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);

    pngs = malloc(sizeof(STACK));
    memset(pngs, 0, sizeof(STACK));
    init_stack(pngs, STACK_SIZE);

    done = false;
    num_waiting_on_url = 0;
    num_running = 0;

    pthread_cond_init(&frontier_empty, NULL);
    pthread_mutex_init(&frontier_mutex, NULL);
    pthread_mutex_init(&visited_mutex, NULL);
    pthread_mutex_init(&pngs_mutex, NULL);
}

/**
 * @brief cleanup global variables and synchronization variables
 */
void cleanup_global()
{
    cleanup_stack(frontier);
    free(frontier);
    frontier = NULL;

    cleanup_hset(visited);
    free(visited);
    visited = NULL;

    cleanup_stack(pngs);
    free(pngs);
    pngs = NULL;

    pthread_cond_destroy(&frontier_empty);
    pthread_mutex_destroy(&frontier_mutex);
    pthread_mutex_destroy(&pngs_mutex);
    pthread_mutex_destroy(&visited_mutex);
}

/**
 * @brief runner function that crawls urls in the global frontier
 * @param _ void*: not used; only defined to satisfy thread API
 * @return NULL
 * @details
 * Any number of runner threads can be started.
 * The runner function manages concurrency.
 * The runner function assumes that all global structures and
 *  synchronization variables are initialized.
 * The runner function will stop once there are no more urls to crawl
 *  or when we've found num_pngs_to_find pngs.
 * 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 -- */
        pthread_mutex_lock(&frontier_mutex);
        {
            // If the crawl is finished, signal sleeping threads to
            //  wake up so they can exit
            if (is_empty_stack(frontier) && num_running == 0)
            {
                done = true;
                if (num_waiting_on_url > 0)
                {
                    pthread_cond_broadcast(&frontier_empty);
                }
            }

            // If there are no urls to crawl and the crawl is not done, wait
            while (is_empty_stack(frontier) && !done)
            {
                ++num_waiting_on_url;
                pthread_cond_wait(&frontier_empty, &frontier_mutex);
                --num_waiting_on_url;
            }

            // If the crawl is finished, exit the loop
            if (done)
            {
                pthread_mutex_unlock(&frontier_mutex);
                break;
            }

            // Take the top url on the frontier
            pop_stack(frontier, &url_to_crawl);

            // Check if the url has been visited
            pthread_mutex_lock(&visited_mutex);
            {
                // If the url has been visited, go back to the top of the loop
                //  (go to the next url in the frontier or if frontier is empty, wait)
                if (search_hset(visited, url_to_crawl) == 1)
                {
                    pthread_mutex_unlock(&visited_mutex);
                    pthread_mutex_unlock(&frontier_mutex);
                    continue;
                }
                // If the url has not been visited, mark it as visited.
                //  The thread will now process the url.
                else
                {
                    add_hset(visited, url_to_crawl);
                }
            }
            pthread_mutex_unlock(&visited_mutex);
            ++num_running;
        }
        pthread_mutex_unlock(&frontier_mutex);
        /* ----------------- */

#ifdef DEBUG_URL_PRINT
        printf("URL: %s\n", url_to_crawl);
#endif

        /* -- Crawl the url -- */
        // download the contents at the url and process it
        process_url(curl_handle, url_to_crawl, &content_type, urls_found, &response_code);
        /* ----------------- */

