## 1 Writing a simple concurrent caching http proxy

**Solution** The created proxy is a simple implementation of a forward proxy. It supports only HTTP GET request. But allow for multiple clients to be served concurrently. Also, it supports response caching with least recently used (LRU) cache policy. It uses two additional custom C modules sbuf and cache:

- The sbuf is a simple implementation of a concurrent producer/consumer buffer. It uses semaphores to limit the number of readers and writers based on available item slots. And an exclusive lock for accessing buffered items.
- The cache is an LRU cache, with linked list cache structure. LRU is provided by maintaining a linked list queue for access history. It moves cache item access record to the end of the queue each time the item is accessed. When total cache size becomes greater than the limit, it evicts item with LRU records located at the beginning of the LRU queue.

The source code of proxy is shown on listing 1, the source code of sbuf and cache is shown on listing 2 and listing 3 correspondingly.

```
#include <stdio.h>
1
   #include "csapp.h"
   #include "sbuf.h"
   #include "cache.h"
6
   /* Max size of a cacheable request */
   #define MAX_OBJECT_SIZE 102400
8
   /* HTTP requset line limits */
9
10
   #define MAX_HOSTNAME_LEN 256
11
   #define MAX_PORT_LEN 6
12
   #define MAX_QUERY_LEN 2048
13
   /* HTTP max header limit */
14
15
   #define MAX_HEADERS_NUMBER 200
16
17
    /* Thread pool constants */
18
   #define NTHREADS 4
19
   #define SBUFSIZE 16
20
21
    /* Task required proxy headers */
   static const char *user_agent_hdr = "User-Agent: Mozilla/5.0 (X11; Linux
        x86_64; rv:10.0.3) Gecko/20120305 Firefox/10.0.3\r\n";
   static const char *connection_hdr = "Connection: close\r\n";
23
   static const char *proxy_connection_hdr = "Proxy-Connection: close\r\n";
24
25
26
   typedef struct uri_info
27
        char hostname[MAX_HOSTNAME_LEN];
28
        char port[MAX_PORT_LEN];
29
30
        char query[MAX_QUERY_LEN];
31
   } Uri_info;
32
33
   typedef struct headers
34
35
        int cout;
36
        char *data[MAX_HEADERS_NUMBER];
37
   } Headers:
38
39
   static void proxy(int fd);
   static int parse_headers(rio_t *rp, Headers *headers);
40
41
   static int parse_uri(const char *uri, Uri_info *uri_info);
   static void forward(int fd, Uri_info *uri_info, Headers *headers);
42
43
   static void clear_headers(Headers *headers);
   static void *thread(void *vargp);
44
   static int build_cache_key(char key[], size_t length, Uri_info *uri_info); static void clienterror(int fd, char *cause, char *errnum, char *shortmsg,
45
46
        char *longmsg);
47
   static void servererror(int fd);
48
49
   /* Thread safe buffer */
50 static sbuf_t sbuf;
```

```
Listing 1 (Cont.): proxy.c
    /* Local proxy cache */
52 static Cache cache;
53
54
    int main(int argc, char **argv)
55
    {
56
         if (argc != 2)
57
         {
58
             fprintf(stderr, "usage: %s <port>\n", argv[0]);
59
             exit(1);
60
         }
61
         /* Blocking SIGPIPE signal */
62
         /* SIGPIPE signal will be send by write() when
63
64
         * the connection socket closes prematurely.
         * An unhandled SIGPIPE signal terminates the program */
65
66
         sigset_t mask;
67
         if (Sigemptyset(&mask) < 0)</pre>
68
             exit(1);
69
70
         if (Sigaddset(&mask, SIGPIPE) < 0)
71
             exit(1);
72
73
         if (Sigprocmask(SIG_BLOCK, &mask, NULL) < 0)
74
             exit(1);
75
76
         /* Creatin listening socket */
77
         int listenfd:
         if ((listenfd = Open_listenfd(argv[1])) < 0)</pre>
78
79
             exit(1);
80
         /* Creating thread pool */
81
         if (sbuf_init(&sbuf, SBUFSIZE) < 0)</pre>
82
83
             exit(1);
84
         if (cache_init(&cache) < 0)</pre>
85
86
             exit(1);
87
88
         int i;
         pthread_t tid;
89
         for (i = 0; i < NTHREADS; i++)
90
91
             if (Pthread_create(&tid, NULL, thread, NULL) != 0)
92
                 exit(1);
93
94
         struct sockaddr_storage clientaddr;
95
         socklen_t clientlen = sizeof(clientaddr);
96
         int connfd;
97
         while (1)
98
         ł
             if ((connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen)) < 0)</pre>
99
100
                 continue;
101
102
             /* Logging client info */
103
             char client_host[MAXLINE];
104
             char port[MAXLINE];
105
             Getnameinfo((SA *)&clientaddr, clientlen, client_host, MAXLINE, port,
                 MAXLINE. 0):
106
             printf("Accepted connection from (%s, %s)\n", client_host, port);
107
             /* Adding request to the buffer */
             if (sbuf_insert(&sbuf, connfd) < 0)
108
109
                 exit(1);
110
         }
111
112
         return 0;
113 }
114
115 static void *thread(void *vargp)
116 {
117
         if (Pthread_detach(pthread_self()) != 0)
118
             exit(1):
119
120
         int connfd;
121
         while (1)
```

