Lecture 27 — Asynchronous I/O with cURL

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 3, 2020

Using cURL

We've already seen that network communication is a great example of a way that you could use asynchronous I/O.

You can start a network request and move on to creating more without waiting for the results of the first one.

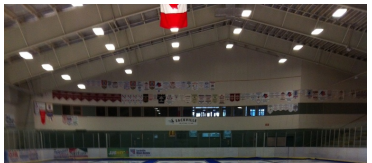
For requests to different recipients, it certainly makes sense to do this.

And yet, we've seen that while socket programming is important and necessary, for a lot of situations we prefer to use cURL.

Curl Multi

Curl:

Curl Multi:



Quick Recap of the Easy interface:

```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

The call to `curl_easy_perform()` is blocking and we wait for the curl execution to take place.

We want to change that!

The tool for this is the “multi handle” - this is a structure that lets us have more than one curl easy handle.

And rather than waiting, we can start them and then check on their progress.

Some Things Don't Change

There are still the global initialization and cleanup functions.

The structure for the new multi-handle type is `CURLM`.

It is initialized with the `curl_multi_init()` function.

Climb Aboard

Once we have a multi handle, we can add easy handles – however many we need – to the multi handle.

Creation of the easy handle is the same as it is when being used alone - use `curl_easy_init()` to create it.

Then we can set however many options on this we need.

Then, we add the easy handle to the multi handle with `curl_multi_add_handle( CURLM* cm, CURL* eh ).`

BOM-BART-MENT!

Image Credit: The Simpsons

We can dispatch all easy handles at once with
`curl_multi_perform( CURLM* cm, int* still_running ).`

Still Running

The second parameter is a pointer to an integer that is updated with the number of the easy handles in that multi handle that are still running.

If it's down to 0, then we know that they are all done.

If it's nonzero it means that some of them are still in progress.

Asking is Okay

This does mean that we're going to call`curl_multi_perform()` more than once.

Doing so doesn't restart or interfere with anything that was already in progress.

We can check as often as we would like – but we should be doing something useful rather than polling… right?

Naps Are Okay

Suppose we've run out of things to do though. What then?

```
curl_multi_wait( CURLM *multi_handle, struct curl_waitfd extra_fds[],
    unsigned int extra_nfds, int timeout_ms, int *numfds )} .
```

`multi_handle`: the multi handle to watch.

`extra_fds`: an array of extra file descriptors to watch (NULL).

`extra_nfds`: how many extra file descriptors there are (0).

`timeout_ms` timeout in ms… clearly…

`numfds` updated with the number of "interesting" events that occurred.

Meanwhile...

In the meantime though, the perform operations are happening, and so are whatever callbacks we have set up (if any).

And as the I/O operation moves through its life cycle, the state of the easy handle is updated appropriately.

Each easy handle has an associated status message as well as a return code.

Both of them?

The message could be, for example “done”, but does that mean finished with success or finished with an error?

Both of them?

We can ask about the status of the request using `curl_multi_info_read( CURLM* cm, int* msgs_left )`.

This returns a pointer to information “next” easy handle, if there is one.

The return value is a pointer to a struct of type `CURLMsg`.

Along side this, the parameter `msgs_left` is updated to say how many messages remain.

Read Your E-mails!

We will therefore check the `CURLMsg` message to see what happened and make sure all is well.

If our message that we got back with the info read is called `m`, What we are looking for is that the `m->msg` is equal to `CURLMSG_DONE` – request completed.

If not, this request is still in progress and we aren't ready to evaluate whether it was successful or not.

If done, we should look at the return code in and the result, in `m->data.result`.

If it is `CURLE_OK` then everything succeeded. If anything else, it indicates an error.

Remove Easy Handle

When a handle has finished, you need to remove it from the multi handle.

A pointer to it is inside the `CURLMsg` under `m->easy_handle`.

Removed with `curl_multi_remove_handle( CURLM* cm, CURL* eh )`.

Once removed, clean it up with: `curl_easy_cleanup( CURL* eh )`.

Clean the Multi

There is the multi cleanup function `curl_multi_cleanup( CURLM * cm ) .`

The last step, as before, is to use the global cleanup function.

After that we are done.

cURL Multi Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <curl/multi.h>

#define MAX_WAIT_MSECS 30*1000 /* Wait max. 30 seconds */

const char *urls[] = {
  "http://www.microsoft.com",
  "http://www.yahoo.com",
  "http://www.wikipedia.org",
  "http://slashdot.org"
};
#define CNT 4

size_t cb(char *d, size_t n, size_t l, void *p) {
  /* take care of the data here, ignored in this example */
  return n*l;
```

cURL Multi Example

```
void init( CURLM *cm, int i ) {
  CURL *eh = curl_easy_init();
  curl_easy_setopt( eh, CURLOPT_WRITEFUNCTION, cb );
  curl_easy_setopt( eh, CURLOPT_HEADER, OL );
  curl_easy_setopt( eh, CURLOPT_URL, urls[i] );
  curl_easy_setopt( eh, CURLOPT_PRIVATE, urls[i]) ;
  curl_easy_setopt( eh, CURLOPT_VERBOSE, OL );
  curl_multi_add_handle( cm, eh );
}

int main( int argc, char** argv ) {
    CURLM *cm = NULL;
    CURL *eh = NULL;
    CURLMsg *msg = NULL;
    CURLcode return_code = O;
    int still_running = O;
    int msgs_left = O;
    int http_status_code;
    const char *szUrl;

    curl_global_init( CURL_GLOBAL_ALL );
    cm = curl_multi_init( );

    for ( int i = O; i < CNT; ++i ) {
        init( cm, i );
    }
```

