시스템 프로그래밍 과제 5 proxylab 레포트

Writing a Caching Web Proxy

2019-13674
양현서
바이오시스템소재학부

Contents

1 Introduction

세상에 존재하는 웹 서비스들은 수천 수백만개의 클라이언트로부터 요청을 처리해야 하며 또 현대는 글로벌 시대가 도래하여 클라이언트와 서버 사이의 물리적 거리가 증가하였다. 물론 빛의 속도로 통신을 하긴 하지만, 물리적 거리가 길어짐으로써 발생하는 통신 비용이 존재하며, 잦은 원거리 접속은 전체적인 웹망 사용에 부담을 준다. 따라서 사람들은 웹 캐시 프록시 서버들을 운영하기 시작하였고, 이러한 프록시 서버들은 본서버의 내용을 캐시하여 클라이언트에게 보다 빠르게 컨텐츠를 전달해주며, 본 서버의 부담을 줄여준다. 이 랩에서도 비슷하게, GET요청만 처리할 수 있는 간단한 웹 캐시 프록시 서버 프로그램을 만든다.

2 Part 1: Implementing a sequential web proxy

첫 번째 과정에서는 우선 단순히 클라이언트의 요청을 해석하여 서버에 그대로 전달하는 프록시 서버를 만든다. 여기에 암호화 기능을 넣으면 VPN으로 사용할 수도 있을 것이다.
본인은 handout에 일러진 대로 firefox에 proxy 서버를 설정하여 테스트를 하기로 했는데, firefox 브라우저가 지속적으로 google 등에 POST 요청과 CONNECT 요청을 보내고, Part 2의 concurrent server 기능이 완성되지 않았을 때 계속 그러한 요청들에 의해 블록이 되어 테스트가 방해가 되어, POST와 CONNECT 요청에 대해서도 사소하고 미봉책스러운 기능을 만들어 두었다.
원래는 강의 자료에 쓰여 있는 것을 따라하여 서버의 기초 뼈대를 만드려 하였는데, tiny.c를 보니 csapp의 코드를 이용해도 된다는 것이 생각나서 csapp.c의 여러 유틸리티 함수들을 이용하여 기초 main 함수를 작성하였다.

```
60 int main(int argc, char** argv)
61 {
62     char hostname[MAXLINE];
63     char clientPort[MAXLINE];
64     if(argc != 2) {
65         fprintf(stderr, "Usage: %s <port>\n", argv[0]);
66         exit(1);
67     }
68 
69     int portNumber = atoi(argv[1]);
70     printf("Runnion on %d port\n", portNumber);
```

```
71      int listenSocket; // = Socket(AF_INET, SOCK_STREAM, 0);
72      struct sockaddr_in clientAddr;
73      listenSocket = Open_listenfd(argv[1]);
74      initCache(&cache);
75      initSignal();
76      pthread_t tid;
77      while(1) {
78          const int clientlen = sizeof(clientAddr);
79          int connfd = Accept(listenSocket, (SA *) &clientAddr, &clientlen);
80          Getnameinfo((SA *) &clientAddr, clientlen, hostname, MAXLINE, clientPort,
            ↪  MAXLINE, 0);
81          printf("Accepted connection from (%s, %s)\n", hostname, clientPort);
82          // hp = gethostbyaddr((const char
            ↪  *)&clientAddr.sin_addr.s_addr,sizeof(clientAddr.sin_addr.s_addr),
            ↪  AF_INET);
83          int *connfdp = malloc(sizeof(int));
84          *connfdp = connfd;
85          Pthread_create(&tid, NULL, proxy_thread, connfdp);
86          // Close(connfd);
87      }
88      freeCache(&cache);
89      return 0;
90  }
```

74 76행, 83 85행, 88행은 part 2와 3에서 삽입된 것이다. 원래 코드에서는 단순히 doProxy를 호출하였다.

```
136 int doProxy(int fd) {
137     struct parsedRequest parsed = parseRequest(fd);
138     printf("Parse request done\n");
139     if(!parsed.method || (strncmp(parsed.method, "GET", 3)!=0 &&
        ↪  strncmp(parsed.method, "POST", 4)!=0)) {
140         printf("[PROXY] ============ Ignoring method %s ============\n",
            ↪  parsed.method);
141         freeParsedRequest(&parsed);
142         return -1;
143     }
144     struct response result = execRequest_Cached(&parsed);
145     printf("execRequest done\n");
146     replyRequest(fd, &result);
147     printf("replyRequest done\n");
148     freeParsedRequest(&parsed);
149     printf("doProxy about to return\n");
150     return 0;
151 }
```

