Lab Assignment 5: Proxy Lab

공과대학 컴퓨터공학부
2020-14378
윤교준 / Gyojun Youn
youngyojun@snu.ac.kr

개요

Object caching의 역할을 수행하는 thread 기반 concurrent HTTP proxy server를 구현하였다. 이를 구현하기 위하여 web response and request, sockets, concurrent threads, semaphores가 요구되었다.

이 과정은 크게 세 단계로 나뉘어 진행되었고, 직접 구현한 서버 프로그램을 Linux Firefox browser의 proxy server로 설정하여 테스트해본 결과, HTTP 웹 사이트에 정상적으로 접속할 수 있음을 확인하였다.

구현

Part 1

단순하게 단 하나의 client에 대하여 server와 naive connection을 맺어주는 simple sequential web proxy를 구현해야 한다.

Given listening port로 들어오는 모든 connections을 accept하고, client의 request를 server에게 그대로 전달한 다음, server의 response를 그대로 client에게 전달하면 충분하다.

그러나, client의 request line과 headers를 parse해야만 target server과 통신할 수 있기 때문에, 생각보다 많은 구현량이 요구되었다.

먼저, 인자로 주어지는 수를 listening port로 하여 열어두어야 한다. 이후, client의 connection을 accept하고 proxy service를 수행하는 일련의 과정을 sequential하게 수행해야 한다.

```c
int main(int argc, char *argv[])
{
  /* The listening port must be given as an argument. */
  if (argc < 2) {
    fprintf(stderr, "Usage: %s <listening port>\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  const int listen_fd = Open_listenfd(argv[1]);

  struct sockaddr_storage client_addr;
  socklen_t client_addrlen;

  while (1) {
```

```
    /* Accept connection. */
    client_addrlen = sizeof(client_addr);
    const int conn_fd = Accept(listen_fd, (SA*)(&client_addr), &client_addrlen);

    /* Do transaction service. */
    run_service(conn_fd);
    Close(conn_fd);
  }

  return EXIT_SUCCESS;
}
```

`run_service`는 먼저, 인자로 받은 connection fd를 통해 client의 request를 받고, 첫 줄의 request line을 parse하여 request uri의 hostname과 relative path, port를 모두 알아내어야 한다. 이 parsing 과정은 `parse_request_line` 함수가 수행하며, 그 구현은 기계적이다.

```
/*
 * run_service - Do HTTP services for the client
 *    with given connection fd.
 */
void run_service(const int conn_fd)
{
  rio_t client_rio;
  rio_readinitb(&client_rio, conn_fd);

  request_line req_line;
  if (parse_request_line(&req_line, &client_rio)) {
    fprintf(stderr, "Failed to parse the request line\n");
    return;
  }

  // 후략
}
```

이후, request headers를 모두 읽어들여 특정 fields에 대한 예외 처리를 한 후, hostname의 server에게 새로운 request headers를 보내야 한다. 이 작업은 `parse_request_headers`에서 수행된다.

```
  char port[16];
  sprintf(port, "%d", req_line.port);

  const int server_fd = Open_clientfd(req_line.hostname, port);
  if (server_fd < 0) {
    fprintf(stderr, "Connection to the server failed\n");
    return;
  }

  if (parse_request_headers(server_fd, &req_line, &client_rio)) {
    fprintf(stderr, "Failed to send a request to the server\n");
    return;
  }
```

이제, request를 받은 server는 response를 보내준다. 여기에는 headers와 requested object가 모두 있지만, caching을 하지 않아도 되기에 아직은 이를 구분할 필요는 없다.

```
void run_service(const int conn_fd)
{
  // 전략

  rio_t server_rio;
  rio_readinitb(&server_rio, server_fd);

  char line[MAXLINE];
  ssize_t len;
  while (0 < (len = Rio_readnb(&server_rio, line, MAXLINE))) {
    Rio_writen(conn_fd, line, len);
  }

  Close(server_fd);
}
```

각 sub-function에 대해서는 part section 뒤에서 설명한다.

Part 2

이제, part 1의 코드를 concurrent하게 수정해야 한다. Concurrent program을 구현하는 방법은 다양하나, overhead와 구현량 간의 trade-off의 중용에 해당하는 thread 기반 방법을 채택하였다.

