

multithreaded/findpng2.c

```

1  #include "findpng2.h"
2
3  // I defined this for the code sample review
4  // so if you want to make and run the program,
5  // you can see the urls being crawled on command line
6  #define DEBUG_URL_PRINT
7
8  /* -- Global Variables -- */
9  // global collection of urls to be crawled by runner threads
10 STACK *frontier;
11 // pngs found (png urls)
12 STACK *pngs;
13 // urls visited
14 HSET *visited;
15 // whether we are done with the entire crawl
16 bool done;
17 // number of threads waiting for a non-empty frontier
18 size_t num_waiting_on_url;
19 // number of thread runners currently processing a url
20 size_t num_running;
21 // number of pngs to find before stopping
22 int num_pngs_to_find;
23 /* ----- */
24
25 /* -- Synchronization --*/
26 // condition variable for threads to wait on when the frontier is empty
27 pthread_cond_t frontier_empty;
28 // lock for frontier, done, num_waiting_on_url, and num_running;
29 // also used for frontier_empty
30 pthread_mutex_t frontier_mutex;
31 // lock for pngs stack
32 pthread_mutex_t pngs_mutex;
33 // lock for visited hash set
34 pthread_mutex_t visited_mutex;
35 /* ----- */
36
37 /**
38  * @brief initialize global variables and synchronization variables
39  */
40 void initialize_global()
41 {
42     frontier = malloc(sizeof(STACK));
43     memset(frontier, 0, sizeof(STACK));
44     init_stack(frontier, STACK_SIZE);
45
46     visited = malloc(sizeof(HSET));
47     memset(visited, 0, sizeof(HSET));
48     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.
```

```
99  */
100 void *runner(void *)
101 {
102     /* -- Initialize cURL easy handle -- */
103     CURL *curl_handle = curl_easy_init();
104     if (curl_handle == NULL)
105     {
106         fprintf(stderr, "curl_easy_init: returned NULL\n");
107         exit(1);
108     }
109     /* ----- */
110
111     /* -- Defining variables used in the loop -- */
112     // response code from accessing url
113     long response_code;
114     // content type of data at url (e.g. HTML, PNG)
115     int content_type = DEFAULT_TYPE;
116     // the current url the thread is crawling
117     char *url_to_crawl = NULL;
118     // urls found on the web page visited; we will add these to the frontier
119     STACK *urls_found = NULL;
120     // if we have cleaned urls_found
121     bool cleaned_urls_found = false;
122     /* ----- */
123
124     while (true)
125     {
126         /* -- Cleanup structures from last iteration and re-initialize -- */
127         if (urls_found != NULL)
128         {
129             if (!cleaned_urls_found)
130             {
131                 cleanup_stack(urls_found);
132                 cleaned_urls_found = true;
133             }
134             free(urls_found);
135         }
136         urls_found = malloc(sizeof(STACK));
137         memset(urls_found, 0, sizeof(STACK));
138         init_stack(urls_found, 1);
139         cleaned_urls_found = false;
140
141         if (url_to_crawl != NULL)
142         {
143             free(url_to_crawl);
144             url_to_crawl = NULL;
145         }
146         /* ----- */
147
148         /* -- 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                     {
```

```
249         done = true;
250         pthread_cond_broadcast(&frontier_empty);
251     }
252     pthread_mutex_unlock(&frontier_mutex);
253 }
254 }
255     pthread_mutex_unlock(&pngs_mutex);
256 }
257 }
258 /* ----- */
259
260 /* -- The thread is no longer processing a url -- */
261 pthread_mutex_lock(&frontier_mutex);
262 {
263     --num_running;
264 }
265 pthread_mutex_unlock(&frontier_mutex);
266 /* ----- */
267 }
268
269 /* -- The thread is done all processing: clean up -- */
270 if (urls_found != NULL)
271 {
272     if (!cleaned_urls_found)
273     {
274         cleanup_stack(urls_found);
275     }
276     free(urls_found);
277 }
278
279 if (url_to_crawl != NULL)
280 {
281     free(url_to_crawl);
282     url_to_crawl = NULL;
283 }
284
285 curl_easy_cleanup(curl_handle);
286 /* ----- */
287
288 return NULL;
289 }
290
291 int main(int argc, char **argv)
292 {
293     /* -- command line inputs -- */
294     char *seed_url;
295     char *logfile = NULL;
296     size_t t = 1;
297     num_pngs_to_find = 50;
298 }
```

```
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);
```

```
349         break;
350     }
351 }
352 /* ----- */
353
354 /* -- initialize global variables and synchronization variables -- */
355 initialize_global();
356 /* ----- */
357
358 /* -- CURL global init -- */
359 curl_global_init(CURL_GLOBAL_DEFAULT);
360 /* ----- */
361
362 /* -- Initialize XML Parser -- */
363 xmlInitParser();
364 /* ----- */
365
366 /* -- Put the seed URL in the frontier -- */
367 push_stack(frontier, seed_url);
368 /* ----- */
369
370 /* -- Record time to be used for measuring speed -- */
371 double times[2];
372 struct timeval tv;
373 if (gettimeofday(&tv, NULL) != 0)
374 {
375     perror("gettimeofday");
376     exit(1);
377 }
378 times[0] = (tv.tv_sec) + tv.tv_usec / 1000000.;
379 /* ----- */
380
381 /* -- Create threads -- */
382 pthread_t *runners = malloc(t * sizeof(pthread_t));
383 memset(runners, 0, sizeof(pthread_t) * t);
384 if (runners == NULL)
385 {
386     perror("malloc\n");
387     exit(-1);
388 }
389 for (int i = 0; i < t; ++i)
390 {
391     pthread_create(&runners[i], NULL, runner, NULL);
392 }
393 /* ----- */
394
395 /* -- Wait for threads to finish -- */
396 for (int i = 0; i < t; ++i)
397 {
398     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 }
```