`doProxy` 함수에서는 요청을 해석하고 GET이나 POST 요청이 아니면 빈 응답을 내보내며, 제대로 된 요청일 경우 처리하고 결과를 답신한다. 그 후 메모리를 해제하고 종료한다. 여기서 `execRequest_Cached` 대신 `execRequest`를 호출하였었다. 캐싱 기능 없이 바로 요청을 하고 결과를 돌려주는 함수이다.

2.1 요청 해석

요청을 해석한 결과를 저장하기 위해 `parsedRequest`라는 구조체를 정의하였다.

```
25  struct parsedRequest {
26      char * uri;
27      char * httpVersion;
28      char * method;
29      char * host;
30      char has_user_agent;
31      char has_connection;
32      char has_proxy_connection;
```

```
33      char has_host;
34      ssize_t content_length;
35      char * content;
36      struct httpParam * params;
37  };
```

요청의 uri와 http version, http method, host, 그리고 http param에 인자들을 정해진 값으로 치환하기 위한 보조 변수들, http request body를 위한 메모리 블록 포인터와 사이즈, http 매개 변수들을 저장할 링크드 리스트를 포함한다.
http 요청의 매개 변수들은 다음과 같은 구조체를 이용해 링크드 리스트 형태로 저장한다.

```
19  struct httpParam {
20      char * field;
21      char * value;
22      struct httpParam * next;
23  };
```

`parseRequest` 함수에서는 주어진 descriptor에서 요청을 읽어들이고, `parsedRequest` 구조체를 작성하여 리턴하는 역할을 한다.

```
153 struct parsedRequest parseRequest(int fd) {
154     size_t n;
155     char buf[MAXLINE];
156     char originalBuf[MAXLINE];
157     rio_t rio;
158     Rio_readinitb(&rio, fd);
159     struct parsedRequest result;
160     result.params = NULL;
161     result.content_length = 0;
162     result.uri = NULL;
163     result.method =  NULL;
164     result.httpVersion = NULL;
165     result.host = NULL;
166     result.content = NULL;
167     result.has_host = 0;
168     result.has_proxy_connection = 0;
169     result.has_connection = 0;
170     result.has_user_agent = 0;
171     struct httpParam * iterator = NULL;
172     size_t content_length = 0;
```

변수 초기화 부분이다.

```
174     while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
175         //
176         printf("line: %s", buf);
177         strncpy(originalBuf, buf, MAXLINE);
178         char * saveptr;
179         char * first = strtok_r(buf, " ", &saveptr);
180         if(!first)
181             continue;
182         if(strncmp(first, "\r\n", 2)==0) {
183             printf("Last buf: %s\n", originalBuf);
184             break;
185         }
186         if((strcasecmp(first, "GET") == 0) || (strcasecmp(first, "POST") == 0)) {
187             // GET or POST
188             char * uri = strtok_r(NULL, " ", &saveptr);
189             char * httpVersion = strtok_r(NULL, " ", &saveptr);
190             char * method = first;
191             printf("uri: %s, version: %s\n", uri, httpVersion);
192             result.uri = strdup(uri);
```

```
193             result.httpVersion = strdup(httpVersion);
194             result.method = strdup(method);
195             continue;
196         } else {
```

HTTP 요청이 GET인지 POST인지 확인하고, HTTP 버전을 확인한다.

```
197             char *field = strtok_r(originalBuf, ":", &saveptr);
198             char *value = strtok_r(NULL, ":", &saveptr);
199             // printf("Field: %s, value: %s\n", field, value);
200             struct httpParam *param = malloc(sizeof(struct httpParam));
201             param->next = NULL;
202             param->field = strdup(trimwhitespace(field));
203             param->value = strdup(trimwhitespace(value));
204             if(strcmp(param->field, connection)==0) {
205                 free(param->value);
206                 param->value = strdup(connection_close_hdr);
207                 result.has_connection = 1;
208             } else if(strcmp(param->field, proxy_connection) == 0) {
209                 free(param->value);
210                 param->value = strdup(proxy_connection_close_hdr);
211                 result.has_proxy_connection = 1;
212             } else if(strcmp(param->field, user_agent)==0) {
213                 free(param->value);
214                 param->value = strdup(user_agent_hdr);
215                 result.has_user_agent = 1;
216             } else if(strcmp(param->field, host) == 0) {
217                 result.has_host = 1;
218                 result.host = strdup(param->value);
219             } else if(strcmp(param->field, contentLengthField) == 0) {
220                 result.content_length = atoi(param->value);
221             }
222             if(iterator == NULL) {
223                 iterator = param;
224                 result.params = iterator;
225             } else {
226                 iterator -> next = param;
227                 iterator = param;
228             }
229             continue;
230         }
231
232     }
```

