Multi-threaded Web Crawler Built with C

Possible Files of Interest for Code Sample Review

- `findpng2.c` is the main driver program and is about 350 lines of code
- `stack.c` and `hash.c` (300 lines combined) define two memory-safe data structures
- Please feel free to look at any of the files you feel are appropriate. I included descriptions of each file below.
- Please also feel free to build using `make`

Project context

Purpose

Starting from a seed URL, recursively crawl to all other URLs linked on the web page of the seed URL, searching for PNGs. Stop when we find a specified number of PNGs or run out of URLs to visit. The program outputs all the pngs found in a `png_urls.txt` file at the end of the program. The user can also specify to output all the unique URLs the program crawled to a log file.

To speed up the search, the program launches multiple threads. The user may specify the number of runner threads that the program will launch. Each thread can process a different URL concurrent with other threads. The user may also specify the number of PNGs to find before stopping.

We use multiple threads primarily because network downloads can be slow compared to the CPU. Each thread synchronously waits for the network download of a page. The CPU can process other threads while this thread is blocked on the network download. If the program runs on one thread, the program cannot make progress while the one thread waits for its network download to complete.

Files

- `findpng2.c` : main driver program
  - takes input
  - runs the specified number of threads
  - the threads perform the crawl
  - handles synchronization between threads
  - handles clean-up of threads at the end of the program
- `stack.c` :
  - a memory-safe dynamic stack that holds strings
  - used for holding URLs to crawl (the `frontier`)
- `p_stack.c` :
  - a memory-safe dynamic stack that holds pointers
  - used in `hash.c` for holding pointers to memory for later deallocation
- `hash.c` :
  - a memory-safe hash set that holds strings as keys
  - used for holding visited URLs to prevent cycles in the crawling process
- `curl_xml.c` :
  - utility functions for downloading web pages using cURL
  - utility functions for processing XML pages using libxml
  - utility function for checking if a png file is a valid png
  - used for downloading web pages, searching for URLs listed on the pages, and checking the validity of found pngs

External libraries used

- cURL (https://curl.se/libcurl/)
- libxml (https://github.com/GNOME/libxml2)

Notes

- I wrote this program for the course, ECE 252 (Systems Programming and Concurrency).
  - I received 100% on this project (marked based on functionality, speed, lack of deadlocks, and proper memory management).
  - I received 100% on the course (60% final exam, 40% projects).
- The program is named `findpng2` because I have a few other versions that perform the same task with different techniques (e.g. using processes and interprocess communication for synchronization; or using asynchronous cURL).

Building

- run `make` in this directory

Usage

`findpng2 [OPTION]... [ROOT_URL]`

- options:
  - -t=NUM - the program will create NUM threads to crawl the web (default: 1)
  - -m=NUM - the program will find up to NUM unique PNG URLs (default: 50)
  - -v=LOGFILE - if specified, program will log the unique URLs visited in a file named LOGFILE
- output:
  - on terminal, `findpng2 execution time: S seconds`

    - the program will create a `png_urls.txt` file containing all the valid PNG URLs found
    - if a log file is specified, the program will create a `<LOGFILE>` file containing all unique URLs visited
- for example, `./findpng2 -t 1 -m 1 -v test.txt https://www.cleanpng.com/static/img/logo.png` will launch 1 thread to find 1 png starting from the URL `https://www.cleanpng.com/static/img/logo.png` and output all visited URLs into `test.txt`. Note that since the seed URL is a png itself, the crawl will end immediately after the first visit.

Code Best Practice Notes

- `findpng2.c` uses global variables, which is not good practice; however, the course instructors recommended them for simplicity for the sake of the small project.
    - I could instead pass to the thread runner a pointer to a struct containing all the global variables.
- `findpng2.c` 's thread runner function exits with a `return` rather than `pthread_exit`, which is the recommended way to exit a thread. However, valgrind will report a small number of reachable blocks for threads that use `pthread_exit`. This is not a real issue in most applications. However, since a part of the marking scheme for this project was to achieve 0 reachable blocks at the end of program, my only option was to use a `return`.

Algorithm overview

findpng2.c

Global data structures

- `frontier`: a stack of discovered URLs that we have yet to process, shared by all threads
- `pngs`: a stack of all pngs found so far
- `visited`: a hash set of URLs we have visited (so we don't crawl repeat URLs)

Synchronization

- `frontier_mutex`: a lock for accessing `frontier` and other accounting variables related to the state of the crawl (whether if the crawl is done; number of threads waiting for the frontier to be non-empty; number of threads processing a URL)
    - used for the `frontier_empty` condition variable as well
- `pngs_mutex`: a lock for accessing `pngs`
- `visited_mutex`: a lock for accessing `visited`
- `frontier_empty`: a condition variable that threads will wait on when `frontier` is empty
    - when another thread adds to `frontier`, it will broadcast to wake up the sleeping threads
    - alternatively, a thread may broadcast when the program is finished (no more URLs we can recursively crawl or we have found `num_pngs_to_find` pngs) so that sleeping threads can wake up and exit