cURL Multi Example

```
curl_multi_perform( cm, &still_running );

do {
    int numfds = 0;
    int res = curl_multi_wait( cm, NULL, 0, MAX_WAIT_MSECS, &numfds );
    if( res != CURLM_OK ) {
        fprintf( stderr, "error: curl_multi_wait() returned %d\n", res );
        return EXIT_FAILURE;
    }
    curl_multi_perform( cm, &still_running );

} while( still_running );
```

cURL Multi Example

```
while ( ( msg = curl_multi_info_read( cm, &msgs_left ) ) ) {
    if ( msg->msg == CURLMSG_DONE ) {
        eh = msg->easy_handle;
        return_code = msg->data.result;
        if ( return_code != CURLE_OK ) {
            fprintf( stderr, "CURL error code: %d\n", msg->data.result );
            curl_multi_remove_handle( cm, eh );
            curl_easy_cleanup( eh );
            continue;
        }
        http_status_code = 0;
        szUrl = NULL;
        curl_easy_getinfo( eh, CURLINFO_RESPONSE_CODE, &http_status_code );
        curl_easy_getinfo( eh, CURLINFO_PRIVATE, &szUrl );

        if( http_status_code == 200 ) {
            printf( "200 OK for %s\n", szUrl ) ;
        } else {
            fprintf( stderr, "GET of %s returned http status code %d\n",
                 szUrl, http_status_code );
        }
        curl_multi_remove_handle( cm, eh );
        curl_easy_cleanup( eh );
    } else {
        fprintf( stderr, "error: after curl_multi_info_read(), CURLMsg=%d\n",
             msg->msg );
    }
}
```

```
    curl_multi_cleanup( cm );
    curl_global_cleanup();
    return 0;
}
```

Recycle, Reduce, Reuse

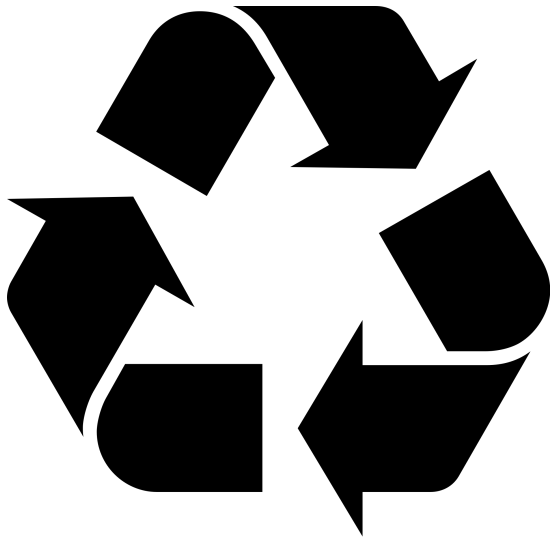


Image Credit: Wikipedia user Krdan

Can we re-use an easy handle rather than destroy and create a new one?

Recycle, Reduce, Reuse

The official docs say that you can re-use one.

But you have to remove it from the multi handle and then re-add it.

… presumably after having changed anything that you want to change about that handle.

Always Running

You could have a situation where there are constantly handles in progress.

You might never be at a situation where there are no messages left.

And that is okay.

Scaling This

The developer claims that you can have multiple thousands of connections in a single multi handle.

60k ought to be enough for anyone!

cURL + select

cURL + select

Well, alright, if you insist...

The first part of using cURL with select is of course setting up your multi handle and adding all of the easy handles to that.

Once you've done that, you can ask cURL to prep things for you:

```
CURLMcode curl_multi_fdset( CURLM *multi_handle, fd_set *read_fd_set,
    fd_set *write_fd_set, fd_set *exc_fd_set, int *max_fd );
```

It takes as arguments, the multi handle, and then pointers to the `fd_sets`.

It's the responsibility of the caller to set these to zero with FD_ZER0 first.

The function is kind enough to tell you what the maximum file descriptor is.

cURL + select

```
fd_set fdr;
fd_set fdw;
fd_set fde;
FD_ZERO( &fdr );
FD_ZERO( &fdw );
FD_ZERO( &fde );
int maxfd = -1;

CURLMCode res = curl_multi_fdset( cm, &fdr, &fdw, &fde, &maxfd );
```

When calling select we still have to put a +1 on the max file descriptor, but that is no big deal.

The missing element is the timeout. For that there is another function:

```
CURLMcode curl_multi_timeout( CURLM *multi_handle, long *timeout );
```

Unfortunately, though, this returns a `long`...

Guys, Why Can't You Get Along...

So you have to do the math yourself.

Use this!

```
struct timeval timeout;
long timeo;

curl_multi_timeout( cm, &timeo );
if(timeo < 0) {
  /* no set timeout, use a default */
  timeo = 980;
}

timeout.tv_sec = timeo / 1000;
timeout.tv_usec = (timeo % 1000) * 1000;
```

Are We Forgetting Something?

But we aren't ready to call `select` yet – we haven't started any of the transfers that use the sockets.

For that we use `curl_multi_perform` just as in the earlier example.

We can do that before or after setting up what we need for select.

Waking Up From a Nap

When we wake up, it is because something happened.

We still use `curl_multi_info_read` to find out what's going on with a given easy handle and find out its status and also find out how many are still running.

As with the earlier code where we used `select` on sockets directly, we have to remember to reset (repopulate) the values.

Select Headaches

One of the other problems with `select` is that you are limited in the number of file descriptors because you could overflow the bitmasks.

But even if you don't, this really does not scale, so you would be better off using the regular curl multi call.

As far as I can see it's better in pretty much every way.

Poll? Nope

Can we use cURL with `poll`? Generally, no.

Long story short: don't.