HTTP 요청 매개 변수들을 처리하는데, 미리 정한 값으로 치환하기도 한다.

```
233     if(result.content_length > 0) {
234         printf("Content length: %ld\n", result.content_length);
235         int n = 0;
236         int totalN = 0;
237         result.content = NULL;
238         while((n=Rio_readnb(&rio, buf, MAXLINE > result.content_length ?
            ↪  result.content_length: MAXLINE))>0) {
239             // printf("something");
240             // printf("n:%d\n", n);
241             totalN += n;
242             result.content = realloc(result.content, totalN);
243             memcpy(result.content + totalN - n, buf, n);
244             if(totalN == result.content_length)
245                 break;
246         }
247         // result.content_length = totalN;
```

```
248             printf("Wrote content of size %d\n", totalN);
249 
250         }
```

요청에 content가 포함된 경우를 처리한다. while 문에서 괄호와 부등호 순서를 잘못하여 알 수 없는 버그에 빠진 적이 있었다. 조심하여야 한다.

```
251         if(!result.has_host) {
252             result.host = getHostFromURI(result.uri);
253         }
254         return result;
255     }
```

호스트 매개변수가 지정되지 않았을 경우 uri에서 호스트를 파싱한다. 메모리 해제 문제 때문에 이 `parseRequest` 함수 내부에서만 호출한다.

2.2 요청 실행

이제는 실제로 요청을 실행하는 부분을 구현한다. 이 부분에서는 프록시 서버가 상대 서버에 대한 클라이언트 역할이 된다.

```
267 struct response execRequest(struct parsedRequest * parsed) {
268     struct response responseData; // = malloc(sizeof(struct response));
269     responseData.data = NULL;
270     responseData.length = -1;
271     rio_t rio;
272     char * hostname = parsed->host;
273     printf("Hostname: %s has_host: %d host: %s\n", hostname, parsed->has_host,
        ↪  parsed->host);
274     char buf[MAXLINE];
275     int port = guessPortFromURI(parsed->uri);
276     printf("Port: %d\n", port);
277     int clientfd;
278     struct hostent *  hp;
279     struct sockaddr_in serveraddr;
```

필요한 변수들을 초기화하는 부분이다. 처음에는 동적으로 메모리를 할당하려 했지만 메모리 누수 제거 과정에서 굳이 동적 메모리 할당이 필요하지 않을 것 같아서 지역변수로 선언하였다. `guessPortFromURI` 함수를 호출하여 uri에 포트를 명시할 경우 그 포트로 접속하게 하였다.

```
280     if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
281         return responseData;
282     if ((hp = gethostbyname(hostname)) == NULL)
283         return responseData;
284     bzero((char *) &serveraddr, sizeof(serveraddr));
285     serveraddr.sin_family = AF_INET;
286     bcopy((char *)hp->h_addr_list[0], (char *)&serveraddr.sin_addr.s_addr,
        ↪  hp->h_length);
287     serveraddr.sin_port = htons(port);
288     if (connect(clientfd, (SA *) &serveraddr, sizeof(serveraddr)) < 0)
289         return responseData;
```