Thread runner

- The main function launches `t` number of these thread runners.
- The runner does the following in an infinite loop.
    - With lock `frontier_mutex`:
        - Check if our crawl is finished. If so, signal all threads to exit.
            - If `frontier` is empty and there are no threads currently processing a URL, we don't and won't have any more URLs to crawl: our crawl is finished.
            - If our crawl is finished, broadcast `frontier_empty` to wake up sleeping threads so they can exit.
        - Wait until there is a URL to crawl or our crawl is finished.
            - The thread does this by waiting on the condition variable `frontier_empty`.
            - The signal to wake up will come from a thread pushing a URL to `frontier` or a thread signalling our crawl is finished.
        - If our crawl is finished, exit loop.
        - Else, grab the first URL on `frontier` for processing.
        - With lock `visited_mutex`:
            - Check if we've already visited the URL.
            - If we have, don't progress further and return to the start of the loop.
            - If we haven't, add it to the `visited` hash set and continue on.
    - Download the URL's contents
        - If it is a HTML file, grab all URLs that it links to.
        - If it is a PNG file, determine if it is a valid png.
    - (If HTML file) With lock `frontier_mutex`:
        - Push all URLs found onto `frontier`.
        - If there are any sleeping threads waiting for a non-empty frontier, broadcast on `frontier_empty`.
    - (If PNG file) With lock `pngs_mutex`:
        - Push URL into `pngs`.
        - If we hit `num_pngs_to_find`:
            - With lock `frontier_mutex`:
                - Set a global variable indicating our crawl is finished, and broadcast on `frontier_empty`.
- When the runner exits the infinite loop, clean up any data structures used (memory deallocation, network libraries).

stack.c and p_stack.c

- Memory-safe stacks
- Resizing is done by allocating a larger chunk of memory, moving the old items over, and deallocating the old memory.

hash.c

- A hash set that holds strings as keys

Data structures

- `hmap`: `hsearch_data` struct used as the underlying hashmap
- `elements`: an array of pointers to keys currently in the hash set
  - used for linear access to keys, moving keys over on resize, and for deallocating the keys at destruction
- `ps`: a stack of pointers to keys that were used when searching `hmap`
  - used for deallocating the strings at destruction
  - note we don't deallocate the keys/strings immediately after the search since `hsearch_data` may hold on to the pointer to the key and use it in future operations after the search

Adding a key

- If full, resize.
- Create a copy of the passed-in key/string.
- Add the key (copy) to `hmap`.
- Add the key (copy) to `elements`.

Searching for a string

- Create a copy of the passed-in key/string.
- Search for the key (copy) in `hmap`.
- If we found the key:
  - Add the key (copy) to `ps`.
- Else:
  - Free memory for the key (copy).
- Why we add the key to `ps`:
  - If the key is found in `hmap`, `hmap` may use the memory of the passed-in string for future references instead of the string used when the key was initially added to `hmap`.
  - `hsearch` does not deallocate memory upon destruction. Thus, we keep track of `ps`: the strings to deallocate at destruction.

Resizing

- Destroy and reinitialize `hmap` with a larger size.
- Allocate a larger `elements`.
- Transfer keys from the old `elements` to the new `hmap` and `elements`.
- Deallocate old `elements`.