122

```
Listing 1 (Cont.): proxy.c
             /* Dequeuing a request from the buffer */
             if (sbuf_remove(&sbuf, &connfd) < 0)
124
125
                  exit(1);
126
127
             /* Request pipeline */
128
             proxy(connfd);
129
             Close(connfd);
130
131
    }
132
133
    /* This is the main request pipeline routine.
    * It gets a connected socket descriptor. Parses the request,
134
     * sends a proxy request to the requested server and
135
136
     * forwards the response back to the client. */
    static void proxy(int fd)
137
138
139
         rio_t rio;
140
         Rio_readinitb(&rio, fd);
141
         char buf[MAXLINE];
142
         /* Reading an HTTP request line from the connected socket */
         if (Rio_readlineb(&rio, buf, MAXLINE) < 0)
143
144
145
             servererror(fd);
146
             return;
147
148
         char method[MAXLINE];
149
150
         char uri[MAXLINE];
151
         char version[MAXLINE];
         if (sscanf(buf, "%s %s %s", method, uri, version) == EOF)
152
153
             clienterror(fd, "", "400", "Bad request",
154
155
                           "Request is empty");
156
             return;
157
         }
158
159
         printf("%s", buf);
         if (strcasecmp(method, "GET"))
160
161
             clienterror(fd, method, "501", "Not implemented",
162
163
                           "Proxy does not implement this method");
164
             return;
165
         }
166
         if (strcasecmp(version, "HTTP/1.0") &&
    strcasecmp(version, "HTTP/1.1"))
167
168
169
170
             clienterror(fd, version, "501", "Not implemented",
171
                           "Proxy supports only HTTP/1.0(1.1) protocol versions");
172
         }
173
174
         Uri_info reciever;
175
         if (parse_uri(uri, &reciever) < 0)</pre>
176
177
             clienterror(fd, uri, "400", "Bad request",
                           "Invalid request URI.
178
                          "URI must have the following structure: "
179
                           "http[s]://{hostname}[:{port}]/{location}");
180
181
             return:
182
183
         Headers headers;
184
185
         int status:
186
         if ((status = parse_headers(&rio, &headers)) < 0)</pre>
187
188
             switch (status)
189
             {
190
                  clienterror(fd, "", "413", "Entity is too large",
191
                               "Maximum header count or maximum header length is
192
                                   exceeded");
193
                 break;
194
             case -2:
```

```
Listing 1 (Cont.): proxy.c
                 servererror(fd);
196
                 break:
197
             }
198
199
             clear_headers(&headers);
200
             return;
201
202
203
         forward(fd, &reciever, &headers);
         clear_headers(&headers);
204
205 }
206
207 /* This is the part of the request pipeline, where the request is being send
        to the server
     * and the response is being forwarded to the client */
209 static void forward(int fd, Uri_info *uri_info, Headers *headers)
210 {
211
         jmp_buf fd_closed;
212
         if (setjmp(fd_closed))
213
             return;
214
215
         /* Returning cached data if present */
216
         void *cached = NULL;
217
         size_t cached_length;
218
         char key[MAXLINE];
219
         if (build_cache_key(key, MAXLINE, uri_info))
220
221
             if (cache_get(&cache, key, &cached, &cached_length) > 0)
222
             {
                 Rio_writen(fd, cached, cached_length, fd_closed);
223
224
                 return;
225
             }
226
         }
227
228
         rio_t rio;
229
         int clientfd;
230
         if ((clientfd = Open_clientfd(uri_info->hostname, uri_info->port)) < 0)</pre>
231
232
             switch (clientfd)
233
234
             case -1:
235
                 servererror(fd);
236
                 break:
237
             case -2:
                 clienterror(fd, uri_info->hostname, "400", "Host not found",