소켓을 만들고 서버에 접속한다. 그 과정에서 오류가 나면 그 상태의 `reponseData`를 리턴하는데, `reponseData.length`를 미리 -1로 만들어 두어서 오류임을 알 수 있게 했다. 이것이 포인터를 리턴하지 않는 것의 거의 유일한 단점인 것 같다.

```
290     printf("Requesting...\n");
291     char * relativeURI = makeRelativeURI(parsed -> uri);
292     sprintf(buf, "%s %s HTTP/1.0\r\n", parsed -> method,  relativeURI);
293     free(relativeURI);
294     Rio_writen(clientfd, buf, strlen(buf));
295     // printf("Wrote %s", buf);
296     if(!parsed->has_host) {
```

```
297             sprintf(buf, "Host: %s\r\n", hostname);
298             Rio_writen(clientfd, buf, strlen(buf));
299             // printf("Wrote %s", buf);
300         }
301         if(!parsed->has_user_agent) {
302             sprintf(buf, "%s: %s\r\n", user_agent, user_agent_hdr);
303             Rio_writen(clientfd, buf, strlen(buf));
304             // printf("Wrote %s", buf);
305         }
306         if(!parsed->has_connection) {
307             sprintf(buf, "%s: %s\r\n", connection, connection_close_hdr);
308             Rio_writen(clientfd, buf, strlen(buf));
309             // printf("Wrote %s", buf);
310         }
311         if(!parsed->has_proxy_connection) {
312             sprintf(buf, "%s: %s\r\n", proxy_connection, proxy_connection_close_hdr);
313             Rio_writen(clientfd, buf, strlen(buf));
314             // printf("Wrote %s manually", buf);
315         }
```

요청을 소켓에 쓰기 시작한다. `makeRelativeURI` 함수를 호출하여 relative uri를 구하고, 그것에 맞추어 요청을 보낸다. 프록시 서버에서 고정으로 보내는 매개 변수들을 적절하게 추가한다.

```
316         struct httpParam * iterator = parsed->params;
317         while(iterator) {
318             sprintf(buf, "%s: %s\r\n", iterator->field, iterator->value);
319             Rio_writen(clientfd, buf, strlen(buf));
320             // printf("Wrote %s", buf);
321             iterator = iterator -> next;
322         }
323         Rio_writen(clientfd, "\r\n", strlen("\r\n"));
```

요청의 나머지 HTTP 매개 변수들을 전달한다. `"\r\n"`을 전송한다.

```
324         if(parsed->content_length > 0) {
325             Rio_writen(clientfd, parsed->content, parsed->content_length);
326         }
```

요청에 body가 있었을 경우 그것도 같이 전송한다.

```
328         ssize_t n = 0;
329         int totalN = 0;
330         printf("Reading\n");
331 
332         Rio_readinitb(&rio, clientfd);
333         while((n=Rio_readnb(&rio, buf, MAXLINE))>0) {
334             // printf("something");
335             printf("n:%ld\n", n);
336             totalN += n;
337             responseData.data = realloc(responseData.data, totalN);
338             memcpy(responseData.data + totalN - n, buf, n);
339         }
340         responseData.length = totalN;
341         printf("total N: %d\n", totalN);
342         Close(clientfd);
343         return responseData;
344 }
```

서버로부터 들어오는 응답을 그대로 읽어 버퍼에 계속 저장하고, 최종 응답 길이를 `reponseData`에 저장한다. 그리고 소켓을 닫는다.

2.3 응답 전달

단순하게 서버의 응답을 클라이언트에 전달해 준다.

```
368
369 void replyRequest(int fd, struct response * result) {
370     if(result)
371         Rio_writen(fd, result->data, result->length);
372     else
373         printf("Warning: result is null\n");
```

2.4 보조 함수들

whitespce 제거

문자열에서 whitespace를 제거한다. Stack overflow를 참고하였다.

```
369 void replyRequest(int fd, struct response * result) {
370     if(result)
371         Rio_writen(fd, result->data, result->length);
372     else
373         printf("Warning: result is null\n");
374 }
```

URI에서 호스트 파싱

원시적 오토마타를 이용하여 매칭을 한다. ://에서 시작하여 첫 번째 /를 만나기 전까지의 문자열을 잘라 리턴한다.

```
376 char * getHostFromURI(char * uri) {
377     // ://부터 :이나 / 전까지 뽑아낸다
378     int state = 0;
379     char * start;
380     int len = 0;
381     while(*uri) {
382         switch(state) {
383             case 0:
384                 if(*uri == ':') {
385                     state++;
386                 } else {
387                     state = 0;
388                 }
389                 break;
390             case 1:
391                 if(*uri == '/') {
392                     state++;
393                 } else {
394                     state = 0;
395                 }
396                 break;
397             case 2:
398                 if(*uri == '/') {
399                     state++;
400                     start = uri + 1;
401                 } else {
402                     state = 0;
403                 }
404                 break;
405             case 3:
406                 if(*uri == '/') {
407                     state++;
408                 } else {
```

```
                    len++;
                    state = 3;
                }
                break;
            case 4:
                break;
        }
        uri++;
    }
    char * result = malloc(sizeof(char) * (len + 1));
    result[len] = '\0';
    memcpy(result, start, len);
    return result;
}
```