먼저, `csapp.c`의 wrapper 함수들은 에러 케이스에 대하여 `exit(0)`으로 대응한다. `exit` 함수는 해당 process의 모든 threads를 종료시키므로, 문제가 발생한 thread만 종료되도록 수정해야만 한다. 또한, additional process에서 발생한 오류에 대해서는 thread가 죽어버리지 않도록 사용할 함수를 신중하게 선택하여야 한다.

```
void unix_error(char *msg) /* Unix-style error */
{
```

```
  fprintf(stderr, "%s: %s\n", msg, strerror(errno));
  pthread_exit(NULL);    // exit(0);을 pthread_exit(NULL);로 대체.
}

void posix_error(int code, char *msg) /* Posix-style error */
{
  fprintf(stderr, "%s: %s\n", msg, strerror(code));
  pthread_exit(NULL);
}

// 중략

void dns_error(char *msg) /* Obsolete gethostbyname error */
{
  fprintf(stderr, "%s\n", msg);
  pthread_exit(NULL);
}
```

`main` 함수에서는 pipeline error `SIGPIPE` signal을 무시하도록 설정해야 한다. 또한, connection acception이 발생할 때마다 새로운 thread를 만들어서 처리해야 한다.

Main thread는 절대로 죽어버리면 안된다. Acception 부분에서는 wrapper 함수를 사용하지 않고 직접 error handling을 하였다.

```
int main(int argc, char *argv[])
{
  // 전략

  /* Ignore SIGPIPE signal. */
  Signal(SIGPIPE, SIG_IGN);

  // 중략

  /* Accept incoming connections. */

  int failed_fd = -1;

  while (1) {
    /* Try to resolve failed fd. */
    if (0 <= failed_fd) {
      if (!pthread_create(&tid, NULL, service_thread, (void*)(intptr_t)failed_fd)) {
        /* It succeeded. */
        fprintf(stderr, "Failed fd %d is resolved\n", failed_fd);
        failed_fd = -1;
      } else {
        fprintf(stderr, "Failed to resolve fd %d\n", failed_fd);
        continue;
      }
    }

    /* Accept connection. */
```

```
    client_addrlen = sizeof(client_addr);
    const int conn_fd = accept(listen_fd, (SA*)(&client_addr), &client_addrlen);

    if (conn_fd < 0) {
      fprintf(stderr, "Failed to accept incoming connection\n");
      continue;
    }

    /* Do proxy service. */
    if (pthread_create(&tid, NULL, service_thread, (void*)(intptr_t)conn_fd)) {
      /* It failed. */
      fprintf(stderr, "Failed to create a new service thread\n");
      failed_fd = conn_fd;
    }
  }

  return EXIT_SUCCESS;
}
```

`service_thread`는 인자로 받은 connection fd를 바탕으로 기존 part 1과 유사한 과정을 수행한다. Main thread가 명시적으로 reap하지 않아도 memory leak가 발생하지 않도록, 스스로를 detach해야 한다.

```
/*
 * service_thread - Thread function to service.
 */
void* service_thread(void *vargp)
{
  /* Get the connection fd. */
  const int conn_fd = (int)(intptr_t)vargp;

  if (conn_fd < 0) {
    fprintf(stderr, "Invalid connection fd\n");
    return NULL;
  }

  /* Detach thread. */
  if (pthread_detach(pthread_self())) {
    fprintf(stderr, "Failed to detach\n");
    return NULL;
  }

  /* Do proxy service. */
  run_service(conn_fd);

  if (close(conn_fd) < 0) {
    fprintf(stderr, "Failed to close connection fd\n");
    return NULL;
  }

  return NULL;
```

```
}
```

Part 3

단순히 캐싱해서 그대로 전달하면 되는 쉬운 part라고 생각했지만, 큰 판단 실수였다. Cache linked list부터 semaphores까지 그 구조를 고려하면 Malloc lab 급의 방대한 구현이 요구된다.

