multithreaded/curl_xml.c

```c
#include "curl_xml.h"

/**
 * @brief set options of curl easy handle
 * @param curl_handle CURL*: (pointer to) already-initialized curl easy handle to configure
 * @param ptr RECV_BUF*: (pointer to) user data needed by the curl write call back function
 * @param url const char*: target url to crawl
 * @return a valid CURL handle upon sucess; NULL otherwise
 * note the caller is responsible for cleaning the returned curl handle
 */
CURL *easy_handle_config(CURL *curl_handle, RECV_BUF *ptr, const char *url)
{
    if (ptr == NULL || url == NULL)
    {
        return NULL;
    }

    // init user defined call back function buffer
    if (recv_buf_init(ptr, BUF_SIZE) != 0)
    {
        return NULL;
    }

    // specify URL to get
    curl_easy_setopt(curl_handle, CURLOPT_URL, url);

    // register write call back function to process received data
    curl_easy_setopt(curl_handle, CURLOPT_WRITEFUNCTION, write_cb_curl);
    // user defined data structure passed to the call back function
    curl_easy_setopt(curl_handle, CURLOPT_WRITEDATA, (void *)ptr);

    // register header call back function to process received header data
    curl_easy_setopt(curl_handle, CURLOPT_HEADERFUNCTION, header_cb_curl);
    // user defined data structure passed to the call back function
    curl_easy_setopt(curl_handle, CURLOPT_HEADERDATA, (void *)ptr);

    /// ome servers require a user-agent field
    curl_easy_setopt(curl_handle, CURLOPT_USERAGENT, CURL_USER_AGENT_FIELD);

    // follow HTTP 3XX redirects
    curl_easy_setopt(curl_handle, CURLOPT_FOLLOWLOCATION, 1L);
    // continue to send authentication credentials when following locations
    curl_easy_setopt(curl_handle, CURLOPT_UNRESTRICTED_AUTH, 1L);
    // max numbre of redirects to follow sets to 5
    curl_easy_setopt(curl_handle, CURLOPT_MAXREDIRS, 5L);
    // supports all built-in encodings
    curl_easy_setopt(curl_handle, CURLOPT_ACCEPT_ENCODING, "");

```

```
49      // Enable the cookie engine without reading any initial cookies
50      curl_easy_setopt(curl_handle, CURLOPT_COOKIEFILE, "");
51      // allow whatever auth the proxy speaks
52      curl_easy_setopt(curl_handle, CURLOPT_PROXYAUTH, CURLAUTH_ANY);
53      // allow whatever auth the server speaks
54      curl_easy_setopt(curl_handle, CURLOPT_HTTPAUTH, CURLAUTH_ANY);
55
56      return curl_handle;
57  }
58
59  /**
60   * @brief process a downloaded html page: get all urls from the page and push it onto stack
61   * @param curl_handle CURL*: (pointer to) curl handler that was used to access the url
62   * @param p_recv_buf RECV_BUF*: (pointer to) buffer that contains the received data
63   * @param content_type int*: (pointer to) int to be set with content type code
64   * @param stack STACK*: (pointer to) stack that will be populated with further urls to crawl
65   * @return 0 on success; non-zero otherwise
66   */
67  int process_html(CURL *curl_handle, RECV_BUF *p_recv_buf, int *content_type, STACK *stack)
68  {
69      *content_type = HTML;
70
71      int follow_relative_link = 1;
72      char *url = NULL;
73
74      curl_easy_getinfo(curl_handle, CURLINFO_EFFECTIVE_URL, &url);
75      find_http(p_recv_buf->buf, p_recv_buf->size, follow_relative_link, url, stack);
76      return 0;
77  }
78
79  /**
80   * @brief check if a png is a valid png
81   * @param buf uint8_t*: (pointer to) memory containing the supposed png image
82   * @param n size_t: size of the memory
83   * @return true if memory contains a valid png; false otherwise
84   * @details
85   * The check is derived from the png specification: https://www.w3.org/TR/png/
86   */
87  bool is_png(uint8_t *buf, size_t n)
88  {
89      if (n < 8)
90      {
91          return false;
92      }
93
94      if ((buf[0] == 0x89) &&
95          (buf[1] == 0x50) &&
96          (buf[2] == 0x4E) &&
97          (buf[3] == 0x47) &&
98          (buf[4] == 0x0D) &&
```

```
99          (buf[5] == 0x0A) &&
100         (buf[6] == 0x1A) &&
101         (buf[7] == 0x0A))
102     {
103         return true;
104     }
105
106     return false;
107 }
108
109 /**
110  * @brief process a downloaded png: check if it's a valid png
111  * @param curl_handle CURL*: (pointer to) curl handler that was used to access the url
112  * @param p_recv_buf RECV_BUF*: (pointer to) buffer that contains the received data
113  * @param content_type int*: (pointer to) int to be set with content type code
114  * @return 0 on success; non-zero otherwise
115  */
116 int process_png(CURL *curl_handle, RECV_BUF *p_recv_buf, int *content_type)
117 {
118     // effective url
119     char *eurl = NULL;
120     curl_easy_getinfo(curl_handle, CURLINFO_EFFECTIVE_URL, &eurl);
121
122     if (is_png((uint8_t *)p_recv_buf->buf, p_recv_buf->size))
123     {
124         *content_type = VALID_PNG;