238
                              "The DNS entry for the hostname was not resolved");
239
240
                 break;
241
             }
242
243
             return;
244
        }
245
         /* jmp if clientfd closes connection */
246
247
         jmp_buf client_close_conn;
248
         if (setjmp(client_close_conn))
249
250
             Close(clientfd);
251
             servererror(fd);
252
             return;
253
254
         /* Sending the HTTP request line to the client socket */
255
256
         Rio_readinitb(&rio, clientfd);
257
         char buf[MAXLINE];
         sprintf(buf, "GET ");
258
259
         Rio_writen(clientfd, buf, strlen(buf), client_close_conn);
         sprintf(buf, "%s", uri_info->query);
260
261
         Rio_writen(clientfd, buf, strlen(buf), client_close_conn);
262
         sprintf(buf, " HTTP/1.0\r\n");
         Rio_writen(clientfd, buf, strlen(buf), client_close_conn);
263
264
         \slash* Sending the HTTP headers from the request to the client socket */
265
         int host_in_headers = 0;
266
        int i;
```

```
267
         for (i = 0; i < headers->cout; i++)
268
269
             if (!host_in_headers && strstr(headers->data[i], "Host:"))
270
                 host_in_headers = 1;
271
             else if (strstr(headers->data[i], "User-Agent:") ||
                       strstr(headers->data[i], "Connection:") ||
strstr(headers->data[i], "Proxy-Connection"))
272
273
274
                 continue;
275
276
             Rio_writen(clientfd, headers->data[i], strlen(headers->data[i]),
                 client_close_conn);
277
         }
278
279
         if (!host_in_headers)
280
281
             /* Sending a Host header */
282
             sprintf(buf, "Host: ");
             Rio_writen(clientfd, buf, strlen(buf), client_close_conn);
283
284
             sprintf(buf, "%s", uri_info->hostname);
285
             Rio_writen(clientfd, buf, strlen(buf), client_close_conn);
286
             sprintf(buf, "\r\n");
287
         }
288
289
         /* Sending a User-Agent header */
290
         sprintf(buf, "%s", user_agent_hdr);
291
         Rio_writen(clientfd, buf, strlen(buf), client_close_conn);
292
         /* Sending a Connection header */
293
         sprintf(buf, "%s", connection_hdr);
294
         Rio_writen(clientfd, buf, strlen(buf), client_close_conn);
295
         /* Sending a Proxy-Connection header */
         sprintf(buf, "%s", proxy_connection_hdr);
296
297
         Rio_writen(clientfd, buf, strlen(buf), client_close_conn);
298
         /* Sending the end of the request */
         sprintf(buf, "\r\n");
299
300
         Rio_writen(clientfd, buf, strlen(buf), client_close_conn);
301
302
         /* jmp is the client close connection while the request is being
             forwarded */
303
         jmp_buf fd_closed_and_client_opened;
304
         if (setjmp(fd_closed_and_client_opened))
305
306
             Close(clientfd);
307
             return:
308
309
         /* Creating a temp cache buffer */
310
311
         size_t cached_len = 0;
312
         int cacheable = 1;
313
         char cache_buffer[MAX_OBJECT_SIZE];
314
         char *cache_free = cache_buffer;
315
         \slash * Transmitting the response from the client socket to the listening
             socket*/
316
         Rio_readinitb(&rio, clientfd);
317
         size_t response_len = 0;
318
         while ((response_len = Rio_readnb(&rio, buf, MAXLINE)) > 0)
319
320
             if (cacheable)
321
322
                 if (cached_len + response_len > MAX_OBJECT_SIZE)
323
                 {
324
                      cacheable = 0;
                 }
325
326
                 else
327
                 {
328
                      memcpy(cache_free, buf, response_len);
329
                      cached_len += response_len;
                      cache_free += response_len;
330
331
332
             }
333
334
             Rio_writen(fd, buf, response_len, fd_closed_and_client_opened);
335
         }
336
```