Global Cache

Semaphores

먼저, 여러 개의 threads가 동시다발적으로 하나뿐인 global cache에 접근할 수 있어야 한다. 이미 있는 cache를 가져와서 server와의 expensive connection overhead를 아끼는 것은, 이미 끝난 connection에 대한 caching을 진행하는 것보다 더 중요하다고 판단하였다. 따라서, 이는 3rd readers-writers problem으로 모델링할 수 있다. 수업에서 배운 fast solution을 적용하였다.

```
/* Semaphores for accessing the cache. */
struct {
  sem_t sem_in, sem_out, sem_w;
  int incnt, outcnt, wait;
} cache_sems;

/*
 * access_read_cache - Wait until
 *    being able to read the cache
 *    and do __func.
 */
#define access_read_cache(__func)  {\
    P(&(cache_sems.sem_in));\
    cache_sems.incnt += 1;\
    V(&(cache_sems.sem_in));\
    { (__func); }\
    P(&(cache_sems.sem_out));\
    cache_sems.outcnt += 1;\
    if (1 == cache_sems.wait\
      && cache_sems.incnt == cache_sems.outcnt)\
        V(&(cache_sems.sem_w));\
    V(&(cache_sems.sem_out));\
  }

/*
 * access_write_cache - Wait until
 *    being able to write the cache
 *    and do __func.
 */
#define access_write_cache(__func)  {\
    P(&(cache_sems.sem_in));\
    P(&(cache_sems.sem_out));\
    if (cache_sems.incnt == cache_sems.outcnt)\
      V(&(cache_sems.sem_out));\
    else {\
```

```
      cache_sems.wait = 1;\
      V(&(cache_sems.sem_out));\
      P(&(cache_sems.sem_w));\
      cache_sems.wait = 0;\
    }\
    { (__func); }\
    V(&(cache_sems.sem_in));\
  }
```

Semaphores를 다루다가 발생한 error를 resolve하는 방법을 모르겠다. Fatal disaster로 간주하고, wrapper 함수 `P`, `V`를 사용하였다.

예를 들어, global cache를 read해야 한다면, 다음과 같이 편하게 구현할 수 있다.

```
/*
 * get_response_cache - Get the duplicated
 *    response cache corresponding
 *    to the given request line.
 */
response_cache* get_response_cache(const request_line* const req_line)
{
  if (!req_line)
    return NULL;

  response_cache *ret;
  access_read_cache((ret = __get_response_cache(req_line)));
  return ret;
}
```

Cache entry

Cache 결과를 적절하게 활용하기 위해서는, client의 request 정보(hostname, path, port)와 server의 response 정보(headers, target object)를 모두 기록해두어야 한다.

먼저, client의 request 정보를 관리하자. 적합한 cache data가 있는지 빠르게 찾기 위해서는 이 정보의 comparison이 충분히 가벼워야 한다. String comparison이 bottleneck이며, 여기서는 hash를 통해 이를 해결하였다.

```
/* Hash values of request_line. */
typedef struct request_line_hash {
  uint32_t hostname;
  uint32_t path;
} request_line_hash;

/*
 * RFC 1945 Request-Line.
 * Method SP Request-URI SP HTTP-Version CRLF
 */
typedef struct request_line {
```

```
  /* Method must be "GET". */
  /* Request-URI will be parsed as below. */
  /* HTTP version will be considered as "HTTP/1.0". */

  /* Parsing data of the request-uri. */
  char hostname[MAXLINE];
  char path[MAXLINE];
  int port;

  /* Hashes of hostname and path. */
  request_line_hash hash_info;
} request_line;
```

Hash는 충분히 가벼우면서도 collision 확률이 충분히 낮은 Fowler-Noll-Vo의 것을 채택하였다.

```
/*
 * hash - Fowler-Noll-Vo hash.
 */
uint32_t hash(const void* const data, int len)
{
  if (!data || len < 1)
    return 0;

  const uint8_t *p = (const uint8_t*)data;
  uint32_t r = 0x811c9dc5;
  while (0 < len) {
    r ^= *p;
    r *= 0x1000193;
    p++;
    len--;
  }
  return r;
}
```