multithreaded/findpng2.c

```c
#include "findpng2.h"

// I defined this for the code sample review
//  so if you want to make and run the program,
//  you can see the urls being crawled on command line
#define DEBUG_URL_PRINT

/* -- Global Variables -- */
// global collection of urls to be crawled by runner threads
STACK *frontier;
// pngs found (png urls)
STACK *pngs;
// urls visited
HSET *visited;
// whether we are done with the entire crawl
bool done;
// number of threads waiting for a non-empty frontier
size_t num_waiting_on_url;
// number of thread runners currently processing a url
size_t num_running;
// number of pngs to find before stopping
int num_pngs_to_find;
/* ----------------- */

/* -- Synchronization --*/
// condition variable for threads to wait on when the frontier is empty
pthread_cond_t frontier_empty;
// lock for frontier, done, num_waiting_on_url, and num_running;
//  also used for frontier_empty
pthread_mutex_t frontier_mutex;
// lock for pngs stack
pthread_mutex_t pngs_mutex;
// lock for visited hash set
pthread_mutex_t visited_mutex;
/* ----------------- */

/**
 * @brief initialize global variables and synchronization variables
 */
void initialize_global()
{
    frontier = malloc(sizeof(STACK));
    memset(frontier, 0, sizeof(STACK));
    init_stack(frontier, STACK_SIZE);

    visited = malloc(sizeof(HSET));
    memset(visited, 0, sizeof(HSET));
    init_hset(visited, HMAP_SIZE);
```

```
49
50      pngs = malloc(sizeof(STACK));
51      memset(pngs, 0, sizeof(STACK));
52      init_stack(pngs, STACK_SIZE);
53
54      done = false;
55      num_waiting_on_url = 0;
56      num_running = 0;
57
58      pthread_cond_init(&frontier_empty, NULL);
59      pthread_mutex_init(&frontier_mutex, NULL);
60      pthread_mutex_init(&visited_mutex, NULL);
61      pthread_mutex_init(&pngs_mutex, NULL);
62  }
63
64  /**
65   * @brief cleanup global variables and synchronization variables
66   */
67  void cleanup_global()
68  {
69      cleanup_stack(frontier);
70      free(frontier);
71      frontier = NULL;
72
73      cleanup_hset(visited);
74      free(visited);
75      visited = NULL;
76
77      cleanup_stack(pngs);
78      free(pngs);
79      pngs = NULL;
80
81      pthread_cond_destroy(&frontier_empty);
82      pthread_mutex_destroy(&frontier_mutex);
83      pthread_mutex_destroy(&pngs_mutex);
84      pthread_mutex_destroy(&visited_mutex);
85  }
86
87  /**
88   * @brief runner function that crawls urls in the global frontier
89   * @param _ void*: not used; only defined to satisfy thread API
90   * @return NULL
91   * @details
92   * Any number of runner threads can be started.
93   * The runner function manages concurrency.
94   * The runner function assumes that all global structures and
95   *  synchronization variables are initialized.
96   * The runner function will stop once there are no more urls to crawl
97   *  or when we've found num_pngs_to_find pngs.
98   * The runner function does not clean up global variables.
```

```
 */
void *runner(void *_)
{
    /* -- Initialize cURL easy handle -- */
    CURL *curl_handle = curl_easy_init();
    if (curl_handle == NULL)
    {
        fprintf(stderr, "curl_easy_init: returned NULL\n");
        exit(1);
    }
    /* ----------------- */

    /* -- Defining variables used in the loop -- */
    // response code from accessing url
    long response_code;
    // content type of data at url (e.g. HTML, PNG)
    int content_type = DEFAULT_TYPE;
    // the current url the thread is crawling
    char *url_to_crawl = NULL;
    // urls found on the web page visited; we will add these to the frontier
    STACK *urls_found = NULL;
    // if we have cleaned urls_found
    bool cleaned_urls_found = false;
    /* ----------------- */

    while (true)
    {
        /* -- Cleanup structures from last iteration and re-initialize -- */
        if (urls_found != NULL)
        {
            if (!cleaned_urls_found)
            {
                cleanup_stack(urls_found);
                cleaned_urls_found = true;
            }
            free(urls_found);
        }
        urls_found = malloc(sizeof(STACK));
        memset(urls_found, 0, sizeof(STACK));
        init_stack(urls_found, 1);
        cleaned_urls_found = false;

        if (url_to_crawl != NULL)
        {
            free(url_to_crawl);
            url_to_crawl = NULL;
        }
        /* ----------------- */

        /* -- Check status of frontier and overall crawl -- */
```

```
149         pthread_mutex_lock(&frontier_mutex);
150         {
151             // If the crawl is finished, signal sleeping threads to
152             //  wake up so they can exit
153             if (is_empty_stack(frontier) && num_running == 0)
154             {
155                 done = true;
156                 if (num_waiting_on_url > 0)
157                 {
158                     pthread_cond_broadcast(&frontier_empty);
159                 }
160             }
161 
162             // If there are no urls to crawl and the crawl is not done, wait
163             while (is_empty_stack(frontier) && !done)
164             {
165                 ++num_waiting_on_url;
166                 pthread_cond_wait(&frontier_empty, &frontier_mutex);
167                 --num_waiting_on_url;
168             }
169 
170             // If the crawl is finished, exit the loop
171             if (done)
172             {
173                 pthread_mutex_unlock(&frontier_mutex);
174                 break;
175             }
176 
177             // Take the top url on the frontier
178             pop_stack(frontier, &url_to_crawl);
179 
180             // Check if the url has been visited
181             pthread_mutex_lock(&visited_mutex);
182             {
183                 // If the url has been visited, go back to the top of the loop
184                 //  (go to the next url in the frontier or if frontier is empty, wait)
185                 if (search_hset(visited, url_to_crawl) == 1)
186                 {
187                     pthread_mutex_unlock(&visited_mutex);
188                     pthread_mutex_unlock(&frontier_mutex);
189                     continue;
190                 }
191                 // If the url has not been visited, mark it as visited.
192                 //  The thread will now process the url.
193                 else
194                 {
195                     add_hset(visited, url_to_crawl);
196                 }
197             }
198             pthread_mutex_unlock(&visited_mutex);
```

```
199             ++num_running;
200         }
201         pthread_mutex_unlock(&frontier_mutex);
202         /* ---------------- */
203 
204 #ifdef DEBUG_URL_PRINT
205         printf("URL: %s\n", url_to_crawl);
206 #endif
207 
208         /* -- Crawl the url -- */
209         // download the contents at the url and process it
210         process_url(curl_handle, url_to_crawl, &content_type, urls_found, &response_code);
211         /* ---------------- */
212 
213         /* -- Process url based on its contents -- */
214         if (is_processable_response(response_code))
215         {
216             // If the url was a HTML page, add all urls on that page to the frontier
217             if (content_type == HTML)
218             {
219                 char *url_in_html = NULL;
220                 while (pop_stack(urls_found, &url_in_html) == 0)
221                 {
222                     // Add to the frontier and signal sleeping threads
223                     //  (that a url is ready in frontier)
224                     pthread_mutex_lock(&frontier_mutex);
225                     {
226                         push_stack(frontier, url_in_html);
227                         if (num_waiting_on_url > 0)
228                         {
229                             pthread_cond_broadcast(&frontier_empty);
230                         }
231                     }
232                     pthread_mutex_unlock(&frontier_mutex);
233                     free(url_in_html);
234                     url_in_html = NULL;
235                 }
236             }
237             // If the url was a valid PNG, add it to our collection of found pngs
238             else if (content_type == VALID_PNG)
239             {
240                 pthread_mutex_lock(&pngs_mutex);
241                 {
242                     push_stack(pngs, url_to_crawl);
243                     // If we've reached the maximum number of PNGs we want to find,
244                     //  end the program
245                     if (num_elements_stack(pngs) >= num_pngs_to_find)
246                     {
247                         pthread_mutex_lock(&frontier_mutex);
248                         {
```

```
                            done = true;
                            pthread_cond_broadcast(&frontier_empty);
                        }
                        pthread_mutex_unlock(&frontier_mutex);
                    }
                }
                pthread_mutex_unlock(&pngs_mutex);
            }
        }
        /* ----------------- */

        /* -- The thread is no longer processing a url -- */
        pthread_mutex_lock(&frontier_mutex);
        {
            --num_running;
        }
        pthread_mutex_unlock(&frontier_mutex);
        /* ----------------- */
    }

    /* -- The thread is done all processing: clean up -- */
    if (urls_found != NULL)
    {
        if (!cleaned_urls_found)
        {
            cleanup_stack(urls_found);
        }
        free(urls_found);
    }

    if (url_to_crawl != NULL)
    {
        free(url_to_crawl);
        url_to_crawl = NULL;
    }

    curl_easy_cleanup(curl_handle);
    /* ----------------- */

    return NULL;
}

int main(int argc, char **argv)
{
    /* -- command line inputs -- */
    char *seed_url;
    char *logfile = NULL;
    size_t t = 1;
    num_pngs_to_find = 50;