Listing 1 (Cont.): proxy.c

```
Listing 1 (Cont.): proxy.c
337
         Close(clientfd);
         /* Caching the respone */
338
339
         if (!(response_len < 0) && cacheable)
340
341
             char key[MAXLINE];
             if (build_cache_key(key, MAXLINE, uri_info))
342
343
                 cache_add(&cache, key, cache_buffer, cached_len);
344
345 }
346
347 static int parse_uri(const char *uri, Uri_info *uri_info)
348 {
349
         char buf[MAXLINE];
350
         char *index = buf;
         /* Parsing a protocol */
351
352
         while (*uri != ':')
353
354
             if (*uri == '\0')
355
                 return -1;
356
357
             *index++ = *uri++;
358
         }
359
360
         *index = '\0';
361
         if (strcasecmp(buf, "http") && strcasecmp(buf, "https"))
362
             return -1;
363
364
         /* Omitting duplicated slashes */
365
         uri++;
         while (*uri == '/')
366
367
            uri++;
368
         /* Parsing a hostname */
369
370
         index = buf;
         while (*uri != ':' && *uri != '/')
371
372
373
             if (*uri == '\0')
374
                 return -1;
375
376
             *index++ = *uri++;
377
378
         *index = '\0';
379
380
         /* Filling Uri_info with the hostname */
381
         size_t hostname_len = strlen(buf) + 1;
382
         if (hostname_len > MAX_HOSTNAME_LEN)
383
             return -1;
384
         memcpy(uri_info->hostname, buf, hostname_len);
385
386
         /* Parsing a port value */
387
         index = buf;
388
         if (*uri == ':')
389
         ł
390
             uri++:
391
             while (*uri != '/')
392
             {
                 if (*uri == '\0')
393
394
                     return -1;
395
396
                 *index++ = *uri++;
397
398
399
             *index = '\0';
             /* Filling Uri_info with the port value*/
400
401
             size_t port_len = strlen(buf) + 1;
402
             if (port_len > MAX_PORT_LEN)
403
                 return -1;
404
405
             memcpy(uri_info->port, buf, port_len);
406
         }
407
         else
408
         {
409
             char *default_port = "80";
```

```
Listing 1 (Cont.): proxy.c
410
             strcpy(uri_info->port, default_port);
411
412
413
         uri++;
414
         /* Omitting duplicated slashes*/
415
         while (*uri == '/')
416
             uri++;
417
418
         /* Parsing a query */
         uri--;
419
420
         index = buf;
421
         while (*uri != '\0')
422
             *index++ = *uri++;
423
424
         *index = '\0';
425
         /*\ \textit{Filling Uri\_info with query data */}
426
         size_t query_len = strlen(buf) + 1;
427
         if (query_len > MAX_QUERY_LEN)
428
             return -1:
429
430
         memcpy(uri_info->query, buf, query_len);
431
432
         return 0;
433 }
434
435 static int parse_headers(rio_t *rp, Headers *headers)
436 {
437
         headers->cout = 0;
         char buf[MAXLINE];
438
439
         ssize_t line_len;
         /* Reading first header line from the connected socket*/
440
441
         if ((line_len = Rio_readlineb(rp, buf, MAXLINE)) < 0)</pre>
442
             return -2;
443
         /* Checking if the header line length is less than MAXLINE */
444
445
         if (line_len == MAXLINE - 1)
             if (buf[line_len - 1] != '\n')
446
447
                 return -1;
448
449
         /* While the line is not the end of the request */
450
         while (strcmp(buf, "\r\n"))
451
             if (headers->cout > MAX_HEADERS_NUMBER)
452
453
                 return -1;
454
             /* Copying the header to the Headers struct*/
455
456
             char *header;
457
             if ((header = Malloc(line_len + 1)) == NULL)
458
                 return -2;
459
460
             memcpy(header, buf, line_len + 1);
461
             headers -> data[headers -> cout++] = header;
462
463
             /* Reading next line */
464
             if ((line_len = Rio_readlineb(rp, buf, MAXLINE)) < 0)</pre>
465
                 return -2:
466
467
             \slash Checking if the header line length is less than MAXLINE */
             if (line_len == MAXLINE - 1)
468
469
                 if (buf[line_len - 1] != '\n')
470
                     return -1;
         }
471
472
473
         return 0;
474 }
475
476 \, static void clienterror(int fd, char *cause, char *errnum,
477
                              char *shortmsg, char *longmsg)
478
         jmp_buf fd_closed;
479
         if (setjmp(fd_closed))
480
481
             return;
482
```

