

# Lecture 06X—Building Servers

ECE 459: Programming for Performance

“hors série”

# Part I

## Building Servers

## Editor's Note

I found these notes from last year.

There's a lot of overlap with content seen so far,  
but also some general design discussion that may help.

Enjoy!

# 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<sup>1</sup>.

---

<sup>1</sup><https://plus.google.com/105706754763991756749/posts/VPMT8ucAcFH>

# The Real Question

*How do you want to do I/O?*

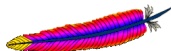
Not really “how many threads?”.

## 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 understand and easy to program.”

Disadvantage:

- High overhead from starting 1000s of processes.  
(can somewhat mitigate with process pool).

Can handle  $\sim 10\,000$  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<sup>2</sup>:

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).

---

<sup>2</sup><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.



# Building on node.js

Usually we want a higher-level view, e.g. `expressjs`<sup>3</sup>.

An example from the Internet<sup>4</sup>:

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

---

<sup>3</sup><http://expressjs.com>

<sup>4</sup><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)