```

```
299     if (argc == 1)
300     {
301         printf("Usage: ./findpng2 OPTION[-t=<NUM> -m=<NUM> -v=<LOGFILE>] SEED_URL\n");
302         return -1;
303     }
304 
305     seed_url = argv[argc - 1];
306 
307     int c;
308     char *str = "option requires an argument";
309 
310     while ((c = getopt(argc, argv, "t:m:v:")) != -1)
311     {
312         switch (c)
313         {
314         case 't':
315             if (optarg == NULL)
316             {
317                 t = 1;
318                 break;
319             }
320             t = strtoul(optarg, NULL, 10);
321             if (t <= 0)
322             {
323                 fprintf(stderr, "%s: %s > 0 -- 't'\n", argv[0], str);
324                 return -1;
325             }
326             break;
327         case 'm':
328             if (optarg == NULL)
329             {
330                 num_pngs_to_find = 50;
331                 break;
332             }
333             num_pngs_to_find = atoi(optarg);
334             if (num_pngs_to_find < 0)
335             {
336                 fprintf(stderr, "%s: %s >= 0 -- 'm'\n", argv[0], str);
337                 return -1;
338             }
339             break;
340         case 'v':
341             if (optarg == NULL)
342             {
343                 logfile = NULL;
344                 break;
345             }
346             logfile = malloc(sizeof(char) * FILE_PATH_SIZE);
347             memset(logfile, 0, sizeof(char) * FILE_PATH_SIZE);
348             strcpy(logfile, optarg);
```

```
            break;
        }
    }
    /* ---------------- */

    /* -- initialize global variables and synchronization variables -- */
    initialize_global();
    /* ---------------- */

    /* -- CURL global init -- */
    curl_global_init(CURL_GLOBAL_DEFAULT);
    /* ---------------- */

    /* -- Initialize XML Parser -- */
    xmlInitParser();
    /* ---------------- */

    /* -- Put the seed URL in the frontier -- */
    push_stack(frontier, seed_url);
    /* ---------------- */

    /* -- Record time to be used for measuring speed -- */
    double times[2];
    struct timeval tv;
    if (gettimeofday(&tv, NULL) != 0)
    {
        perror("gettimeofday");
        exit(1);
    }
    times[0] = (tv.tv_sec) + tv.tv_usec / 1000000.;
    /* ---------------- */

    /* -- Create threads -- */
    pthread_t *runners = malloc(t * sizeof(pthread_t));
    memset(runners, 0, sizeof(pthread_t) * t);
    if (runners == NULL)
    {
        perror("malloc\n");
        exit(-1);
    }
    for (int i = 0; i < t; ++i)
    {
        pthread_create(&runners[i], NULL, runner, NULL);
    }
    /* ---------------- */

    /* -- Wait for threads to finish -- */
    for (int i = 0; i < t; ++i)
    {
        pthread_join(runners[i], NULL);
```

```
399     }
400     /* ----------------- */
401
402     /* -- Write to files -- */
403     // Write png urls
404     FILE *fpngs = fopen("./png_urls.txt", "w+");
405     if (fpngs == NULL)
406     {
407         fprintf(stderr, "Opening png file for write failed\n");
408         exit(1);
409     }
410     char *temp = NULL;
411     while (pop_stack(pngs, &temp) == 0)
412     {
413         fprintf(fpngs, "%s\n", temp);
414         free(temp);
415     }
416     fclose(fpngs);
417
418     // Write all urls visited into a log file if user desires
419     if (logfile != NULL)
420     {
421         char *logfile_name = malloc(sizeof(char) * FILE_PATH_SIZE);
422         memset(logfile_name, 0, sizeof(char) * FILE_PATH_SIZE);
423         sprintf(logfile_name, "./%s", logfile);
424         FILE *flogs = fopen(logfile_name, "w+");
425         free(logfile_name);
426         if (flogs == NULL)
427         {
428             fprintf(stderr, "Opening log file for write failed\n");
429             exit(1);
430         }
431         temp = NULL;
432         for (size_t i = 0; i < visited->cur_size; ++i)
433         {
434             fprintf(flogs, "%s\n", visited->elements[i]);
435             free(temp);
436         }
437         fclose(flogs);
438     }
439     free(logfile);
440     /* ----------------- */
441
442     /* -- Cleanup global variables and synchronization variables -- */
443     cleanup_global();
444     /* ----------------- */
445
446     /* -- Free threads -- */
447     free(runners);
448     /* ----------------- */
```

```
449
450     /* -- Clean up libraries used -- */
451     curl_global_cleanup();
452     xmlCleanupParser();
453     /* ----------------- */
454
455     /* -- Print time it took for crawl from the seed url -- */
456     if (gettimeofday(&tv, NULL) != 0)
457     {
458         perror("gettimeofday");
459         exit(1);
460     }
461     times[1] = (tv.tv_sec) + tv.tv_usec / 1000000.;
462     printf("findpng2 execution time: %.6lf seconds\n", times[1] - times[0]);
463     /* ----------------- */
464
465     return 0;
466 }
```