이러면, 두 `request_line` 간의 일치 여부를 매우 가볍게 알 수 있다. 일치하지 않는 경우의 expected time complexity가 $\mathcal{O}(1)$임에 유의하라.

```
/*
 * request_line_eq - Return whether
 *    the two given request lines are
 *    equivalent or not.
 */
int request_line_eq(const request_line* const a, const request_line* const b)
{
  if (a == b)
    return 1;
  if (!a || !b)
    return 0;
```

```
  if (a->hash_info.hostname != b->hash_info.hostname)
    return 0;
  if(a->hash_info.path != b->hash_info.path)
    return 0;
  if(a->port != b->port)
    return 0;

  if (strcmp(a->hostname, b->hostname))
    return 0;
  if (strcmp(a->path, b->path))
    return 0;

  return 1;
}
```

Server의 response 결과는 headers과 object를 구분해서 가지고 있어야 한다. 이는 specification에 명시된 cache object size 제한을 고려해야 하기 때문이다. Response line에 대한 추가적인 처리를 하고 싶지 않았기에, header를 field name과 value로 구분하지 않았다.

```
/* Response line and headers. */
typedef struct response_header {
  char data[MAXLINE];

  struct response_header *nxt;
} response_header;

/* Block data of an object. */
typedef struct object_block {
  char data[MAXLINE];
  ssize_t len;

  struct object_block *nxt;
} object_block;

/* Cache for response. */
typedef struct response_cache {
  /* Response headers. */
  response_header *hdr;

  /* Object blocks; may be nullptr. */
  object_block *obj;

  /* Total allocated bytes. */
  int totsz;
} response_cache;
```

Global cache는 request line에 일치하는 cache entry를 빠르게 찾을 수 있어야 하고, LRU eviction policy에 따라 메모리 사용량을 관리할 수 있어야 한다. Hash 덕분에 comparison이 충분히 가벼워졌으므로, LRU policy을 구현하기 쉬운 circular linked list 구조를 채택하였다. 사용된 시점이 최근과 가장 가까운 순서대로 linked list에 순방향으로 관리될 것이다. 이러면 LRU entry는 `root->prv`로 접근할 수 있게 된다.

```
/* Cache entry structure. */
typedef struct cache_entry {
  /* Request line. */
  request_line *req_line;

  /* Response cache data. */
  response_cache *cache;

  /* Circular linked-list pointers. */
  struct cache_entry *prv, *nxt;
} cache_entry;

/* Global cache. */
struct {
  cache_entry *root;

  int left_bytes;
} cache;
```

Linked list의 global cache에서 target entry를 찾고 삽입하고 삭제하는 작업의 구현은 기계적이다. Circular하기 때문에 일부 경우에서 singleton 혹은 empty set 처리를 해야 한다. 흥미로운 sub-function은 part section 뒤에서 소개한다.

Respond to the Client

Client의 request를 받았다면, server와 connect하기 전에 먼저 cached data가 있는지 확인해야 한다. Cached data는 global cache의 `get_response_cache` 함수로 받아올 수 있고, `respond_with_cache`는 이를 바탕으로 server와 전혀 통신하지 않고 client에게 적절한 responce를 제공한다.

```
/*
 * service_response - Service to the client.
 *    req_hdrs_root may be nullptr.
 *    The list of req_hdrs_root will be freed.
 */
int service_response(request_line* const req_line, request_header* req_hdrs_root, rio_t*
const client_rio, const int conn_fd)
{
  if (!req_line || !client_rio || conn_fd < 0)
    return -1;

  /* Get cache data if exists. */
  response_cache* const cache = get_response_cache(req_line);
  if (cache) {
```

```
    /* Do service with the cache. */
    const int retval = respond_with_cache(cache, conn_fd);

    free_response_cache(cache);
    free(cache);

    return retval;
  }

  // 후략
}
```

만약, cached data가 없다면, 기존과 같이 server의 response를 client에게 그대로 전달하는 과정을 거쳐야 한다. 그리고, 이 과정을 기록하여, 가능한 경우에는 global cache에 기록해야 한다. 이 일련의 과정은 `respond_naive` 함수가 처리한다.