## Listing 1 (Cont.): proxy.c

```
483
         char buf[MAXLINE];
484
         /* Print the HTTP response headers */
         sprintf(buf, "HTTP/1.0 %s %s\r\n", errnum, shortmsg);
Rio_writen(fd, buf, strlen(buf), fd_closed);
485
486
         sprintf(buf, "Content-type: text/html\r\n\r\n");
487
         Rio_writen(fd, buf, strlen(buf), fd_closed);
488
489
490
         /* Print the HTTP response body */
         sprintf(buf, "<html><title>Proxy Error</title>");
Rio_writen(fd, buf, strlen(buf), fd_closed);
491
492
         493
494
495
                       ">\r\n");
496
         Rio_writen(fd, buf, strlen(buf), fd_closed);
         sprintf(buf, "%s: %s\r\n", errnum, shortmsg);
497
498
         Rio_writen(fd, buf, strlen(buf), fd_closed);
499
         sprintf(buf, "%s: %s\r\n", longmsg, cause);
         Rio_writen(fd, buf, strlen(buf), fd_closed);
500
501
         sprintf(buf, "<hr><em>The Proxy</em>\r\n");
502
         Rio_writen(fd, buf, strlen(buf), fd_closed);
503 }
504
505
    static void servererror(int fd)
506
    {
507
         clienterror(fd, "", "500", "Internal server error",
508
                      "Something went wrong");
509 }
510
511
    static void clear_headers(Headers *headers)
512 {
513
         int i;
514
         for (i = 0; i < headers->cout; i++)
515
             free(headers->data[i]);
516 }
517
518
    static int build_cache_key(char key[], size_t length, Uri_info *uri_info)
519 {
520
         size_t key_length = strlen(uri_info->hostname) +
521
                               strlen(uri_info->port) +
522
                               strlen(uri_info->query);
523
         if (length < key_length + 1)</pre>
524
             return 0;
525
526
         memset(key, 0, length);
527
         strcat(key, uri_info->hostname);
528
         strcat(key, uri_info->port);
529
         strcat(key, uri_info->query);
530
         return 1;
531 }
```