URI에서 포트 파싱

원시적 오토마타를 이용하여 매칭을 한다. 만일 찾지 못했을 경우 HTTP 기본 포트인 80을 반환한다. 기본적으로는 ://로부터 첫번째 :와 첫 번째 /가 숫자이면 그것을 포트로 간주하고 숫자로 변환해 반환한다.

```
int guessPortFromURI(char * uri) {
    if(uri == NULL)
        return 80;
    printf("GuessPortFromURI uri:%s\n", uri);
    int state = 0;
    char *  portStart = NULL;
    int port = 0;
    while(*uri) {
        char ch = *uri;
        if(*uri == ':') {
            state = 1;
            portStart = NULL;
        }
        else if(isdigit(ch)) {
            if(state == 1) {
                portStart = uri;
                state = 2;
            } else if(state == 2){

            } else {
            }
        }
        else if(ch == '/') {
            if(state == 2) {
                // portEnd =  uri-1;
                state = 3;
            }else {
                portStart = NULL;
                state = 0;
            }
        } else {
            portStart = NULL;
            state = 0;
        }
        if(state == 3)
            break;
        uri++;
    }
    if(portStart)
        port = atoi(portStart);
```

```
515     if(port == 0)
516         port = 80;
517     return port;
518 }
```

URI에서 상대 경로 얻기

```
424 char *makeRelativeURI(char * uri) {
425     if(*uri == '/') {
426         return strdup(uri);
427     }
428     if(strstr(uri, "://") == NULL) {
429         int len = strlen(uri);
430         char * result = malloc(sizeof(char) * (len + 2));
431         result[0] = '/';
432         strcpy(result+1, uri);
433         result[len+1] = '\0';
434         return strdup(uri);
435     }
436     int state = 0;
437     char * pathStart = NULL;
438     while(*uri) {
439         char ch = *uri;
440         switch(state) {
441             case 0:
442                 if(ch == ':') {
443                     state++;
444                 }
445                 break;
446             case 1:
447                 if(ch == '/') {
448                     state++;
449                 } else {
450                     state = 0;
451                 }
452                 break;
453             case 2:
454                 if(ch == '/') {
455                     state++;
456                 } else {
457                     state = 0;
458                 }
459                 break;
460             case 3:
461                 if(ch == '/') {
462                     state++;
463                     pathStart = uri;
464                 }
465                 break;
466         }
467         uri++;
468     }
469     if(state == 4) {
470         return strdup(pathStart);
471     }
472     return strdup("/");
473 }
```

용어가 조금 이상하긴 하지만 '/'로 시작하는 URI의 경우 호스트가 없는 상대 경로이므로 그대로 복제하여 리턴한다. 또한 :// 의 uri scheme이 존재할 경우 // 뒤에 호스트가 존재할 것이므로 그 직후에

나오는 /를 기준으로 uri를 잘라 뒷부분을 리턴한다. 만약 uri scheme이 없을 경우 첫글자가 /도 아니면 앞에 /가 없는 상대 경로로 간주하고 /를 앞에 붙인 상대 경로를 리턴한다.

3 Part 2: Dealing with multiple concurrent requests

서버가 concurrent하게 요청을 처리하는 방법은 세 가지가 있다. 첫째는 프로세스를 포크하여 여러 프로세스들을 이용하여 처리하는 방법이고, 두 번째는 스레드를 생성하는 것이고 셋째는 select 함수를 이용하는 것이다. 이 중에서 가장 보편적이고, fork보다는 로드가 적은 스레드 생성으로 concurrent하게 처리하도록 하는 것이 handout의 지시 사항이다.
다시 `main` 함수를 보면

```
83          int *connfdp = malloc(sizeof(int));
84          *connfdp = connfd;
85          Pthread_create(&tid, NULL, proxy_thread, connfdp);
```

