NAME

curl_multi_perform - reads/writes available data from each easy handle

SYNOPSIS

```
#include <curl/curl.h>

CURLMcode curl_multi_perform(CURLM *multi_handle, int *running_handles);
```

DESCRIPTION

This function handles transfers on all the added handles that need attention in an non-blocking fashion.

When an application has found out there's data available for the multi_handle or a timeout has elapsed, the application should call this function to read/write whatever there is to read or write right now etc. *curl_multi_perform(3)* returns as soon as the reads/writes are done. This function does not require that there actually is any data available for reading or that data can be written, it can be called just in case. It will write the number of handles that still transfer data in the second argument's integer-pointer.

If the amount of *running_handles* is changed from the previous call (or is less than the amount of easy handles you've added to the multi handle), you know that there is one or more transfers less "running". You can then call *curl_multi_info_read(3)* to get information about each individual completed transfer, and that returned info includes CURLcode and more. If an added handle fails very quickly, it may never be counted as a running_handle.

When *running_handles* is set to zero (0) on the return of this function, there is no longer any transfers in progress.

EXAMPLE

```
#ifdef _WIN32
#define SHORT_SLEEP Sleep(100)
#else
#define SHORT_SLEEP usleep(100000)
#endif

fd_set fdread;
fd_set fdwrite;
fd_set fdexcep;
int maxfd = -1;

long curl_timeo;

curl_multi_timeout(multi_handle, &curl_timeo);
if(curl_timeo < 0)
  curl_timeo = 1000;

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

FD_ZERO(&fdread);
FD_ZERO(&fdwrite);
FD_ZERO(&fdexcep);

/* get file descriptors from the transfers */
mc = curl_multi_fdset(multi_handle, &fdread, &fdwrite, &fdexcep, &maxfd);

if(maxfd == -1) {
  SHORT_SLEEP;
```

```
  rc = 0;
}
else
  rc = select(maxfd+1, &fdread, &fdwrite, &fdexcep, &timeout);

switch(rc) {
case -1:
  /* select error */
  break;
case 0:
default:
  /* timeout or readable/writable sockets */
  curl_multi_perform(multi_handle, &still_running);
  break;
}

/* if there are still transfers, loop! */
```

RETURN VALUE

CURLMcode type, general libcurl multi interface error code.

Before version 7.20.0: 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". Do note that *curl_multi_perform(3)* will return *CURLM_CALL_MULTI_PERFORM* only when it wants to be called again **immediately**. When things are fine and there is nothing immediate it wants done, it'll return *CURLM_OK* and you need to wait for "action" and then call this function again.

This function only returns errors etc regarding the whole multi stack. Problems still might have occurred on individual transfers even when this function returns *CURLM_OK*. Use *curl_multi_info_read(3)* to figure out how individual transfers did.

TYPICAL USAGE

Most applications will use *curl_multi_fdset(3)* to get the multi_handle's file descriptors, and *curl_multi_timeout(3)* to get a suitable timeout period, then it'll wait for action on the file descriptors using **select(3)**. As soon as one or more file descriptor is ready, *curl_multi_perform(3)* gets called.

SEE ALSO

curl_multi_cleanup(3), **curl_multi_init**(3), **curl_multi_wait**(3), **curl_multi_fdset**(3), **curl_multi_info_read**(3), **libcurl-errors**(3)