Cached case

구현은 아주 간결하다. 기록된 response headers를 순차적으로 전송한 후, object를 각 block 단위로 전송하면 충분하다.

```
/*
 * respond_with_cache - Respond to the client
 *    by using the information of the given cache.
 */
int respond_with_cache(const response_cache* const cache, const int conn_fd)
{
  if (!cache || conn_fd < 0)
    return -1;

  /* Send the headers. */
  const response_header *hdr = cache->hdr;
  while (hdr) {
    print_rio(conn_fd, hdr->data);
    hdr = hdr->nxt;
  }
  print_rio(conn_fd, CRLF);

  /* Send the object. */
  const object_block *obj = cache->obj;
  while (obj) {
    Rio_writen(conn_fd, (void*)(obj->data), obj->len);
    obj = obj->nxt;
  }

  return 0;
}
```

Uncached case

기존의 전송 과정을 즉시 caching 할 수 있도록 적절히 기록해두어야 한다. Responded object의 크기가 너무 커지게 되면 더이상 기록하지 않도록 적절히 메모리 관리를 해야한다.

```
/*
 * respond_naive - Respond to the client
 *    by just serving the responce from the server
 *    and caching it if possible.
 */
int respond_naive(rio_t* const server_rio, request_line* const req_line, const int
conn_fd)
{
  if (!server_rio || !req_line || conn_fd < 0)
    return -1;

  response_cache *cache = malloc(sizeof(response_cache));

  if (cache) {
    cache->hdr = NULL;
    cache->obj = NULL;
    cache->totsz = sizeof(response_cache);
  }

  char line[MAXLINE];
  ssize_t len;

  /* Read response line and headers. */

  response_header *hdr_p = NULL;

  while (0 < (len = Rio_readlineb(server_rio, line, MAXLINE))) {
    if (len < 2 || strcmp(line + len - 2, CRLF)) {
      fprintf(stderr, "Response header is too long or not ended with CRLF\n");
      return -1;
    }

    /* Just concat to the client. */
    Rio_writen(conn_fd, line, len);

    if (2 == len)
      break;

    if (!cache)
      continue;

    response_header* const hdr = malloc(sizeof(response_header));

    if (!hdr) {
      free_response_cache(cache);
      free(cache);
      cache = NULL;
      continue;
```

```
    }

    strcpy(hdr->data, line);
    hdr->nxt = NULL;

    if (cache->hdr)
      hdr_p->nxt = hdr;
    else
      cache->hdr = hdr;

    hdr_p = hdr;

    cache->totsz += sizeof(response_header);
  }

  /* Read the response object. */

  object_block *obj_p = NULL;
  ssize_t obj_totlen = 0;

  while (0 < (len = Rio_readnb(server_rio, line, MAXLINE))) {
    /* Just concat to the client. */
    Rio_writen(conn_fd, line, len);

    if (!cache)
      continue;

    /* Update total length of the object. */
    obj_totlen += len;

    if (MAX_OBJECT_SIZE < obj_totlen) {
      free_response_cache(cache);
      free(cache);
      cache = NULL;
      continue;
    }

    object_block* const obj = malloc(sizeof(object_block));

    if (!obj) {
      free_response_cache(cache);
      free(cache);
      cache = NULL;
      continue;
    }

    memcpy(obj->data, line, len * sizeof(char));
    obj->len = len;
    obj->nxt = NULL;

    if (cache->obj)
      obj_p->nxt = obj;
    else
```

```
      cache->obj = obj;

    obj_p = obj;

    cache->totsz += sizeof(object_block);
  }

  /* If possible to cache, then should not free the cache. */
  if (cache && !push_response_cache(req_line, cache))
    return 0;

  /* Free the (useless) cache. */
  if (cache) {
    free_response_cache(cache);
    free(cache);
    cache = NULL;
  }

  return 0;
}
```

Client에게 충실하게 respond한 이후에 global cache에 cache entry를 기록할 수 있도록 semaphore wait하는 것이 가장 이상적이다. 함수의 구조상 `push_response_cache` 전에 connection fd의 close가 이루어지지 않았다. Clever한 client라면 response header의 object size를 바탕으로, 충분한 data receiving이 이루어지면 알아서 close할 것이라고 예상한다.

