing what happened after `curl_multi_perform`

`curl_multi_info_read` will tell you.

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

and also how many messages are left.

`msg->msg` can be `CURLMSG_DONE` or an error;
`msg->easy_handle` tells you who is done.

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()`. In fact, you don't need a new handle at all.

Call `curl_multi_cleanup` on the multi handle.

Then, call `curl_easy_cleanup` on each easy handle.

If you replace `curl_easy_init` by `curl_global_init`,
then call `curl_global_cleanup` also.

curl_multi_perform example

Not a great example:

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

I'm not even sure it works verbatim.

Nevertheless, you could use it as a solution template.
You'll have to add more code to replace completed transfers.

A better choice: curl_multi_wait

Instead of using `select()`,
you can use `curl_multi_wait()`.

It's just better.
`https://gist.github.com/clemensg/4960504`

curl_multi, option 3: curl_multi_socket_action

So, I couldn't quite figure out how this works. Sorry.

Similar to the `perform` interface, but you have more control.

Advantage:

> *2 - When the application discovers action on a single socket, it calls libcurl and informs that there was action on this particular socket and libcurl can then act on that socket/transfer only and not care about any other transfers. (The previous API always had to scan through all the existing transfers.)*

`http://curl.haxx.se/dev/readme-multi_socket.html`

multi_socket usage

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`.

multi_socket example

This example is even worse than the last one:

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

It contains more moving parts than we need to understand the API, and gets another library (`libevent`) involved.

Concurrent Socket I/O

Complete change of topic. A Quora 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
- async/non-blocking I/O.

We'll see the answer by Robert Love, Linux kernel hacker[1].

[1] `https://plus.google.com/105706754763991756749/posts/VPMT8ucAcFH`

How do you want to do I/O?

Not really "how many threads?".

Four Choices

- 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

Old Apache model:

- Main thread waits for connections.
- Upon connect, 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.
  (can somewhat mitigate with process pool).

Can handle ∼10 000 processes, but doesn’t generally scale.

Blocking I/O; 1 thread per request

We know that threads are more lightweight than processes.

Same as 1 process per request, but less overhead.

I/O is the same—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 Benefits

In 2006, perf benefits of asynchronous I/O on lighttpd[2]:

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).

[2] `http://blog.lighttpd.net/articles/2006/11/12/lighty-1-5-0-and-linux-aio/`

Using Asynchronous I/O in Linux (select/poll)

Basic workflow:

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

Asynchronous I/O Code Example I: Setup

```
#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;
```

Asynchronous I/O Code Example II: Read

```
// 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);
```

Nonblocking I/O in Servers using Select/Poll

Each thread handles 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

Weird programming model; not popular.

Instead of select/poll, pass along a callback,
to be executed upon success or failure.

JavaScript does this extensively, but more unwieldy in C.

We'll see the Go programming model, which makes this easy.

Callback-Based Example

```
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);

  c->killing_task = delaycb(10, 0, wrap(&clean_up, c));

  /* next step: read information on the new connection */
  fdcb (cfd, selread, wrap (&read_http_cb, cfd, c, true,
                   wrap(&read_req_complete_cb)));
}
```

node.js: A Superficial View

node.js is another event-based nonblocking I/O model.

(Since JavaScript is singlethreaded, nonblocking I/O mandatory.)

Canonical example from 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.

Usually we want a higher-level view, e.g. expressjs[3].

An example from the Internet[4]:

```
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('/');
      });
    }
  });
});
```

[3] http://expressjs.com

[4] https://github.com/tglines/nodrr/blob/master/controllers/nod.js

Summary: Building Servers

- Blocking I/O; 1 process per request (old Apache).
- Blocking I/O; 1 thread per request (Java).
- Asynchronous I/O, pool of threads, callbacks,
  each thread handles multiple connections. (no one does this)
- Nonblocking I/O, pool of threads,
  multiplexed with select/poll, event-driven,
  each thread handles multiple connections. (JavaScript)