

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.

```
1  #include <stdio.h>
2  #include "csapp.h"
3  #include "sbuf.h"
4  #include "cache.h"
5
6  /* Max size of a cacheable request */
7  #define MAX_OBJECT_SIZE 102400
8
9  /* HTTP request line limits */
10 #define MAX_HOSTNAME_LEN 256
11 #define MAX_PORT_LEN 6
12 #define MAX_QUERY_LEN 2048
13
14 /* HTTP max header limit */
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 */
22 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";
23 static const char *connection_hdr = "Connection: close\r\n";
24 static const char *proxy_connection_hdr = "Proxy-Connection: close\r\n";
25
26 typedef struct uri_info
27 {
28     char hostname[MAX_HOSTNAME_LEN];
29     char port[MAX_PORT_LEN];
30     char query[MAX_QUERY_LEN];
31 } Uri_info;
32
33 typedef struct headers
34 {
35     int count;
36     char *data[MAX_HEADERS_NUMBER];
37 } Headers;
38
39 static void proxy(int fd);
40 static int parse_headers(rio_t *rp, Headers *headers);
41 static int parse_uri(const char *uri, Uri_info *uri_info);
42 static void forward(int fd, Uri_info *uri_info, Headers *headers);
43 static void clear_headers(Headers *headers);
44 static void *thread(void *vargp);
45 static int build_cache_key(char key[], size_t length, Uri_info *uri_info);
46 static void clienterror(int fd, char *cause, char *errnum, char *shortmsg,
    char *longmsg);
47 static void servererror(int fd);
48
49 /* Thread safe buffer */
50 static sbuf_t sbuf;
```

Listing 1 (Cont.): proxy.c

```

51  /* 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
62      /* Blocking SIGPIPE signal */
63      /* SIGPIPE signal will be send by write() when
64       * the connection socket closes prematurely.
65       * An unhandled SIGPIPE signal terminates the program */
66      sigset_t mask;
67      if (Sigemptyset(&mask) < 0)
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;
78      if ((listenfd = Open_listenfd(argv[1])) < 0)
79          exit(1);
80
81      /* Creating thread pool */
82      if (sbuf_init(&sbuf, SBUFSIZE) < 0)
83          exit(1);
84
85      if (cache_init(&cache) < 0)
86          exit(1);
87
88      int i;
89      pthread_t tid;
90      for (i = 0; i < NTHREADS; i++)
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      {
99          if ((connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen)) < 0)
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,
106                      MAXLINE, 0);
107          printf("Accepted connection from (%s, %s)\n", client_host, port);
108          /* Adding request to the buffer */
109          if (sbuf_insert(&sbuf, connfd) < 0)
110              exit(1);
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

```

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

```

Listing 1 (Cont.): proxy.c

```

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

```

Listing 1 (Cont.): proxy.c

```

267     for (i = 0; i < headers->count; 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:") ||
272                 strstr(headers->data[i], "Connection:") ||
273                 strstr(headers->data[i], "Proxy-Connection"))
274             continue;
275
276         Rio_writen(clientfd, headers->data[i], strlen(headers->data[i]),
277                   client_close_conn);
278     }
279     if (!host_in_headers)
280     {
281         /* Sending a Host header */
282         sprintf(buf, "Host: ");
283         Rio_writen(clientfd, buf, strlen(buf), client_close_conn);
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 */
296     sprintf(buf, "%s", proxy_connection_hdr);
297     Rio_writen(clientfd, buf, strlen(buf), client_close_conn);
298     /* Sending the end of the request */
299     sprintf(buf, "\r\n");
300     Rio_writen(clientfd, buf, strlen(buf), client_close_conn);
301
302     /* jmp is the client close connection while the request is being
303        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
310     /* Creating a temp cache buffer */
311     size_t cached_len = 0;
312     int cacheable = 1;
313     char cache_buffer[MAX_OBJECT_SIZE];
314     char *cache_free = cache_buffer;
315     /* Transmitting the response from the client socket to the listening
316        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;
330                 cache_free += response_len;
331             }
332         }
333
334         Rio_writen(fd, buf, response_len, fd_closed_and_client_opened);
335     }
336

```

Listing 1 (Cont.): proxy.c