125     }
126     else
127     {
128         *content_type = INVALID_PNG;
129     }
130
131     return 0;
132 }
133
134 /**
135  * @brief process the downloaded content data
136  * @param curl_handle CURL*: (pointer to) curl handler that was used to access the url
137  * @param p_recv_buf RECV_BUF*: (pointer to) buffer that contains the received data
138  * @param content_type int*: (pointer to) int to be set with content type code
139  * @param stack STACK*: (pointer to) stack that will be populated with further urls to crawl
140  * @param response_code_p long*: (pointer to) int to be set with the response code
141  * @return 0 on success; non-zero otherwise
142  * @details
143  * if url points to a HTML page, populate stack with urls linked on the page
144  * if url points to a png, check if it's a valid png
145  * set the content type and response code via the appropriate pointers
146  */
147 int process_data(CURL *curl_handle, RECV_BUF *p_recv_buf, int *content_type, STACK *stack, long
    *response_code_p)
```

```
{
    CURLcode res;

    // get response code, and return if it's a fail
    long response_code;
    res = curl_easy_getinfo(curl_handle, CURLINFO_RESPONSE_CODE, &response_code);
    *response_code_p = response_code;
    if (response_code >= BAD_REQUESTS)
    {
        return 1;
    }

    // get content, and handle differently depending on content type
    char *ct = NULL;
    res = curl_easy_getinfo(curl_handle, CURLINFO_CONTENT_TYPE, &ct);
    if (res != CURLE_OK || ct == NULL)
    {
        return 2;
    }
    if (strstr(ct, CT_HTML))
    {
        return process_html(curl_handle, p_recv_buf, content_type, stack);
    }
    else if (strstr(ct, CT_PNG))
    {
        return process_png(curl_handle, p_recv_buf, content_type);
    }

    return 0;
}

/**
 * @brief crawl specified url and process the downloaded data
 * @param curl_handle CURL*: (pointer to) the curl handler that will be used to process the url
 * @param seed_url char*: string containing the url to crawl
 * @param content_type int*: (pointer to) int to be set with content type code
 * @param stack STACK*: (pointer to) stack that will be populated with further urls to crawl
 * @param response_code_p long*: (pointer to) int to be set with the response code
 * @return 0 on success; non-zero otherwise
 * @details
 * if url points to a HTML page, populate stack with urls linked on the page
 * if url points to a png, check if it's a valid png
 * set the content type and response code via the appropriate pointers
 */
int process_url(CURL *curl_handle, char *seed_url, int *content_type, STACK *stack, long
*response_code_p)
{
    // set default response code to failure (if nothing fails, the code will be set later)
    *response_code_p = INTERNAL_SERVER_ERRORS;

```

```
197     // configure the easy curl handle
198     char url[URL_LENGTH];
199     RECV_BUF recv_buf;
200     strcpy(url, seed_url);
201     curl_handle = easy_handle_config(curl_handle, &recv_buf, url);
202     if (curl_handle == NULL)
203     {
204         fprintf(stderr, "Curl configuration failed. Exiting...\n");
205         curl_global_cleanup();
206         abort();
207     }
208 
209     // process the url
210     CURLcode res;
211     res = curl_easy_perform(curl_handle);
212     if (res != CURLE_OK)
213     {
214         recv_buf_cleanup(&recv_buf);
215         return 1;
216     }
217 
218     // process the data from the url
219     process_data(curl_handle, &recv_buf, content_type, stack, response_code_p);
220 
221     // clean up data buffer
222     recv_buf_cleanup(&recv_buf);
223     return 0;
224 }
225 
226 /**
227  * @brief returns whether the response code is not an error (i.e. okay or redirect)
228  * @param response_code long: response_code in question
229  * @return true if response code is crawlable (not an error); false otherwise
230  */
231 bool is_processable_response(long response_code)
232 {
233     return (response_code >= OK_REQUESTS && response_code <= OK_REQUESTS + CODE_RANGE) ||
    (response_code >= REDIRECT_REQUESTS && response_code <= REDIRECT_REQUESTS + CODE_RANGE);
234 }
235 
236 /**
237  * @brief cURL header call back function to extract image sequence number from
238  *        http header data. An example header for image part n (assume n = 2) is:
239  *        X-Ece252-Fragment: 2
240  * @param p_recv char*: (pointer to) header data delivered by cURL
241  * @param size size_t: number of data elements
242  * @param nmemb size_t: size of one data element
243  * @param userdata void*: (pointer to) buffer containing user-defined data
244  *                        structure used for extracting sequence number
245  * @return total size of header data received
```

```
 * @details
 * cURL documentation: https://curl.se/libcurl/c/CURLOPT_HEADERFUNCTION.html.
