ECE459: Programming for Performance Winter 2015

Lecture 07X — "hors série", 2015

Patrick Lam *version 1*

I found these lecture notes lying around from previous year. I feel that I don't need to lecture about them: there's a lot of overlap with the non-blocking I/O content. But you may enjoy reading these, as they provide more context about programming servers for performance. This material will not appear on exams.

Building Servers: Concurrent Socket I/O

Switching gears, we'll talk about building software that handles tons of connections. From 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
- non-blocking/async I/O.

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

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.

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

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 ∼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[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).

The basic workflow is as follows:

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

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[3], and here's an 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;
```

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

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

```
      n.text = req.body.nod;
      n.date = new Date();
      n.save(function(err){
        res.redirect('/');
      });
    }
  });
});
```