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.

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.

Each single transfer is built up with an easy handle. You must create them, and setup the appropriate options for each easy handle, as outlined in the *libcurl(3)* man page, using *curl_easy_setopt(3)*.

When the easy handle is setup for a transfer, then instead of using *curl_easy_perform(3)* (as when using the easy interface for transfers), you should instead add the easy handle to the multi handle using *curl_multi_add_handle(3)*. The multi handle is sometimes referred to as a ´multi stack´ because of the fact that it may hold a large amount of easy handles.

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.

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.

Your application can acquire knowledge from libcurl when it would like to get invoked to transfer data, so that you don't have to busy-loop and call that *curl_multi_perform(3)* like crazy. *curl_multi_fdset(3)* offers an interface using which you can extract 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. This also makes it very easy for your program to wait for input on your own private file descriptors at the same time or perhaps timeout every now and then, should you want that.

A little note here about the return codes from the multi functions, and especially the *curl_multi_perform(3)*: if you receive *CURLM_CALL_MULTI_PERFORM*, this basically means that you should call *curl_multi_perform(3)* again, before you select() on more actions. You don't have to do it immediately, but the return code means that libcurl may have more data available to return or that there may be more data to

send off before it is "satisfied".

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, cleanup the multi handle with *curl_multi_cleanup(3)*. Be careful and please note that you **MUST** invoke separate *curl_easy_cleanup(3)* calls on 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

Since 7.16.0, the *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)*.

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 on non-windows unless c-ares is used
- GnuTLS SSL connections
- NSS SSL connections
- Active FTP connections
- HTTP proxy CONNECT operations
- SOCKS proxy handshakes
- file:// transfers
- TELNET transfers