Listing 1: proxy.c

```
1 #include "sbuf.h"
3
   int sbuf_init(sbuf_t *sp, int n)
4
5
        if ((sp->buf = Calloc(n, sizeof(int))) < 0)</pre>
6
             return -1;
        sp->n = n;
sp->front = sp->rear = 0;
8
9
10
        if (Sem_init(&sp->mutex, 0, 1) < 0)
11
             return -1;
12
13
        if (Sem_init(&sp->slots, 0, n) < 0)</pre>
14
            return -1;
15
        if (Sem_init(&sp->items, 0, 0) < 0)
16
17
             return -1;
18
19
        return 0;
20 }
21
22 void sbuf_free(sbuf_t *sp)
23 {
24
        Free(sp->buf);
25 }
26
27
   int sbuf_insert(sbuf_t *sp, int item)
28
   {
29
        if (Sem_wait(&sp->slots) < 0)</pre>
30
             return -1;
31
        if (Sem_wait(&sp->mutex) < 0)</pre>
32
33
            return -1;
34
35
        sp->buf[(++sp->rear) % (sp->n)] = item;
        if (Sem_post(&sp->mutex) < 0)</pre>
36
37
             return -1;
38
39
        if (Sem_post(&sp->items) < 0)</pre>
40
            return -1;
41
42
        return 0;
43 }
44
45
   int sbuf_remove(sbuf_t *sp, int *item)
46
   {
47
        if (Sem_wait(&sp->items) < 0)</pre>
48
             return -1;
49
50
        if (Sem_wait(&sp->mutex) < 0)</pre>
             return -1;
51
52
53
        *item = sp->buf[(++sp->front) % (sp->n)];
        if (Sem_post(&sp->mutex) < 0)</pre>
54
55
             return -1;
56
        if (Sem_post(&sp->slots) < 0)</pre>
57
58
             return -1;
59
60
        return 0;
61 }
```