 */
size_t header_cb_curl(char *p_recv, size_t size, size_t nmemb, void *userdata)
{
    int realsize = size * nmemb;
    RECV_BUF *p = userdata;

    if (realsize > strlen(ECE252_HEADER) &&
        strncmp(p_recv, ECE252_HEADER, strlen(ECE252_HEADER)) == 0)
    {
        // extract image sequence number
        p->seq = atoi(p_recv + strlen(ECE252_HEADER));
    }
    return realsize;
}

/**
 * @brief cURL write callback function that saves a copy of received data
 * @param p_recv char*: (pointer to) memory that will be populated with received data
 * @param size size_t: number of data elements
 * @param nmemb size_t: size of one data element
 * @param p_userdata void*: (pointer to) user data buffer that can be acccessed outside CURL;
 *                          this will be populated with the web page (or png file)
 * @return total size of data received
 * @details
 * cURL documentation: https://curl.se/libcurl/c/CURLOPT_WRITEFUNCTION.html
 */
size_t write_cb_curl(char *p_recv, size_t size, size_t nmemb, void *p_userdata)
{
    size_t realsize = size * nmemb;
    RECV_BUF *p = (RECV_BUF *)p_userdata;

    if (p->size + realsize + 1 > p->max_size)
    {
        // since received data is not 0 terminated, add one byte for terminating 0
        size_t new_size = p->max_size + max(BUF_INC, realsize + 1);
        char *q = realloc(p->buf, new_size);
        if (q == NULL)
        {
            // out of memory
            perror("realloc");
            return -1;
        }
        p->buf = q;
        p->max_size = new_size;
    }

    // copy data from libcurl into a buffer we can access later
    memcpy(p->buf + p->size, p_recv, realsize);
```

```
296     p->size += realsize;
297     p->buf[p->size] = 0;
298 
299     return realsize;
300 }
301 
302 /**
303  * @brief initialize data structure used for downloading data via cURL
304  * @param ptr RECV_BUF*: a pointer to uninitialized memory
305  * @param max_size size_t: maximum expected size of data to download
306  * @return 0 on success; non-zero otherwise
307  * @details
308  *  note that if the size of downloaded data is larger than the max_size,
309  *  we will reallocate to accommodate
310  */
311 int recv_buf_init(RECV_BUF *ptr, size_t max_size)
312 {
313     void *p = NULL;
314 
315     if (ptr == NULL)
316     {
317         return 1;
318     }
319 
320     p = malloc(max_size);
321     if (p == NULL)
322     {
323         return 2;
324     }
325 
326     ptr->buf = p;
327     ptr->size = 0;
328     ptr->max_size = max_size;
329     // a valid sequence number should be positive
330     ptr->seq = -1;
331     return 0;
332 }
333 
334 /**
335  * @brief clean up data structure used for downloading data via cURL: deallocate memory
336  * @param ptr RECV_BUF*: (pointer to) RECV_BUF to clean
337  * @return 0 on success; non-zero otherwise
338  */
339 int recv_buf_cleanup(RECV_BUF *ptr)
340 {
341     if (ptr == NULL)
342     {
343         return 1;
344     }
345 
```

```
346     if (ptr->buf != NULL)
347     {
348         free(ptr->buf);
349         ptr->buf = NULL;
350     }
351 
352     ptr->size = 0;
353     ptr->max_size = 0;
354     return 0;
355 }
356 
357 /**
358  * @brief clean up all cURL related data structures
359  * @param curl CURL*: (pointer to) curl easy handle to clean up
360  * @param ptr RECV_BUF*: a pointer to data structure used for downloading data via cURL
361  */
362 void cleanup(CURL *curl, RECV_BUF *ptr)
363 {
364     curl_easy_cleanup(curl);
365     curl_global_cleanup();
366     recv_buf_cleanup(ptr);