```

337     Close(clientfd);
338     /* Caching the response */
339     if (!(response_len < 0) && cacheable)
340     {
341         char key[MAXLINE];
342         if (build_cache_key(key, MAXLINE, uri_info))
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;
351     /* Parsing a protocol */
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++;
366     while (*uri == '/')
367         uri++;
368
369     /* Parsing a hostname */
370     index = buf;
371     while (*uri != ':' && *uri != '/')
372     {
373         if (*uri == '\0')
374             return -1;
375
376         *index++ = *uri++;
377     }
378
379     *index = '\0';
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
385     memcpy(uri_info->hostname, buf, hostname_len);
386     /* Parsing a port value */
387     index = buf;
388     if (*uri == ':')
389     {
390         uri++;
391         while (*uri != '/')
392         {
393             if (*uri == '\0')
394                 return -1;
395
396             *index++ = *uri++;
397         }
398
399         *index = '\0';
400         /* Filling Uri_info with the port value */
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 */
419     uri--;
420     index = buf;
421     while (*uri != '\0')
422         *index++ = *uri++;
423
424     *index = '\0';
425     /* 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->count = 0;
438     char buf[MAXLINE];
439     ssize_t line_len;
440     /* Reading first header line from the connected socket */
441     if ((line_len = Rio_readlineb(rp, buf, MAXLINE)) < 0)
442         return -2;
443
444     /* Checking if the header line length is less than MAXLINE */
445     if (line_len == MAXLINE - 1)
446         if (buf[line_len - 1] != '\n')
447             return -1;
448
449     /* While the line is not the end of the request */
450     while (strcmp(buf, "\r\n"))
451     {
452         if (headers->count > MAX_HEADERS_NUMBER)
453             return -1;
454
455         /* Copying the header to the Headers struct */
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->count++] = header;
462
463         /* Reading next line */
464         if ((line_len = Rio_readlineb(rp, buf, MAXLINE)) < 0)
465             return -2;
466
467         /* Checking if the header line length is less than MAXLINE */
468         if (line_len == MAXLINE - 1)
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 {
479     jmp_buf fd_closed;
480     if (setjmp(fd_closed))
481         return;
482

```

Listing 1 (Cont.): proxy.c

```

483     char buf[MAXLINE];
484     /* Print the HTTP response headers */
485     sprintf(buf, "HTTP/1.0 %s %s\r\n", errnum, shortmsg);
486     Rio_writen(fd, buf, strlen(buf), fd_closed);
487     sprintf(buf, "Content-type: text/html\r\n\r\n");
488     Rio_writen(fd, buf, strlen(buf), fd_closed);
489
490     /* Print the HTTP response body */
491     sprintf(buf, "<html><title>Proxy Error</title>");
492     Rio_writen(fd, buf, strlen(buf), fd_closed);
493     sprintf(buf, "<body bgcolor="
494               "ffffff"
495               ">\r\n");
496     Rio_writen(fd, buf, strlen(buf), fd_closed);
497     sprintf(buf, "%s: %s\r\n", errnum, shortmsg);
498     Rio_writen(fd, buf, strlen(buf), fd_closed);
499     sprintf(buf, "<p>%s: %s\r\n", longmsg, cause);
500     Rio_writen(fd, buf, strlen(buf), fd_closed);
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->count; 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)
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"
2
3  int sbuf_init(sbuf_t *sp, int n)
4  {
5      if ((sp->buf = Calloc(n, sizeof(int))) < 0)
6          return -1;
7
8      sp->n = n;
9      sp->front = sp->rear = 0;
10     if (Sem_init(&sp->mutex, 0, 1) < 0)
11         return -1;
12
13     if (Sem_init(&sp->slots, 0, n) < 0)
14         return -1;
15
16     if (Sem_init(&sp->items, 0, 0) < 0)
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)
30         return -1;
31
32     if (Sem_wait(&sp->mutex) < 0)
33         return -1;
34
35     sp->buf[(++sp->rear) % (sp->n)] = item;
36     if (Sem_post(&sp->mutex) < 0)
37         return -1;
38
39     if (Sem_post(&sp->items) < 0)
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)
48         return -1;
49
50     if (Sem_wait(&sp->mutex) < 0)
51         return -1;
52
53     *item = sp->buf[(++sp->front) % (sp->n)];
54     if (Sem_post(&sp->mutex) < 0)
55         return -1;
56
57     if (Sem_post(&sp->slots) < 0)
58         return -1;
59
60     return 0;
61 }

```

Listing 2: sbuf.c