Listing 2: sbuf.c

```
1 #include "cache.h"
3 /* Max size of a cacheable request */
   #define MAX_OBJECT_SIZE 102400
4
   #define MAX_CACHE_SIZE 1049000
6
   #define PAYLOAD(node_p) ((char *)node_p + sizeof(CacheNode))
8
   #define NODE_SIZE(payload_size) (sizeof(CacheNode) + payload_size)
10 static void prepend_item(Cache *cache, CacheNode *item);
11 static void remove_item(Cache *cache, CacheNode *item);
12
   static void append_lru(Cache *cache, LruNode *item);
13 static void remove_lru(Cache *cache, LruNode *item);
14
15 int cache_init(Cache *cache)
16 f
17
        cache->total_size = 0;
18
        cache->head = NULL;
        if (Sem_init(&cache->readers_lock, 0, 1) < 0)</pre>
19
20
            return -1;
21
22
        if (Sem_init(&cache->writers_lock, 0, 1) < 0)</pre>
23
            return -1;
24
25
        return 0;
26 }
27
28 int cache_add(Cache *cache, const char *key, const void *buf, size_t length)
29 {
30
        if (length > MAX_OBJECT_SIZE)
31
            return 0;
32
33
        /* Holding writers lock */
34
        if (Sem_wait(&cache->writers_lock) < 0)</pre>
35
            return -1:
36
37
        /* LRU eviction */
38
        while (cache->total_size + length > MAX_CACHE_SIZE)
39
            CacheNode *to_evict = cache->lru_head->item;
40
            cache->total_size -= to_evict->size;
41
42
            remove_item(cache, to_evict);
43
            remove_lru(cache, to_evict->lru);
            printf("Evicted fromt the cache: %s, freed %d bytes.\n",
44
               to_evict->key, (int)to_evict->size);
45
            free(to_evict->lru);
46
            free(to_evict);
47
48
49
        /* Creating a new cache node */
        CacheNode *new_item;
50
51
        if ((new_item = Malloc(NODE_SIZE(length))) < 0)</pre>
52
            return -1;
53
54
        /* Creating a key copy */
        char *key_copy;
55
        if ((key_copy = Malloc(strlen(key) + 1)) < 0)</pre>
56
57
            return -1:
58
        memcpy(key_copy, key, strlen(key) + 1);
59
        /* Allocating a new lru node */
60
61
        LruNode *new_lru;
62
        if ((new_lru = Malloc(sizeof(LruNode))) < 0)</pre>
63
            return -1;
64
65
        new_lru->item = new_item;
66
        /* Updating item fields */
67
        new_item->key = key_copy;
68
        new_item->size = length;
        new_item->lru = new_lru;
69
70
        /* Copying data to the cache node */
71
        void *data = PAYLOAD(new_item);
        memcpy(data, buf, length);
72.
```

```
Listing 3 (Cont.): cache.c
         /* Updating linked lists */
         prepend_item(cache, new_item);
74
75
         append_lru(cache, new_lru);
         printf("Writen to the cache %s. %d bytes.\n", new_item->key,
76
             (int)new_item->size);
77
         if (Sem_post(&cache->writers_lock) < 0)</pre>
78
             return -1;
79
80
         return 0;
81
    }
82
83 int cache_get(Cache *cache, const char *key, void **item, size_t *length)
84 {
85
         int is_cached = 0;
         /* Aquiring writers lock if it is the first reader */
86
87
         if (Sem_wait(&cache->readers_lock) < 0)</pre>
88
             return -1;
89
         cache->readers_count++;
90
91
         if (cache->readers_count == 1)
92
             if (Sem_wait(&cache->writers_lock) < 0)</pre>
93
                 return -1;
94
95
         if (Sem_post(&cache->readers_lock) < 0)</pre>
96
             return -1;
97
         /* Key sequential lookup */
98
99
         CacheNode *current;
100
         for (current = cache->head; current != NULL; current = current->next)
101
102
             if (!strcmp(current->key, key))
103
104
                  /* Moving the lru node to the end of the LRU list */
105
                  if (Sem_wait(&cache->readers_lock) < 0)</pre>
106
                      return -1;
107
108
                 remove_lru(cache, current->lru);
109
                  append_lru(cache, current->lru);
110
                  if (Sem_post(&cache->readers_lock) < 0)</pre>
                      return -1;
111
112
113
                  /* Copying the cached value */
                  if ((*item = Malloc(current->size)) < 0)</pre>
114
115
                      return -1;
116
                  memcpy(*item, PAYLOAD(current), current->size);
117
                  *length = current->size;
118
                  is_cached = 1;
119
120
                  break;
121
             }
122
         }
123
124
         /* Releasing writers lock if it is the last reader */
125
         if (Sem_wait(&cache->readers_lock) < 0)</pre>
126
             return -1;
127
128
         cache -> readers_count --;
129
         if (cache->readers_count == 0)
130
             if (Sem_post(&cache->writers_lock) < 0)</pre>
131
                 return -1;
132
         if (Sem_post(&cache->readers_lock) < 0)</pre>
133
134
             return -1;
135
136
         if (is_cached)
```