367 }
368 
369 /**
370  * @brief get html document from data
371  * @param buf char*: (pointer to) memory containing document data
372  * @param size int: size of data
373  * @param url char*: url string of the html document
374  * @return document pointer if successful; NULL otherwise
375  */
376 htmlDocPtr mem_getdoc(char *buf, int size, const char *url)
377 {
378     int opts = HTML_PARSE_NOBLANKS | HTML_PARSE_NOERROR |
379                HTML_PARSE_NOWARNING | HTML_PARSE_NONET;
380     htmlDocPtr doc = htmlReadMemory(buf, size, url, NULL, opts);
381 
382     if (doc == NULL)
383     {
384         printf("%s\n", url);
385         fprintf(stderr, "Document not parsed successfully.\n");
386         return NULL;
387     }
388 
389     return doc;
390 }
391 
392 /**
393  * @brief get nodes on the xml page
394  * @param doc xmlDocPtr: xml document to get nodes from
395  * @param xpath xmlChar*: xpath
```

```
 * @return nodes on success; NULL otherwise
 */
xmlXPathObjectPtr getnodeset(xmlDocPtr doc, xmlChar *xpath)
{
    xmlXPathContextPtr context;
    xmlXPathObjectPtr result;

    context = xmlXPathNewContext(doc);
    if (context == NULL)
    {
        printf("Error in xmlXPathNewContext\n");
        return NULL;
    }
    result = xmlXPathEvalExpression(xpath, context);
    xmlXPathFreeContext(context);
    if (result == NULL)
    {
        printf("Error in xmlXPathEvalExpression\n");
        return NULL;
    }
    if (xmlXPathNodeSetIsEmpty(result->nodesetval))
    {
        xmlXPathFreeObject(result);
        printf("No result\n");
        return NULL;
    }
    return result;
}

/**
 * @brief get all urls on the xml web page and push them onto stack
 * @param buf char*: (pointer to) buffer that contains the HTML web page
 * @param size int: size of the buffer
 * @param follow_relative_links int: 1 if we're following relative links and 0 otherwise
 * @param base_url const char*: base url of the page
 * @param stack STACK*: (pointer to) stack that will be populated with further urls to crawl
 * @return 0 on success; non-zero otherwise
 */
int find_http(char *buf, int size, int follow_relative_links, const char *base_url, STACK
*stack)
{
    int i;
    htmlDocPtr doc;
    xmlChar *xpath = (xmlChar *)"//a/@href";
    xmlNodeSetPtr nodeset;
    xmlXPathObjectPtr result;
    xmlChar *href;

    if (buf == NULL)
    {
```

```
445         return 1;
446     }
447 
448     doc = mem_getdoc(buf, size, base_url);
449 
450     result = getnodeset(doc, xpath);
451     if (result)
452     {
453         nodeset = result->nodesetval;
454         for (i = 0; i < nodeset->nodeNr; i++)
455         {
456             href = xmlNodeListGetString(doc, nodeset->nodeTab[i]->xmlChildrenNode, 1);
457             if (follow_relative_links)
458             {
459                 xmlChar *old = href;
460                 href = xmlBuildURI(href, (xmlChar *)base_url);
461                 xmlFree(old);
462             }
463             if (href != NULL && !strncmp((const char *)href, "http", 4))
464             {
465                 char *temp = malloc((strlen((char *)href) + 1) * sizeof(char));
466                 memset(temp, 0, (strlen((char *)href) + 1) * sizeof(char));
467                 sprintf(temp, "%s", href);
468                 push_stack(stack, (char *)temp);
469                 free(temp);
470                 temp = NULL;
471             }
472             xmlFree(href);
473         }
474         xmlXPathFreeObject(result);
475     }
476 
477     xmlFreeDoc(doc);
478 
479     return 0;
480 }
```