multithreaded/findpng2.h

```h
#include <stdbool.h>
#include "curl_xml.h"
#include "hash.h"
#include <pthread.h>

#define URL_SIZE 512
#define FILE_PATH_SIZE 512
#define STACK_SIZE 1024
#define HMAP_SIZE 1024

void *runner(void *args);
```

multithreaded/stack.c

```c
/*
A dynamic stack holding strings
- note: this current implementation only resizes stack to increase size, never to decrease size
*/

#include "stack.h"

/**
 * @brief initialize stack with an initial size (capacity)
 * @param p STACK*: a pointer to uninitialized memory
 * @param stack_size size_t: initial capacity of the stack to be initialized
 * @return 0 on success; 1 otherwise
 */
int init_stack(STACK *p, size_t stack_size)
{
    if (p == NULL || stack_size == 0)
    {
        return 1;
    }

    p->size = stack_size;
    p->pos = -1;
    p->items = (char **)malloc(stack_size * sizeof(char *));

    memset(p->items, 0, stack_size * sizeof(char *));

    return 0;
}

/**
 * @brief push an item onto the stack; if the stack is full, resize the stack
 * @param p STACK*: (pointer to) the stack the function will push item onto
 * @param item char*: string to push onto stack
 * @return 0 on success; 1 otherwise
 */
int push_stack(STACK *p, char *item)
{
    if (p == NULL)
    {
        return 1;
    }

    if (is_full_stack(p))
    {
        resize_stack(p);
    }

    ++(p->pos);
```

```c
    // strlen(item) + 1 for end of string character
    p->items[p->pos] = malloc((strlen(item) + 1) * sizeof(char));
    memset(p->items[p->pos], 0, (strlen(item) + 1) * sizeof(char));
    strncpy(p->items[p->pos], item, strlen(item));

    return 0;
}

/**
 * @brief pop from the stack
 * @param p STACK*: (pointer to) the stack the function will pop from
 * @param p_item char**: pointer that will be populated with popped element
 * @return 0 on success; 1 otherwise
 * @note the caller is responsible for deallocating memory assigned to p_item
 */
int pop_stack(STACK *p, char **p_item)
{
    if ((p == NULL) || is_empty_stack(p))
    {
        return 1;
    }

    *p_item = malloc(sizeof(char) * (strlen(p->items[p->pos]) + 1));
    memset(*p_item, 0, sizeof(char) * (strlen(p->items[p->pos]) + 1));
    strncpy(*p_item, p->items[p->pos], strlen(p->items[p->pos]));
    free(p->items[p->pos]);
    p->items[p->pos] = NULL;
    (p->pos)--;
    return 0;
}

/**
 * @brief check if the stack is full
 * @param p STACK*: (pointer to) the stack to check
 * @return true if full; false otherwise
 */
bool is_full_stack(STACK *p)
{
    if (p == NULL)
    {
        return 0;
    }
    return (p->pos == (p->size - 1));
}

/**
 * @brief check if the stack is empty
 * @param p STACK*: (pointer to) the stack to check
 * @return true if empty; false otherwise
 */
```

```c
bool is_empty_stack(STACK *p)
{
    if (p == NULL)
    {
        return 0;
    }
    return (p->pos == -1);
}

/**
 * @brief resize stack to have greater capacity; maintain existing elements
 * @param p STACK*: (pointer to) the stack to resize
 * @return 0 on success; 1 otherwise
 */
int resize_stack(STACK *p)
{
    size_t old_size = p->size;
    char **old_items = p->items;
    p->size = (p->size) * STACK_RESIZE_FACTOR;
    p->items = (char **)malloc((p->size) * sizeof(char *));
    if (p->items == NULL)
    {
        return 1;
    }

    for (size_t i = 0; i < old_size; ++i)
    {
        p->items[i] = old_items[i];
    }
    for (size_t i = old_size; i < p->size; ++i)
    {
        p->items[i] = NULL;
    }

    free(old_items);
    old_items = NULL;

    return 0;
}

/**
 * @brief returns number of elements currently in the stack
 * @param p STACK*: (pointer to) the stack
 * @return number of elements in the stack
 */
size_t num_elements_stack(STACK *p)
{
    return p->pos + 1;
}