각 디스크립터를 힙에 복사한 후 그 각각의 포인터를 스레드의 인자로 넘겨주며 스레드를 만들고 있다. 이것은 의도치 않은 변수 공유를 막기 위한 것이다.

```
92  void * proxy_thread(void *vargp) {
93      int connfd = *((int *)vargp);
94      Pthread_detach(pthread_self());
95      free(vargp);
96      doProxy(connfd);
97      printf("DoProxy done\n");
98      Close(connfd);
99      return NULL;
100 }
```

94행에서 이 스레드 함수가 리턴하면 바로 자신이 reap 되도록 자신을 detach하고 있다. 또한 93, 95행에서 소켓 디스크립터 숫자를 얻어온 후 바로 free를 하여 메모리를 반환하고 있다. 그다음 doProxy 함수를 호출하여 이 함수가 각각의 스레드에서 실행되게 하고, 소켓을 닫아 주었다.
아직까지는 동기화 관련하여 문제가 발생하지 않는다. 여러 스레드가 공동으로 사용하는 자료구조가 없다시피 하기 때문이다.

4 Part 3: Caching web objects

4.1 LRU Cache

LRU cache에 대해 찾아보니, read나 write를 한 것을 '사용'이라고 간주할 때, 오랫동안 사용하지 않은 데이터를 제거하고 최근에 사용된 데이터들은 항상 캐시에 남아 있게 작동하는 캐시이다. 이것의 구현은 링크드 리스트와 해시맵으로 구현할 수 있을 것 같다. 링크드 리스트를 이용하여 어떤 노드가 '사용'되었을 때 맨 앞으로 가져오고, 해시맵을 이용하여 해당 노드를 빠르게 찾을 수 있게 하는 것이다. 이러한 생각을 바탕으로 LRU cache를 구현하였다.

```
150 /* key: uri */
151 int LRUCache_Hash(const char * key) {
152     int hash = 0;
153     while(*key) {
154         hash = ((((hash << 5) - hash))  + *key) % LRU_HASHMAPSIZE;
155         key++;
156     }
157     return hash;
158 }
```

문자열의 해싱은 위와 같이 설계하였다. key는 입력 uri로 하기로 하였다.

자료 구조

```
5   struct LRUNode {
6       int hash;
7       char * data;
```

```
8       int size;
9       struct LRUNode * next,
10                     * prev;
11  };
```

캐시된 데이터를 담을 링크드 리스트의 노드이다.

```
13  struct LRUHashListNode {
14      const char * key;
15      int hash;
16      struct LRUHashListNode * next;
17      struct LRUNode * data; // non null
18  };
```

해시맵에 충돌이 일어났을 때 해시값이 일치하는 여러 개의 노드에 대한 포인터를 담을 수 있는 링크드 리스트 노드이다. 캐시 충돌이 많아지면 트리 형태로 만들 수도 있지만, 현재는 그렇게 하지 않아도 성능이 잘 나온다.

```
20  struct LRUCache {
21      int totalSize;
22      struct LRUNode * front,
23                     * rear;
24      struct LRUHashListNode * hashToLRUNode[LRU_HASHMAPSIZE];
25  };
```

캐시를 총괄하는 구조체이다. 이 프로그램에서 한 개만 존재한다. 삽입과 삭제를 빠르게 하기 위해 front와 rear 변수를 두었다.

작동 구조

캐시를 이용하는 측은 `initCache`와 `freeCache` 함수, 그리고 `LRUCache_Get` 함수만을 호출한다. `LRUCache_Get` 함수를 호출할 때 캐시에 사용되는 키와 그 데이터를 생성하기 위한 metadata를 넘겨준다. metadata는 내부적으로 캐시 미스가 나서 캐시에 데이터를 채워야 할 때 `createData`콜백에 넘겨지는 정보이다. 즉, `LRUCache_Get` 함수 내부에서 캐시 히트가 나면 해당 노드를 맨 앞으로 당겨온 후 리턴하고, 캐시 미스가 나면 `createData` 함수를 이용하여 새로운 노드를 생성하여 맨 앞에 추가하고, 캐시의 용량이 정해진 크기 이내로 들어올 때까지 오래된 노드를 evict 시키는 것이다.

```
41  struct LRUNode * LRUCache_get(struct LRUCache *cache, const char * key, void *
    ↪  metadata) {
42      struct LRUHashListNode * listNode = LRUCache_Hash_Get(cache, key);
43      P(&semaphore);
44      if(listNode) { // pop the node to front of the list
45          struct LRUNode * node = listNode -> data;
46          if(node -> prev) {
47              node -> prev -> next = node -> next; // remove from list
48              node -> next -> prev = node -> prev;
49          }
50          // insert front
51          node -> next = cache->front;
52          node -> prev = NULL;
53          cache -> front -> prev = node;
54          cache -> front = node;
55          V(&semaphore);
56          return node;
57      } else {
```

