Lecture 07—Async I/O

January 19, 2015

Roadmap

Past: Modern Hardware, Threads;

Now: Non-blocking I/O;

Next: Race Conditions, Locking.

Last Time

- Assignment 1 walkthrough.
- Concept behind non-blocking I/O.

Part I

Async I/O with epoll

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.

Live Coding: Level-Triggered vs Edge-Triggered

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.

Part II

Using curl_multi

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.

curl_multi cleanup

Call `curl_multi_cleanup` on the multi handle.

Then, call `curl_easy_cleanup` on each easy handle.

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.

curl_multi, option 2: 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.