Miscellaneous

구현한 모든 전처리문, 구조체, 함수의 prototypes은 이러하다. 작동 과정의 흐름을 이해하기에 충분할 것이다.

```
/* Compute the size of a cache entry of res_cache. */
#define GET_CACHE_ALLOC_SIZE(res_cache)

/*
 * free_list - Free the entire list
 *    starts with __argp.
 *    Each list entry must have a pointer nxt.
 */
#define free_list(__argp)

/*
 * copy_list - Copy the entire list
 *    starts with __argp
 *    and save them in __retp.
 *    The list must be linear (not circular).
 *    If something goes wrong,
 *    then __retp will be nullptr.
 */
#define copy_list(__retp, __argp)
```

```
#define access_read_cache(__func)
#define access_write_cache(__func)
```

```
/* Hash values of request_line. */
typedef struct request_line_hash;

/*
 * RFC 1945 Request-Line.
 * Method SP Request-URI SP HTTP-Version CRLF
 */
typedef struct request_line;

/*
 * RFC 1945 Request-Header.
 * field-name ":" SP field-value CRLF
 */
typedef struct request_header;

/* Response line and headers. */
typedef struct response_header;

/* Block data of an object. */
typedef struct object_block;

/* Cache for response. */
typedef struct response_cache;

/* Cache entry structure. */
typedef struct cache_entry;
```

```
/* Cache-related functions. */
void init_cache();
void __reroot_cache(cache_entry* const p);
response_cache* __get_response_cache(const request_line* const req_line);
response_cache* get_response_cache(const request_line* const req_line);
void __erase_cache_entry(cache_entry* const entry);
int __push_response_cache(const request_line* const req_line, const response_cache* const
res_cache);
int push_response_cache(const request_line* const orig_req_line, const response_cache*
const cache);

/* Service functions. */
void* service_thread(void *vargp);
void run_service(const int conn_fd);
int service_response(request_line* const req_line, request_header* req_hdrs_root, rio_t*
const client_rio, const int conn_fd);
int respond_naive(rio_t* const server_rio, request_line* const req_line, const int
conn_fd);
int respond_with_cache(const response_cache* const cache, const int conn_fd);

/* Service-related helper functions. */
```

```
void push_request_header(request_header** const root, const char* const name, const char*
const value);
int parse_request_line(request_line* const req_line, rio_t* const client_rio);
int parse_request_headers(request_header **root, const request_line* const req_line,
rio_t* const client_rio);

/* Memory-related functions. */
void free_response_cache(response_cache* const p);
request_line* copy_request_line(const request_line* const p);
response_cache* copy_response_cache(const response_cache* const p);

/* Basic rio functions. */
void print_rio(const int fd, const char* const str);
void print_rio_header(const int fd, const char* const name, const char* const value);

/* Hash functions. */
uint32_t hash(const void* const data, int len);
uint32_t hash_str(const char* const str);
void hash_request_line(request_line* const req_line);

/* Equality functions. */
int strprefixeq(const char* const target, const char* const prefix);
int request_line_eq(const request_line* const a, const request_line* const b);
```

__get_response_cache

이 함수는 cache read 권한을 받은 이후에 실행되며, 인자로 주어진 request line와 일치하는 cache entry를 찾아 반환해야 한다. Cache entry를 찾은 후에는 최대한 빠르게 readable semaphore를 반환하는 것이 이상적이기 때문에, memory safe한 구현을 위하여 찾은 cache entry를 deep copy하여 반환하도록 하였다. Deep copy 과정은 `copy_response_cache` 함수가 수행한다.

LRU policy를 고려하여, 찾은 cache entry의 access time을 update해야 한다. 이는 `__reroot_cache` 함수에서 처리한다.

```
/*
 * __get_response_cache - Get the duplicated
 *    response cache corresponding
 *    to the given request line.
 *    req_line must be not nullptr.
 */
response_cache* __get_response_cache(const request_line* const req_line)
{
  /* The global cache is empty. */
  if (!(cache.root))
    return NULL;

  /* Traversal the global cache. */

  cache_entry *p = cache.root;

  do {
```

```
    /* Check the request line is the same. */
    if (request_line_eq(req_line, p->req_line)) {
      /* LRU eviction policy. */
      __reroot_cache(p);
      return copy_response_cache(p->cache);
    }

    p = p->nxt;
  } while (p && p != cache.root);

  return NULL;
}
```

parse_request_line

이 함수는 client의 첫 request line에서 hostname과 relative path, port를 parsing 해야 한다. 이 구현 과정은 매우 기계적이지만, 이 서버 코드에서 핵심적인 함수 중 하나이므로, 간략하게 소개한다.

