NAME

libcurl-multi − how to use the multi interface

DESCRIPTION

This is an overview on how to use the libcurl multi interface in your C programs. There are specific man pages for each function mentioned in here. There's also the *libcurl-tutorial(3)* man page for a complete tutorial to programming with libcurl and the *libcurl-easy(3)* man page for an overview of the libcurl easy interface.

All functions in the multi interface are prefixed with curl_multi.

OBJECTIVES

The multi interface offers several abilities that the easy interface doesn't. They are mainly:

1. Enable a "pull" interface. The application that uses libcurl decides where and when to ask libcurl to get/send data.

2. Enable multiple simultaneous transfers in the same thread without making it complicated for the application.

3. Enable the application to wait for action on its own file descriptors and curl's file descriptors simultaneous easily.

4. Enable event-based handling and scaling transfers up to and beyond thousands of parallel connections.

ONE MULTI HANDLE MANY EASY HANDLES

To use the multi interface, you must first create a 'multi handle' with *curl_multi_init(3)*. This handle is then used as input to all further curl_multi_* functions.

With a multi handle and the multi interface you can do several simultaneous transfers in parallel. Each single transfer is built up around an easy handle. You create all the easy handles you need, and setup the appropriate options for each easy handle using *curl_easy_setopt(3)*.

There are two flavours of the multi interface, the select() oriented one and the event based one we call multi_socket. You will benefit from reading through the description of both versions to fully understand how they work and differentiate. We start out with the select() oriented version.

When an easy handle is setup and ready for transfer, then instead of using *curl_easy_perform(3)* like when using the easy interface for transfers, you should add the easy handle to the multi handle with *curl_multi_add_handle(3)*. You can add more easy handles to a multi handle at any point, even if other transfers are already running.

Should you change your mind, the easy handle is again removed from the multi stack using *curl_multi_remove_handle(3)*. Once removed from the multi handle, you can again use other easy interface functions like *curl_easy_perform(3)* on the handle or whatever you think is necessary. You can remove handles at any point in time during transfers.

Adding the easy handle to the multi handle does not start the transfer. Remember that one of the main ideas with this interface is to let your application drive. You drive the transfers by invoking *curl_multi_perform(3)*. libcurl will then transfer data if there is anything available to transfer. It'll use the callbacks and everything else you have setup in the individual easy handles. It'll transfer data on all current transfers in the multi stack that are ready to transfer anything. It may be all, it may be none. When there's nothing more to do for now, it returns back to the calling application.

Your application extracts info from libcurl about when it would like to get invoked to transfer data or do other work. The most convenient way is to use *curl_multi_wait(3)* that will help you wait until the

application should call libcurl again. The older API to accomplish the same thing is *curl_multi_fdset(3)* that extracts fd_sets from libcurl to use in select() or poll() calls in order to get to know when the transfers in the multi stack might need attention. Both these APIs allow for your program to wait for input on your own private file descriptors at the same time *curl_multi_timeout(3)* also helps you with providing a suitable timeout period for your select() calls.

curl_multi_perform(3) stores the number of still running transfers in one of its input arguments, and by reading that you can figure out when all the transfers in the multi handles are done. 'done' does not mean successful. One or more of the transfers may have failed. Tracking when this number changes, you know when one or more transfers are done.

To get information about completed transfers, to figure out success or not and similar, *curl_multi_info_read(3)* should be called. It can return a message about a current or previous transfer. Repeated invokes of the function get more messages until the message queue is empty. The information you receive there includes an easy handle pointer which you may use to identify which easy handle the information regards.

When a single transfer is completed, the easy handle is still left added to the multi stack. You need to first remove the easy handle with *curl_multi_remove_handle(3)* and then close it with *curl_easy_cleanup(3)*, or possibly set new options to it and add it again with *curl_multi_add_handle(3)* to start another transfer.

When all transfers in the multi stack are done, close the multi handle with *curl_multi_cleanup(3)*. Be careful and please note that you **MUST** invoke separate *curl_easy_cleanup(3)* calls for every single easy handle to clean them up properly.

If you want to re-use an easy handle that was added to the multi handle for transfer, you must first remove it from the multi stack and then re-add it again (possibly after having altered some options at your own choice).

MULTI_SOCKET

curl_multi_socket_action(3) function offers a way for applications to not only avoid being forced to use select(), but it also offers a much more high-performance API that will make a significant difference for applications using large numbers of simultaneous connections.

curl_multi_socket_action(3) is then used instead of *curl_multi_perform(3)*.

When using this API, you add easy handles to the multi handle just as with the normal multi interface. Then you also set two callbacks with the CURLMOPT_SOCKETFUNCTION and CURLMOPT_TIMERFUNCTION options to *curl_multi_setopt(3)*. They are two callback functions that libcurl will call with information about what sockets to wait for, and for what activity, and what the current timeout time is - if that expires libcurl should be notified.

The multi_socket API is designed to inform your application about which sockets libcurl is currently using and for what activities (read and/or write) on those sockets your application is expected to wait for.

Your application must make sure to receive all sockets informed about in the CURLMOPT_SOCKETFUNCTION callback and make sure it reacts on the given activity on them. When a socket has the given activity, you call *curl_multi_socket_action(3)* specifying which socket and action there are.

The CURLMOPT_TIMERFUNCTION callback is called to set a timeout. When that timeout expires, your application should call the *curl_multi_socket_action(3)* function saying it was due to a timeout.

This API is typically used with an event-driven underlying functionality (like libevent, libev, kqueue, epoll or similar) with which the application "subscribes" on socket changes. This allows applications and libcurl to much better scale upward and beyond thousands of simultaneous transfers without losing performance.

When you've added your initial set of handles, you call *curl_multi_socket_action(3)* with CURL_SOCKET_TIMEOUT set in the sockfd argument, and you'll get callbacks call that sets you up and you then continue to call *curl_multi_socket_action(3)* accordingly when you get activity on the sockets you've been asked to wait on, or if the timeout timer expires.

You can poll *curl_multi_info_read(3)* to see if any transfer has completed, as it then has a message saying so.

BLOCKING

A few areas in the code are still using blocking code, even when used from the multi interface. While we certainly want and intend for these to get fixed in the future, you should be aware of the following current restrictions:

- Name resolves unless the c-ares or threaded-resolver backends are used
- NSS SSL connections
- HTTP proxy CONNECT operations
- SOCKS proxy handshakes
- file:// transfers
- TELNET transfers

SEE ALSO

libcurl-errors(3), **libcurl-easy**(3), **libcurl**(3)