printf("Cache hit %s\n", key);

printf("Cache miss %s\n", key);

144 /\* Appending an item to the end of the LRU linked list \*/

return is\_cached;

137

138 139

140 141

142 } 143

## Listing 3 (Cont.): cache.c

```
static void append_lru(Cache *cache, LruNode *item)
146
147
         item->next = NULL;
148
         item->prev = cache->lru_tail;
         if (cache->lru_tail != NULL)
149
150
            cache->lru_tail->next = item;
151
         else
152
             cache->lru_head = item;
153
154
         cache->lru_tail = item;
155
    }
156
157
    /* Removing an item from the LRU linked list*/
158
    static void remove_lru(Cache *cache, LruNode *item)
159
160
         LruNode *prev = item->prev;
161
         LruNode *next = item->next;
         if (prev != NULL)
162
163
            prev->next = next;
164
165
             cache->lru_head = next;
166
167
         if (next != NULL)
168
             next->prev = prev;
169
         else
170
             cache->lru_tail = prev;
171 }
172
    \slash * Prepending an item to the start of the cached items linked list */
173
174
    static void prepend_item(Cache *cache, CacheNode *item)
175
    {
176
         item->prev = NULL;
177
         item->next = cache->head;
178
         if (cache->head != NULL)
179
             cache->head->prev = item;
180
181
         cache->head = item;
182
         cache->total_size += item->size;
183
    }
184
185
    /* Removing an item from the cached items linked list */
186
    static void remove_item(Cache *cache, CacheNode *item)
187
    {
188
         CacheNode *next = item->next;
189
         CacheNode *prev = item->prev;
         if (prev != NULL)
190
191
            prev->next = next;
192
         else
193
             cache->head = next;
194
        if (next != NULL)
195
196
             next->prev = prev;
197 }
```

Listing 3: cache.c

**Results** The proxy was run against the evaluator which verifies basic requirements.

- Validity of the proxy response. Figure 1.
- Support of concurrent requests. Figure 2.
- And functionality of the caching policy. Figure 3.

```
*** Basic ***

Starting tiny on 29683

Starting proxy on 8091

1: home.html
    Fetching ./tiny/home.html into ./.proxy using the proxy
    Fetching ./tiny/home.html into ./.noproxy directly from Tiny
    Comparing the two files
    Success: Files are identical.

2: csapp.c
    Fetching ./tiny/csapp.c into ./.proxy using the proxy
    Fetching ./tiny/csapp.c into ./.noproxy directly from Tiny
    Comparing the two files
    Success: Files are identical.

3: tiny.c
    Fetching ./tiny/tiny.c into ./.proxy using the proxy
    Fetching ./tiny/tiny.c into ./.noproxy directly from Tiny
    Comparing the two files
    Success: Files are identical.

4: godzilla.jpg
    Fetching ./tiny/godzilla.jpg into ./.proxy using the proxy
    Fetching ./tiny/godzilla.jpg into ./.noproxy directly from Tiny
    Comparing the two files
    Success: Files are identical.

5: tiny
    Fetching ./tiny/tiny into ./.proxy using the proxy
    Fetching ./tiny/tiny into ./.proxy using the proxy
    Fetching ./tiny/tiny into ./.noproxy directly from Tiny
    Comparing the two files
    Success: Files are identical.

Killing tiny and proxy
basicScore: 40/40
```

Figure 1: Basic validity check

```
*** Concurrency ***
Starting tiny on port 23727
Starting proxy on port 5107
Starting the blocking NOP server on port 9275
Trying to fetch a file from the blocking nop-server
Fetching ./tiny/home.html into ./.noproxy directly from Tiny
Fetching ./tiny/home.html into ./.proxy using the proxy
Checking whether the proxy fetch succeeded
Success: Was able to fetch tiny/home.html from the proxy.
Killing tiny, proxy, and nop-server
concurrencyScore: 15/15
```

Figure 2: Concurrency check

```
*** Cache ***

Starting tiny on port 24122

Starting proxy on port 15277

Fetching ./tiny/tiny.c into ./.proxy using the proxy

Fetching ./tiny/home.html into ./.proxy using the proxy

Fetching ./tiny/csapp.c into ./.proxy using the proxy

Killing tiny

Fetching a cached copy of ./tiny/home.html into ./.noproxy

Success: Was able to fetch tiny/home.html from the cache.

Killing proxy

cacheScore: 15/15
```

Figure 3: Cache check