먼저, client로부터 한 줄을 읽은 후, CRLF로 종료되는 올바른 줄인지 검사한다.

```
/*
 * parse_request_line - Parse the request line.
 */
int parse_request_line(request_line* const req_line, rio_t* const client_rio)
{
  if (!req_line || !client_rio)
    return -1;

  char line[MAXLINE];
  ssize_t len = Rio_readlineb(client_rio, line, MAXLINE);

  /* Read the very first request line. */
  if (len < 2 || strcmp(line + len - 2, CRLF)) {
    fprintf(stderr, "Request is too long or not ended with CRLF\n");
    return -1;
  }

  /* Strip the line. */
  line[len-2] = line[len-1] = '\0';
  len -= 2;

  // 후략
}
```

HTTP 1.0에는 `PUT` 등의 여러 methods가 있지만, 이 과제에서는 오직 `GET`만 고려하면 충분하다. Request line은 method, request-uri, http version이 공백으로 구분되어 주어지는데, http version은 무시해도 충분하다. Server에게 request는 항상 `HTTP/1.0`으로 해야 하고, client request의 http version이 valid한지는 검사하지 않을 것이기 때문이다.

```
  /* Check the method is "GET". */
  if (!strprefixeq(line, "GET ")) {
    fprintf(stderr, "Unsupported method\n");
    return -1;
  }

  /* Parse the request-uri. */
  char req_uri[MAXLINE];
  sscanf(line + 4, " %s", req_uri);
```

이제 이 함수에서 가장 까다로운, request-uri parsing을 해야 한다. 항상 `http://`로 시작하는 well-formed absolute path가 주어진다고 가정하였다.

```
  /* Parse the request-uri. */
  char req_uri[MAXLINE];
  sscanf(line + 4, " %s", req_uri);

  /*
   * HTTP URL
   * "http:" "//" host [ ":" port ] [ absolute_path ]
   */

  /* Assume the absolute path. */
  if (!strprefixeq(req_uri, "http://")) {
    fprintf(stderr, "Request URI must be a valid http URL\n");
    return -1;
  }

  /* Parse the host. */

  char* const host_sp = req_uri + 7;

  char *bufp = host_sp;
  char *hostp = req_line->hostname;

  while (1) {
    const char c = *bufp;
    if ('\0' == c || ':' == c || '/' == c)
      break;

    *hostp = c;
    hostp++;
    bufp++;
  }

  *hostp = '\0';

  /* Check the hostname is non-empty. */
  if ('\0' == *(req_line->hostname)) {
    fprintf(stderr, "Empty hostname\n");
```

```
    return -1;
  }
```

Colon 뒤에 오는 수는 specified port로 해석하고, 남은 부분을 relative path로 parse한다.

```
int parse_request_line(request_line* const req_line, rio_t* const client_rio)
{
  // 전략

  /* Set the default port as 80. */
  req_line->port = 80;

  /* Port is specified. */
  if (':' == *bufp) {
    int p = -1;

    bufp++;
    while (p <= MAX_PORT) {
      const char c = *bufp;
      if (c < '0' || '9' < c)
        break;

      if (p < 0)
        p = 0;

      p = p*10 + (c&15);
      bufp++;
    }

    if (p < 0 || MAX_PORT < p) {
      fprintf(stderr, "Invalid port\n");
      return -1;
    }

    req_line->port = p;
  }

  /* The remaining part is the path. */
  strcpy(req_line->path, bufp);

  /* Check the path starts with '/'. */

  const char path_sc = *(req_line->path);

  if (path_sc) {
    if ('/' != path_sc) {
      fprintf(stderr, "Absolute path must start with '/'\n");
      return -1;
    }
  } else {
    strcpy(req_line->path, "/");
```

```
  }

  return 0;
}
```

parse_request_header

이 함수는 client의 request headers를 하나씩 읽어들여, field name과 value를 parsing한다. 이 과정에서 몇 가지 specification이 지켜져야 한다.

- `Host` field는 명시되면 그 값을, 그렇지 않다면 hostname을 value로 해야 한다.
- `User-Agent` field는 항상 `HDR_DATA_USR_AGT`을 값으로 하여 보내야 한다.
- `Connection`과 `Proxy-Connection` fields는 `close`를 값으로 하여 보내야 한다.