```

1  #include "cache.h"
2
3  /* Max size of a cacheable request */
4  #define MAX_OBJECT_SIZE 102400
5  #define MAX_CACHE_SIZE 1049000
6
7  #define PAYLOAD(node_p) ((char *)node_p + sizeof(CacheNode))
8  #define NODE_SIZE(payload_size) (sizeof(CacheNode) + payload_size)
9
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 {
17     cache->total_size = 0;
18     cache->head = NULL;
19     if (Sem_init(&cache->readers_lock, 0, 1) < 0)
20         return -1;
21
22     if (Sem_init(&cache->writers_lock, 0, 1) < 0)
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)
35         return -1;
36
37     /* LRU eviction */
38     while (cache->total_size + length > MAX_CACHE_SIZE)
39     {
40         CacheNode *to_evict = cache->lru_head->item;
41         cache->total_size -= to_evict->size;
42         remove_item(cache, to_evict);
43         remove_lru(cache, to_evict->lru);
44         printf("Evicted from the cache: %s, freed %d bytes.\n",
45              to_evict->key, (int)to_evict->size);
46         free(to_evict->lru);
47         free(to_evict);
48     }
49
50     /* Creating a new cache node */
51     CacheNode *new_item;
52     if ((new_item = Malloc(NODE_SIZE(length))) < 0)
53         return -1;
54
55     /* Creating a key copy */
56     char *key_copy;
57     if ((key_copy = Malloc(strlen(key) + 1)) < 0)
58         return -1;
59
60     memcpy(key_copy, key, strlen(key) + 1);
61     /* Allocating a new lru node */
62     LruNode *new_lru;
63     if ((new_lru = Malloc(sizeof(LruNode))) < 0)
64         return -1;
65
66     new_lru->item = new_item;
67     /* Updating item fields */
68     new_item->key = key_copy;
69     new_item->size = length;
70     new_item->lru = new_lru;
71     /* Copying data to the cache node */
72     void *data = PAYLOAD(new_item);
73     memcpy(data, buf, length);

```

Listing 3 (Cont.): cache.c

```

73     /* Updating linked lists */
74     prepend_item(cache, new_item);
75     append_lru(cache, new_lru);
76     printf("Written to the cache %s. %d bytes.\n", new_item->key,
77           (int)new_item->size);
77     if (Sem_post(&cache->writers_lock) < 0)
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;
86     /* Acquiring writers lock if it is the first reader */
87     if (Sem_wait(&cache->readers_lock) < 0)
88         return -1;
89
90     cache->readers_count++;
91     if (cache->readers_count == 1)
92         if (Sem_wait(&cache->writers_lock) < 0)
93             return -1;
94
95     if (Sem_post(&cache->readers_lock) < 0)
96         return -1;
97
98     /* Key sequential lookup */
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)
106                return -1;
107
108            remove_lru(cache, current->lru);
109            append_lru(cache, current->lru);
110            if (Sem_post(&cache->readers_lock) < 0)
111                return -1;
112
113            /* Copying the cached value */
114            if ((*item = Malloc(current->size)) < 0)
115                return -1;
116
117            memcpy(*item, PAYLOAD(current), current->size);
118            *length = current->size;
119            is_cached = 1;
120            break;
121        }
122    }
123
124    /* Releasing writers lock if it is the last reader */
125    if (Sem_wait(&cache->readers_lock) < 0)
126        return -1;
127
128    cache->readers_count--;
129    if (cache->readers_count == 0)
130        if (Sem_post(&cache->writers_lock) < 0)
131            return -1;
132
133    if (Sem_post(&cache->readers_lock) < 0)
134        return -1;
135
136    if (is_cached)
137        printf("Cache hit %s\n", key);
138    else
139        printf("Cache miss %s\n", key);
140
141    return is_cached;
142 }
143
144 /* Appending an item to the end of the LRU linked list */

```

Listing 3 (Cont.): cache.c

```

145 static void append_lru(Cache *cache, LruNode *item)
146 {
147     item->next = NULL;
148     item->prev = cache->lru_tail;
149     if (cache->lru_tail != NULL)
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;
162     if (prev != NULL)
163         prev->next = next;
164     else
165         cache->lru_head = next;
166
167     if (next != NULL)
168         next->prev = prev;
169     else
170         cache->lru_tail = prev;
171 }
172
173 /* Prepending an item to the start of the cached items linked list */
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;
190     if (prev != NULL)
191         prev->next = next;
192     else
193         cache->head = next;
194
195     if (next != NULL)
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 ./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