ECE 252: Systems Programming and Concurrency Spring 2019

Lecture 30 — Asynchronous I/O with cURL

Jeff Zarnett *2018-11-01*

Using cURL Asynchronously

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.

The cURL multi interface has a lot of similarities with the regular cURL interface. It's been a little while since we went over it, so let's recap what we did before. Remember from earlier the example we did that was modified from `https://curl.haxx.se/libcurl/c/https.html` (i.e., the official docs):

```
#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;
}
```

In the previous example, 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.

There are still the global initialization and cleanup functions. The structure for the new multi-handle type is `CURLM` (instead of `CURL`) and it is initialized with the `curl_multi_init()` function.

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 and 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 )`.

Once we have finished putting all the easy handles into the multi handle, we can dispatch them all at once with `curl_multi_perform( CURLM* cm, int* 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.

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 – it just gives us an update on the status of what's going on.

We can check as often as we'd like, but the intention is of course to do something useful while the asynchronous I/O request(s) are going on. Otherwise, why not make it synchronous?

Suppose we've run out of things to do though. What then? Well, we can wait, if we want, using `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 (some event occurs).

The first parameter is the multi handle, which makes sense. The second parameter is a structure of extra file descriptors you can wait on (but we will always want this to be NULL in this course) and the third parameter is the count (the size of the provided array) which would also be zero here. Then the second-last parameter is a maximum time to wait. The last parameter is a pointer that will be updated with the actual number of "interesting" events that occurred (interesting is the word used in the specifications, and what it means is mysterious). For a simple use case you can ignore most of the parameters and just wait for something to happen and go from there.

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.

Why both? Well - one is about what the status of the request is. The message could be, for example "done", but does that mean finished with success or finished with an error? For the second one tells us about that. 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 (so you don't have to remember or know in advance, really).

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 there are more handles to look at, we should go on to the next. If it is 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 it's anything else, it indicates an error.

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`. It is removed with `curl_multi_remove_handle( CURLM* cm, CURL eh )`. Once removed, it should be cleaned up like normal with `curl_easy_cleanup( CURL* eh )`.

There is of course the corresponding cleanup function `curl_multi_cleanup( CURLM * cm )` for the multi handle when we are done with all the easy handles inside. The last step, as before, is to use the global cleanup function. After that we are done.

Let's consider the following code example by Clemens Gruber [Gru13], with slight modifications for compactness, formatting, and to remember the cleanup. This example puts together all the things we talked about in one compact code segment. Here, the callback does nothing, but that's okay – it's just to show what you could do with it.

```
#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;
}

void init( CURLM *cm, int i ) {
  CURL *eh = curl_easy_init();
  curl_easy_setopt( eh, CURLOPT_WRITEFUNCTION, cb );
  curl_easy_setopt( eh, CURLOPT_HEADER, 0L );
  curl_easy_setopt( eh, CURLOPT_URL, urls[i] );
  curl_easy_setopt( eh, CURLOPT_PRIVATE, urls[i]) ;
  curl_easy_setopt( eh, CURLOPT_VERBOSE, 0L );
  curl_multi_add_handle( cm, eh );
}

int main( int argc, char** argv ) {
    CURLM *cm = NULL;
    CURL *eh = NULL;
    CURLMsg *msg = NULL;
    CURLcode return_code = 0;
    int still_running = 0;
    int msgs_left = 0;
    int http_status_code;
    const char *szUrl;

    curl_global_init( CURL_GLOBAL_ALL );
    cm = curl_multi_init( );

    for ( int i = 0; i < CNT; ++i ) {
        init( cm, i );
    }

    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 );

    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 );
                continue;
            }

            // Get HTTP status code
            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;
}
```

You may wonder about re-using an easy handle rather than removing and destroying it and making a new one. 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.

In this example all requests had the same (useless) callback, but of course you could have different callbacks for different easy handles if you wanted them to do different things.

References

[Gru13] Clemens Gruber. libcurl multi interface example, 2013. Online; accessed 30-October-2018. URL: `https://gist.github.com/clemensg/4960504`.