먼저 키를 해시하여 해시맵 안에 노드가 존재하는지를 판단한다. 캐시 히트가 나라 경우에 해당 노드를 제일 앞으로 옮기고, 해당 노드를 리턴한다.

```
58          V(&semaphore);
59          // create data and insert
60          struct LRUNode * newNode = createData(key, metadata);
```

```
61          P(&semaphore);
62          newNode -> next = cache -> front;
63          newNode -> prev = NULL;
64          newNode -> hash = LRUCache_Hash(key);
65          if(cache -> front) {
66              cache -> front -> prev = newNode;
67          }
68          cache -> front = newNode;
69          cache -> totalSize += newNode -> size;
```

캐시 미스가 난 경우, 콜백 함수인 createData 함수를 호출하여 새로운 노드를 생성하고, 그 노드를 링크드 리스트의 맨 앞에 삽입한다.

```
70          while(cache -> totalSize > MAX_CACHE_SIZE) {
71              // evict oldest data
72              struct LRUNode * rearIterator = cache -> front;
73              while(rearIterator -> next) {
74                  rearIterator = rearIterator -> next;
75              }
76              cache -> rear = rearIterator;
77              struct LRUNode * toDelete = cache -> rear;
78              // remove from hashmap
79              int hash = toDelete -> hash;
80              struct LRUHashListNode * iterator =  cache -> hashToLRUNode[hash];
81              struct LRUHashListNode * previt = NULL; // &(cache ->
                ↪  hashToLRUNode[hash]);
82              while(iterator) {
83                  struct LRUHashListNode * next = iterator -> next;
84                  if(iterator -> data == toDelete) {
85                      if(previt != NULL)
86                          previt -> next = iterator -> next;
87                      else
88                          cache -> hashToLRUNode[hash] = iterator -> next;
89                      free(iterator);
90                      break;
91                  }
92                  previt = iterator;
93                  iterator = next;
94              }
95
96              free(toDelete -> data);
97              cache -> totalSize -= toDelete->size;
98              if(toDelete -> prev)
99                  toDelete -> prev -> next = NULL;
100             cache -> rear = toDelete -> prev;
101             free(toDelete);
102         }
```

만약 캐시의 전체 크기가 허용된 크기를 넘어선다면, 마지막 노드부터 eviction을 시행한다.

```
103         V(&semaphore);
104         LRUCache_Hash_Put(cache, key, newNode);
105         return newNode;
106     }
107 }
```

링크드 리스트에서 추가된 노드를 해시맵에도 등록한다.
위 함수 내부에서 호출되는 콜백함수 `createData`함수는 다음과 같다.

```
118 struct LRUNode * createData(const char * key, void * metadata) {
119     printf("CreateData called\n");
120     struct parsedRequest * request = (struct parsedRequest * ) metadata;
```

```
121     struct response response =  execRequest(request);
122     struct LRUNode * node = malloc(sizeof(struct LRUNode));
123     if(response.length == -1) {
124         printf("Warning: response is NULL");
125         node -> data = NULL;
126         node -> size = 0;
127     } else {
128         node -> data = response.data;
129         node -> size = response.length;
130     }
131     node -> next = NULL;
132     node -> prev = NULL;
133     return node;
134 }
```

execRequest를 호출하여 결과를 반환받고, 새로운 LRUNode 를 생성하여 결과를 얕은 복사한 후 반환해준다.

```
109 struct LRUHashListNode * LRUCache_Hash_Get(struct LRUCache *cache, const char *
    ↪  key) {
110     int hash = LRUCache_Hash(key);
111     P(&semaphore);
112     struct LRUHashListNode * listNode = cache->hashToLRUNode[hash];
113     if(!listNode) {
114         V(&semaphore);
115         return NULL;
116     }
117     struct LRUHashListNode * iterator = listNode;
118     while(iterator) {
119         if(listNode->key && key &&  strcmp(key, listNode->key) == 0) {
120             V(&semaphore);
121             return listNode;
122         }
123         iterator = iterator -> next;
124     }
125     V(&semaphore);
126     return NULL;
127 }
```