```

```c
/**
 * @brief deconstruct stack: free all allocated memory
 * @param p STACK*: (pointer to) the stack to deconstruct
 * @return 0 on success; 1 otherwise
 */
int cleanup_stack(STACK *p)
{
    if (p == NULL || p->items == NULL)
    {
        return 0;
    }
    for (size_t i = 0; i < p->size; ++i)
    {
        if (p->items[i] != NULL)
        {
            free(p->items[i]);
            p->items[i] = NULL;
        }
    }
    free(p->items);
    p->items = NULL;
    return 0;
}
```

multithreaded/stack.h

```c
/*
A dynamic stack holding strings
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct stack
{
    // max capacity of the stack
    size_t size;
    // position of the last item pushed onto the stack
    size_t pos;
    // memory for stack
    char **items;
} STACK;

#define STACK_RESIZE_FACTOR 2

int init_stack(STACK *p, size_t stack_size);
bool is_full_stack(STACK *p);
bool is_empty_stack(STACK *p);
int push_stack(STACK *p, char *item);
int pop_stack(STACK *p, char **p_item);
int resize_stack(STACK *p);
size_t num_elements_stack(STACK *p);
int cleanup_stack(STACK *p);
```

multithreaded/p_stack.c

```c
/*
A dynamic stack holding pointers
- note: this current implementation only resizes stack to increase size, never to decrease size
*/

#include "p_stack.h"

/**
 * @brief initialize stack with an initial size (capacity)
 * @param p PSTACK*: a pointer to uninitialized memory
 * @param stack_size size_t: initial capacity of the stack to be initialized
 * @return 0 on success; 1 otherwise
 */
int init_pstack(PSTACK *p, size_t stack_size)
{
    if (p == NULL || stack_size == 0)
    {
        return 1;
    }

    p->size = stack_size;
    p->pos = -1;
    p->items = (void **)malloc(stack_size * sizeof(void *));

    memset(p->items, 0, stack_size * sizeof(void *));

    return 0;
}

/**
 * @brief push an item onto the stack; if the stack is full, resize the stack
 * @param p PSTACK*: (pointer to) the stack the function will push item onto
 * @param item void*: pointer to push onto stack
 * @return 0 on success; 1 otherwise
 */
int push_pstack(PSTACK *p, void *item)
{
    if (p == NULL)
    {
        return 1;
    }

    if (is_full_pstack(p))
    {
        resize_pstack(p);
    }

    ++(p->pos);
```

```c
    p->items[p->pos] = item;

    return 0;
}

/**
 * @brief pop from the stack
 * @param p PSTACK*: (pointer to) the stack the function will pop from
 * @param p_item void**: pointer that will be populated with popped element (which itself is a pointer)
 * @return 0 on success; 1 otherwise
 */
int pop_pstack(PSTACK *p, void **p_item)
{
    if (p == NULL || is_empty_pstack(p))
    {
        return 1;
    }

    *p_item = p->items[p->pos];
    p->items[p->pos] = NULL;
    (p->pos)--;
    return 0;
}

/**
 * @brief check if the stack is full
 * @param p PSTACK*: (pointer to) the stack to check
 * @return true if full; false otherwise
 */
bool is_full_pstack(PSTACK *p)
{
    if (p == NULL)
    {
        return 0;
    }
    return (p->pos == (p->size - 1));
}

/**
 * @brief check if the stack is empty
 * @param p PSTACK*: (pointer to) the stack to check
 * @return true if empty; false otherwise
 */
bool is_empty_pstack(PSTACK *p)
{
    if (p == NULL)
    {
        return 0;
    }
```

```
98      return (p->pos == -1);
99  }
100
101 /**
102  * @brief resize stack to have greater capacity; maintain existing elements
103  * @param p PSTACK*: (pointer to) the stack to resize
104  * @return 0 on success; 1 otherwise
105  */
106 int resize_pstack(PSTACK *p)
107 {
108     size_t old_size = p->size;
109     void **old_items = p->items;
110     p->size = (p->size) * PSTACK_RESIZE_FACTOR;
111     p->items = (void **)malloc((p->size) * sizeof(void *));
112     if (p->items == NULL)
113     {
114         return 1;
115     }
116
117     for (size_t i = 0; i < old_size; ++i)
118     {
119         p->items[i] = old_items[i];
120     }
121     for (size_t i = old_size; i < p->size; ++i)
122     {
123         p->items[i] = NULL;
124     }
125
126     free(old_items);
127     old_items = NULL;
128
129     return 0;
130 }
131
132 /**
133  * @brief returns number of elements currently in the stack
134  * @param p STACK*: (pointer to) the stack
135  * @return number of elements in the stack
136  */
137 size_t num_elements_pstack(PSTACK *p)
138 {
139     return p->pos + 1;
140 }
141
142 /**
143  * @brief deconstruct stack: free all allocated memory
144  * @param p PSTACK*: (pointer to) the stack to deconstruct
145  * @return 0 on success; 1 otherwise
146  */
147 int cleanup_pstack(PSTACK *p)
```

```
148 {
149     if (p == NULL || p->items == NULL)
150     {
151         return 0;
152     }
153     for (size_t i = 0; i < p->size; ++i)
154     {
155         if (p->items[i] != NULL)
156         {
157             free(p->items[i]);
158             p->items[i] = NULL;
159         }
160     }
161     free(p->items);
162     p->items = NULL;
163     return 0;
164 }
```

multithreaded/p_stack.h

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct p_stack_struct
{
    // max capacity of the stack
    size_t size;
    // position of the last item pushed onto the stack
    size_t pos;
    // memory for stack
    void **items;
} PSTACK;