Field name은 case-insensitive 함에 유의해야 한다.

```
/*
 * parse_request_headers - Parse each request header.
 */
int parse_request_headers(request_header **root, const request_line* const req_line,
rio_t* const client_rio)
{
  if (!root || !req_line || !client_rio)
    return -1;

  *root = NULL;

  /* Name and value of the header field. */
  char name[MAXLINE], value[MAXLINE];

  char line[MAXLINE];
  ssize_t len;

  /* Is the host field given? */
  int host_flag = 0;

  while (0 < (len = Rio_readlineb(client_rio, line, MAXLINE))) {
    if (!strcmp(line, CRLF))
      break;

    if (len < 2 || strcmp(line + len - 2, CRLF)) {
      fprintf(stderr, "Request header is too long or not ended with CRLF\n");
      return -1;
    }

    /* Strip the line. */
    line[len-2] = line[len-1] = '\0';
    len -= 2;

    char* const mid_p = strstr(line, CLNSP);
```

```
    if (!mid_p) {
      fprintf(stderr, "Invalid request header\n");
      return -1;
    }

    /* Parse name and value. */
    *mid_p = '\0';
    strcpy(name, line);
    strcpy(value, mid_p + 2);

    if ('\0' == name[0]) {
      fprintf(stderr, "Field name of the request header must be non-empty\n");
      return -1;
    }

    if (!strcasecmp(name, HDR_USR_AGT)
     || !strcasecmp(name, HDR_CONN)
     || !strcasecmp(name, HDR_PRX_CONN))
        continue;

    if (!strcasecmp(name, HDR_HOST))
      host_flag = 1;

    push_request_header(root, name, value);
  }

  if (!host_flag)
    push_request_header(root, HDR_HOST, req_line->hostname);

  return 0;
}
```

어려웠던 점

Global cache를 구현하기 위하여 막대한 기계적인 구현을 해야 한다는 점이 상당히 고통스러웠다. 특히 linked list 관련 기능을 구현할 때가 기계적인 노동이었다. Kernel lab 때 사용하였던 이미 구현된 `list_head` 구조체와 함수, macro를 활용할 수 있었을 법 했지만, 이들의 명확한 specification을 얻을 수 없어, 디버깅을 하지 않기 위해 직접 구현했다.

RFC 1945의 명세를 읽고 이해하는 과정은 그 어떤 사용설명서를 읽는 것보다 어려웠다. 특히, 이번 과제는 RFC 1945의 명세를 모두 지키는 HTTP/1.0 protocol만을 고려하면 충분했지만, 실제 많은 사이트들은 이 명세를 엄격하게 지키지 않고, 실사용되는 브라우저들은 관용에 맞추어 적절히 accept한다. 이번 과제를 구현하면서 어느 정도까지 내가 수용해주어야 할지 고민이 많았다. 수용의 폭이 넓어질 수록 구현량이 상당히 많아졌기 때문에, 최대한 엄격하게 구현했다.

Semaphores를 직접 다루는 일은 매우 흥미로웠으나, 내가 이를 안전하게 구현하였는지는 확인하는 방법은 다양한 케이스를 여러 번 수행해보는 실험적인 방법밖에 없었기에, 이 구현에 대한 강한 확신이 들지는 않는다.

새롭게 배운 점

Semaphores를 활용하여 concurrent access problem을 고민하고 해결하는 과정이 상당히 재밌었고 인상 깊었다. 하나의 큰 프로그램이 어떻게 작동되고, 그 과정 속에서 지켜야 하는 철학, service 우선순위 등을 모두 고려해야만 올바르고 바람직한 concurrent model을 만들 수 있다. 3rd readers-writers problem으로 모델링한 후, 이것이 proper하게 작동하게 하기 위해서 자료 구조를 설계하고 hashing을 적용하는 과정이 가장 즐거웠다.

생각보다 HTTP/1.0 protocol은 매우 단순해서 놀라웠다. 실제 브라우저에 구현한 proxy server를 적용하였더니 http 사이트에 잘 접속되는 것이 인상 깊었다. 브라우저는 다양한 통신 protocol과 함께 이 결과를 시각화하여 보여주는 것까지 수행하여야 한다. 브라우저를 만드는 작업이 생각보다 아주 거대한 프로젝트임을 깨달았다.