해시맵에서 해시를 이용해 데이터를 검색한다. 만약 해당 해시 위치에 충돌이 있을 경우, 키를 비교해가며 정확한 노드를 탐색하고, 실패시 NULL을 반환한다.

```
129 void LRUCache_Hash_Put(struct LRUCache * cache, const char * key, struct LRUNode *
    ↪  data) {
130     int hash = LRUCache_Hash(key);
131     struct LRUHashListNode * newListNode = malloc(sizeof(struct LRUHashListNode));
132     // iterator -> next = newListNode;
133     newListNode -> data = data;
134     newListNode -> key = strdup(key);
135     newListNode -> hash = hash;
136     newListNode -> next = NULL;
137     P(&semaphore);
138     struct LRUHashListNode * temp = cache -> hashToLRUNode[hash];
139     if(temp) {
140         struct LRUHashListNode * iterator = temp;
141         while(iterator -> next)
142             iterator = iterator -> next;
143         iterator -> next = newListNode;
144     } else {
145         cache -> hashToLRUNode[hash] = newListNode;
146     }
```

```
147     V(&semaphore);
148 }
```

key, data쌍을 해시맵에 삽입한다. 충돌이 일어날 경우 링크드 리스트의 맨 끝에 삽입해 준다.

4.2 보조 함수 - 시그널 핸들러를 이용하여 Ctrl+C를 눌렀을 때 메모리를 해제 & SIGPIPE 차단

캐시 메모리는 종료할 때가 되서야 메모리를 전부 해제하게 되는데, 이 프록시 서버 프로그램은 따로 종료 조건을 받아들이지 않기 때문에 시그널을 보내 종료시키게 된다. 이 경우 main 함수가 리턴하며 종료하는 방식이 통하지 않을 수 있기 때문에 적절한 시그널 처리를 통해 종료 전 메모리를 해제해 주는 것이 바람직하다.

```
102 struct sigaction old_action;
103
104 void ctrlC_handler(int sig_no) {
105     freeCache(&cache);
106     exit(0);
107 }
108
109 void initSignal() {
110     struct sigaction action;
111     memset(&action, 0, sizeof(action));
112     action.sa_handler = &ctrlC_handler;
113     sigaction(SIGINT, &action, &old_action);
114     signal(SIGPIPE, SIG_IGN);
115 }
```

SIGINT 핸들러를 등록하여 SIGINT 시그널을 수신할 시 캐시 메모리를 해제하게 하였다. 또한 SIGPIPE 시그널에 대해서는 무시하게 설정하였다. 이와 더불어 csapp.c에서 오류가 발생할 시 exit 함수를 호출하는 부분을 주석처리하였다.

5 Conclusion

5.1 어려웠던 점

메모리 누수, Segmentation fault

C로 짠 어느 프로그램이 다 그렇듯이, 자료 구조에서 동적 메모리를 할당하게 되면 메모리 누수가 생기는 버그가 생길 수 있다. 이번 랩에서도 메모리 누수가 엄청나게 발생하거나 잘못된 포인터 참조로 인해 어려움을 겪었었다. Valgrind를 사용할 줄은 몰랐지만, 사용하지 않는 것보다는 일의 진행에 도움을 줄 것 같아서 이용하였다. 실습 서버에 루트 권한 없이 valgrind를 설치하려 했는데 잘 안 되어서 로컬 머신에서 진행하였다. 이번 랩이 valgrind를 처음 사용해 보는 것이었는데 생각보다 메모리 누수가 일어난 위치를 자세하게 알려주고, 여러 번의 랩 수업에서 쌓인 segmentation fault 처리 노하우가 쌓여서 며칠 만에 메모리 누수를 없애고 잘 작동하게 할 수 있었다.

5.2 놀라웠던 점

프록시 서버가 생각보다 잘 만들어져서 그런지는 모르겠지만 이 프록시를 이용하여 etl에 로그인하고 공지사항을 확인하는 것까지 원활하게 진행이 되어서 놀라웠다.