#define PSTACK_RESIZE_FACTOR 2

int init_pstack(PSTACK *p, size_t PSTACK_size);
bool is_full_pstack(PSTACK *p);
bool is_empty_pstack(PSTACK *p);
int push_pstack(PSTACK *p, void *item);
int pop_pstack(PSTACK *p, void **p_item);
int resize_pstack(PSTACK *p);
size_t num_elements_pstack(PSTACK *p);
int cleanup_pstack(PSTACK *p);
```

multithreaded/hash.c

```c
/*
A hash set with strings as keys
- uses hsearch as the underlying hashmap
*/

#include "hash.h"

/**
 * @brief initialize hash set with an initial size (capacity)
 * @param p HSET*: a pointer to uninitialized memory
 * @param set_size size_t: initial capacity of the hash set to be initialized
 * @return 0 on success; 1 otherwise
 */
int init_hset(HSET *p, size_t set_size)
{
    // create hsearch hash map
    p->hmap = malloc(sizeof(struct hsearch_data));
    memset(p->hmap, 0, sizeof(struct hsearch_data));
    if (hcreate_r(set_size, p->hmap) == 0)
    {
        perror("hcreate\n");
        return 1;
    }

    // array of keys in hash set
    p->elements = (char **)malloc(sizeof(char *) * set_size);
    for (int i = 0; i < set_size; ++i)
    {
        p->elements[i] = NULL;
    }

    p->cur_size = 0;
    p->size = set_size;

    // stack of pointers to strings used as arguments when searching for a string in hsearch;
    //   kept so we can deallocate them at cleanup
    p->ps = malloc(sizeof(PSTACK));
    memset(p->ps, 0, sizeof(PSTACK));
    init_pstack(p->ps, 1);

    return 0;
}

/**
 * @brief check if hash set is at capacity
 * @param p HSET*: (pointer to) the hash set to check
 * @return true if full; false otherwise
 */
```

```c
bool is_full_hset(HSET *p)
{
    return (p->size == p->cur_size);
}

/**
 * @brief check if hash set is empty
 * @param p HSET*: (pointer to) the hash set to check
 * @return true if empty; false otherwise
 */
bool is_empty_hset(HSET *p)
{
    return (p->cur_size == 0);
}

/**
 * @brief add key to the hash set; do nothing if key already exists
 * @param p HSET*: (pointer to) the hash set to add key to
 * @param key char*: key (string) to add
 * @return 0 on success (no error); 1 otherwise (error)
 */
int add_hset(HSET *p, char *key)
{
    if (is_full_hset(p))
    {
        resize_hset(p);
    }

    // add key to hsearch
    ENTRY item;
    // note we create a copy of the parameter key since hsearch may
    //  continue to refer to the passed-in string's memory
    item.key = malloc(strlen(key) * (sizeof(char) + 1));
    memset(item.key, 0, strlen(key) * (sizeof(char) + 1));
    strncpy(item.key, key, strlen(key));
    item.data = NULL;
    ACTION action = ENTER;
    ENTRY *retval = NULL;
    if (hsearch_r(item, action, &retval, p->hmap) == 0)
    {
        perror("hsearch_r\n");
        return 1;
    }

    // add the new string to our array of keys
    //  (so we can transfer them on resize and dellocate them at cleanup)
    p->elements[p->cur_size] = item.key;
    ++p->cur_size;

    return 0;
```

```
}

/**
 * @brief search for the key in the hash set
 * @param p HSET*: (pointer to) the hash set to search
 * @param key char*: key (string) to search
 * @return 1 if the key is found; 0 if the key isn't found
 */
int search_hset(HSET *p, char *key)
{
    // search in hsearch
    ENTRY item;
    // note we create a copy of the parameter key since
    //  hsearch may continue to refer to the passed-in string's memory
    item.key = malloc(strlen(key) * (sizeof(char) + 1));
    memset(item.key, 0, strlen(key) * (sizeof(char) + 1));
    strncpy(item.key, key, strlen(key));
    item.data = NULL;
    ACTION action = FIND;
    ENTRY *retval;
    hsearch_r(item, action, &retval, p->hmap);

    // found key
    if (retval != NULL)
    {
        // add to the pointer stack so we can deallocate at destruction
        //  we don't want to deallocate now, as hsearch may continue to refer
        //  to the string's memory
        push_pstack(p->ps, item.key);
        return 1;
    }

    // did not find key: we can deallocate the key string now
    free(item.key);
    item.key = NULL;
    return 0;
}

/**
 * @brief resize hash set to have greater capacity; maintain existing elements
 * @param p HSET*: (pointer to) the hash set to resize
 * @return 0 on success; 1 otherwise
 */
int resize_hset(HSET *p)
{
    // destroy hsearch
    hdestroy_r(p->hmap);
    free(p->hmap);
    p->hmap = NULL;

```

```c
    // reinitialize hsearch
    p->hmap = malloc(sizeof(struct hsearch_data));
    memset(p->hmap, 0, sizeof(struct hsearch_data));
    if (hcreate_r(HSET_RESIZE_FACTOR * (p->size), p->hmap) == 0)
    {
        perror("hcreate\n");
        return 1;
    }