        /* -- Process url based on its contents -- */
        if (is_processable_response(response_code))
        {
            // If the url was a HTML page, add all urls on that page to the frontier
            if (content_type == HTML)
            {
                char *url_in_html = NULL;
                while (pop_stack(urls_found, &url_in_html) == 0)
                {
                    // Add to the frontier and signal sleeping threads
                    //  (that a url is ready in frontier)
                    pthread_mutex_lock(&frontier_mutex);
                    {
                        push_stack(frontier, url_in_html);
                        if (num_waiting_on_url > 0)
                        {
                            pthread_cond_broadcast(&frontier_empty);
                        }
                    }
                    pthread_mutex_unlock(&frontier_mutex);
                    free(url_in_html);
                    url_in_html = NULL;
                }
            }
            // If the url was a valid PNG, add it to our collection of found pngs
            else if (content_type == VALID_PNG)
            {
                pthread_mutex_lock(&pngs_mutex);
                {
                    push_stack(pngs, url_to_crawl);
                    // If we've reached the maximum number of PNGs we want to find,
                    //  end the program
                    if (num_elements_stack(pngs) >= num_pngs_to_find)
                    {
                        pthread_mutex_lock(&frontier_mutex);
                        {
```

```c
                            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;

    if (argc == 1)
    {
        printf("Usage: ./findpng2 OPTION[-t=<NUM> -m=<NUM> -v=<LOGFILE>] SEED_URL\n");
        return -1;
    }

    seed_url = argv[argc - 1];

    int c;
    char *str = "option requires an argument";

    while ((c = getopt(argc, argv, "t:m:v:")) != -1)
    {
        switch (c)
        {
        case 't':
            if (optarg == NULL)
            {
                t = 1;
                break;
            }
            t = strtoul(optarg, NULL, 10);
            if (t <= 0)
            {
                fprintf(stderr, "%s: %s > 0 -- 't'\n", argv[0], str);
                return -1;
            }
            break;
        case 'm':
            if (optarg == NULL)
            {
                num_pngs_to_find = 50;
                break;
            }
            num_pngs_to_find = atoi(optarg);
            if (num_pngs_to_find < 0)
            {
                fprintf(stderr, "%s: %s >= 0 -- 'm'\n", argv[0], str);
                return -1;
            }
            break;
        case 'v':
            if (optarg == NULL)
            {
                logfile = NULL;
                break;
            }
            logfile = malloc(sizeof(char) * FILE_PATH_SIZE);
            memset(logfile, 0, sizeof(char) * FILE_PATH_SIZE);
            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);
```

```c
    }
    /* ----------------- */

    /* -- Write to files -- */
    // Write png urls
    FILE *fpngs = fopen("./png_urls.txt", "w+");
    if (fpngs == NULL)
    {
        fprintf(stderr, "Opening png file for write failed\n");
        exit(1);
    }
    char *temp = NULL;
    while (pop_stack(pngs, &temp) == 0)
    {
        fprintf(fpngs, "%s\n", temp);
        free(temp);
    }
    fclose(fpngs);

    // Write all urls visited into a log file if user desires
    if (logfile != NULL)
    {
        char *logfile_name = malloc(sizeof(char) * FILE_PATH_SIZE);
        memset(logfile_name, 0, sizeof(char) * FILE_PATH_SIZE);
        sprintf(logfile_name, "./%s", logfile);
        FILE *flogs = fopen(logfile_name, "w+");
        free(logfile_name);
        if (flogs == NULL)
        {
            fprintf(stderr, "Opening log file for write failed\n");
            exit(1);
        }
        temp = NULL;
        for (size_t i = 0; i < visited->cur_size; ++i)
        {
            fprintf(flogs, "%s\n", visited->elements[i]);
            free(temp);
        }
        fclose(flogs);
    }
    free(logfile);
    /* ----------------- */

    /* -- Cleanup global variables and synchronization variables -- */
    cleanup_global();
    /* ----------------- */

    /* -- Free threads -- */
    free(runners);
    /* ----------------- */

    /* -- Clean up libraries used -- */
    curl_global_cleanup();
    xmlCleanupParser();
    /* ----------------- */

    /* -- Print time it took for crawl from the seed url -- */
    if (gettimeofday(&tv, NULL) != 0)
    {
        perror("gettimeofday");
        exit(1);
    }
    times[1] = (tv.tv_sec) + tv.tv_usec / 1000000.;
    printf("findpng2 execution time: %.6lf seconds\n", times[1] - times[0]);
    /* ----------------- */

    return 0;
}
```