Lecture 8 — Asynchronous I/O, epoll, select

Patrick Lam
patrick.lam@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 18, 2019

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

Asynchronous/nonblocking I/O.

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

...

(credit: Yskyflyer, Wikimedia Commons)

Not Quite So Easy

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)

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.

Here's a simple cURL program that we can look over:

```c
#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;
}
```

curl_multi initialization

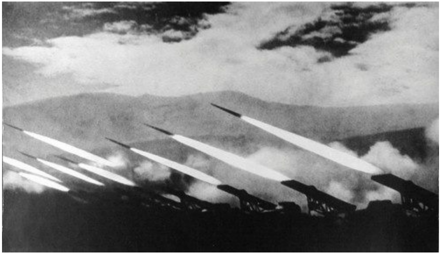
curl_multi: work with multiple resources at once.

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()`.

Start reqs: `curl_multi_perform( CURLM* cm, int* still_running )`



The second parameter is updated with the number of still-in-progress requests.

Meantime, we can do other things!

I'm bored... So take a nap?

Suppose we've run out of things to do and nothing is ready yet. Wait!

```
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.

Choose how long to wait and see how many events occurred.

While You Were Sleeping

While we are asleep or doing other things, callbacks still happen.

The status of the cURL easy handle is updated.

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.

curl_multi cleanup

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.

Long Example

The long example is too big for the slides.

We’ll have to take look at it in the notes!

Process, Threads, AIO?! 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.

In 2006, perf benefits of asynchronous I/O on lighttpd[1]:

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).

[1] `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.