    // allocate resized array of keys
    size_t old_size = p->size;
    char **old_elements = p->elements;
    p->size = (p->size) * HSET_RESIZE_FACTOR;
    p->elements = (char **)malloc((p->size) * sizeof(char *));

    // transfer old keys over into new array
    p->cur_size = 0;
    for (size_t i = 0; i < old_size; ++i)
    {
        add_hset(p, old_elements[i]);
        free(old_elements[i]);
        old_elements[i] = NULL;
    }
    for (size_t i = old_size; i < p->size; ++i)
    {
        p->elements[i] = NULL;
    }

    // deallocate old array of keys
    free(old_elements);
    old_elements = NULL;

    return 0;
}

/**
 * @brief deconstruct hash set: free all allocated memory
 * @param p HSET*: (pointer to) the hash set to deconstruct
 * @return 0 on success; 1 otherwise
 */
int cleanup_hset(HSET *p)
{
    for (size_t i = 0; i < p->size; ++i)
    {
        if (p->elements[i] != NULL)
        {
            free(p->elements[i]);
            p->elements[i] = NULL;
        }
    }
```

```
199     free(p->elements);
200     p->elements = NULL;
201 
202     hdestroy_r(p->hmap);
203     free(p->hmap);
204     p->hmap = NULL;
205 
206     // clean up the pstack, which cleans up the extra string pointers created during search
207     cleanup_pstack(p->ps);
208     free(p->ps);
209 
210     return 0;
211 }
```

multithreaded/hash.h

```c
/*
A hash set with strings as keys
- uses Linux's hsearch as the underlying hashmap
*/

#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <search.h>
#include "p_stack.h"

typedef struct hashmap
{
    // current capacity (max number of keys)
    size_t size;
    // array of keys in the hash set
    //  - for linear access,
    //  - for moving keys over on resize,
    //  - and for deallocation at destruction
    char **elements;
    // hsearch structure
    struct hsearch_data *hmap;
    // current number of keys
    size_t cur_size;
    // stack of pointers to strings used for searching in hsearch;
    //  so we can deallocate them at cleanup
    PSTACK *ps;
} HSET;

#define HSET_RESIZE_FACTOR 2

int init_hset(HSET *p, size_t set_size);
bool is_full_hset(HSET *p);
bool is_empty_hset(HSET *p);
int add_hset(HSET *p, char *key);
int search_hset(HSET *p, char *key);
int resize_hset(HSET *p);
int cleanup_hset(HSET *p);
```

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 }
```

multithreaded/curl_xml.h

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <curl/curl.h>
#include <libxml/HTMLparser.h>
#include <libxml/parser.h>
#include <libxml/xpath.h>
#include <libxml/uri.h>
#include "stack.h"

#define SEED_URL "http://ece252-1.uwaterloo.ca/lab4/"
#define ECE252_HEADER "X-Ece252-Fragment: "
#define CURL_USER_AGENT_FIELD "ece252 lab4 crawler"
#define BUF_SIZE 1048576 /* 1024*1024 = 1M */
#define BUF_INC 524288   /* 1024*512  = 0.5M */

#define CT_PNG "image/png"
#define CT_HTML "text/html"
#define CT_PNG_LEN 9
#define CT_HTML_LEN 9
#define URL_LENGTH 256

#define OK_REQUESTS 200
#define REDIRECT_REQUESTS 300
#define BAD_REQUESTS 400
#define INTERNAL_SERVER_ERRORS 500
#define CODE_RANGE 99

#define DEFAULT_TYPE -1
#define HTML 0
#define VALID_PNG 1
#define INVALID_PNG 2

#define max(a, b) \
  ({ __typeof__ (a) _a = (a); \
      __typeof__ (b) _b = (b); \
    _a > _b ? _a : _b; })

typedef struct recv_buf2
{
  char *buf;       // memory to hold a copy of received data
  size_t size;     // size of valid data in buf in bytes
  size_t max_size; // max capacity of buf in bytes
  int seq;         // >=0 sequence number extracted from http header
                   // <0 indicates an invalid seq number
```

```
49 } RECV_BUF;
50
51 htmlDocPtr mem_getdoc(char *buf, int size, const char *url);
52 xmlXPathObjectPtr getnodeset(xmlDocPtr doc, xmlChar *xpath);
53 int find_http(char *fname, int size, int follow_relative_links, const char *base_url, STACK
   *stack);
54 size_t header_cb_curl(char *p_recv, size_t size, size_t nmemb, void *userdata);
55 size_t write_cb_curl(char *p_recv, size_t size, size_t nmemb, void *p_userdata);
56 int recv_buf_init(RECV_BUF *ptr, size_t max_size);
57 int recv_buf_cleanup(RECV_BUF *ptr);
58 void cleanup(CURL *curl, RECV_BUF *ptr);
59 CURL *easy_handle_config(CURL *curl_handle, RECV_BUF *ptr, const char *url);
60 int process_data(CURL *curl_handle, RECV_BUF *p_recv_buf, int *content_type, STACK *stack, long
   *response_code_p);
61 int process_png(CURL *curl_handle, RECV_BUF *p_recv_buf, int *content_type);
62 bool is_png(uint8_t *buf, size_t n);
63 int process_url(CURL *curl_handle, char *seed_url, int *content_type, STACK *stack, long
   *response_code_p);
64 bool is_processable_response(long response_code);
```