report.md

Proxy Lab

2014-17831 김재원

Overview

- Proxy는 서버와 클라이언트가 통신할 때, 매번 직접 연결을 할 필요가 없도록 중간다리 역할을 해주는 존재이다. 우리가 프록시를 사용하는 가장 큰 이유 중 하나는 proxy가 중간에서 caching을 해서 성능을 올려주기 때문이다. 매번 client가 request를 할 때마다 서버에서부터 직접 받아오면 시간도 오래 걸리고, 특히 웹의 경우 동일한 오브젝트를 여러 번 받아오는 경우가 흔하기 때문에 여러모로 낭비라고 볼 수 있다. 따라서 서버에게는 클라이언트 역할을, 클라이언트에게는 서버 역할을 하는 프록시가, 한 번 받은 object를 캐싱하면 이후에 동일한 요청을 받았을 때 서버까지 직접 가지 않고 프록시 내부에 가지고 있는 cache에서 오브젝트를 꺼내 전달해주어서 통신 시간을 획기적으로 줄여줄 수 있다.
- 이번 과제는 총 3개 파트로 구성되어 있는데, 각 파트의 목표는 다음과 같다:
  - Part 1: 프록시의 기본 기능을 수행할 수 있도록 구현하기. 단, sequential한 서버로써 기능하도록.
  - Part 2: thread를 활용해서 concurrent하게 여러 client의 request를 한꺼번에 처리할 수 있도록 구현하기.
  - Part 3: 캐시를 구현하여 기존 요청이 다시 들어올 경우 서버를 거치지 않고 바로 반환하도록 구현하기.

Implementation

Macros

```
#include <stdio.h>
#include "csapp.h"
#include "sbuf.h"

/* Recommended max cache and object sizes */
#define MAX_CACHE_SIZE 1049000
#define MAX_OBJECT_SIZE 102400
#define NTHREADS 4
#define SBUFSIZE 16
```

- `sbuf.h` 와 `sbuf.c` , 그리고 `NTHREADS` 나 `SBUFSIZE` 와 같은 것들은 교과서의 내용을 활용하였다.

Cache

```
typedef struct CachedItem CachedItem;

struct CachedItem {
    char cache_object[MAX_OBJECT_SIZE];
    char uri[MAXLINE];
    clock_t access_time;
    size_t size;
    struct CachedItem *prev;
    struct CachedItem *next;
};

typedef struct {
    struct CachedItem *head;
    struct CachedItem *tail;
} CacheList;

CacheList list;
size_t cache_size = 0;
```

- object 데이터와 URI, 그리고 LRU를 위한 `access_time` , object 크기, 그리고 Cached Item list 상에서의 prev와 next item을 가리키는 포인터로 구성된 struct를 정의하였다.
  - next는 linked list 구현 대부분을 위해 필요하고, prev의 경우 create를 할 시에 tail에 추가를 해주기 위해, 그리고 evict를 할 시에 evict할 item을 찾은 후 쉽게 deletion을 할 수 있도록 돕기 위해 넣었다.

- list의 맨 앞과 맨 뒤를 가지고 있는 CachedList 구조체도 선언하였다. 맨 앞은 traversal을 위해 필요하고 tail의 경우 insertion 용도로 사용하였다. 사실 cache item을 액세스 할 때 access time을 변경해준 후, 재정렬을 미리 해주지 않기 때문에 tail에 추가하는 것이 큰 의미는 없지만, 구현 당시에는 재정렬 가능성을 염두에 두고 있었기 때문에 이렇게 구현을 했다.
- cache의 최대 사이즈가 정해져있기 때문에, 남아있는 캐시 공간이 있는지 쉽게 판단하고자 cache_size 변수를 만들었다.

Mutex

```
int read_cnt;

sbuf_t sbuf;    /* Shared buffer of connected descriptors */
sem_t w_mutex, r_mutex, u_mutex;
```

- Concurrent programming을 위해 필요한 mutex들이다. Part 2에서는 사실 따로 필요하지 않지만, Part 3에서 cache를 access할 시 write의 독립성이 보장되어야 해서 mutex가 필요했다.

Constants

```
/* You won't lose style points for including this long line in your code */
static const char *user_agent_hdr = "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:10.0.3) Gecko/20120305 Firefox/10.(
static const char *conn_hdr = "Connection: close\r\n";
static const char *proxy_conn_hdr = "Proxy-Connection: close\r\n";
static const char *request_header = "GET %s HTTP/1.0\r\n";
static const char headers[3][30] = {
    "Connection",
    "Proxy-Connection",
    "User-Agent"
};
```

- header를 parsing해서 만들어줄 때 string이 너무 많아 복잡해져서 미리 constant로 선언하였다.

`main()`

```
/*
 * Part 1: Figure 11.29 The TINY Web server (`code/netp/tiny/tiny.c`)
 * Part 2: Figure 12.28 A prethreaded concurent echo server (`code/conc/echoservert-pre.c`)
 */
int main(int argc, char *argv[])
{
    int i, listenfd, connfd;
    char hostname[MAXLINE], port[MAXLINE];
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    pthread_t tid;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(1);
    }

    listenfd = Open_listenfd(argv[1]);

    sbuf_init(&sbuf, SBUFSIZE);
    list.head = malloc(sizeof(CachedItem));
    cache_init();

    for (i = 0; i < NTHREADS; i++) {
        Pthread_create(&tid, NULL, thread, NULL);   /* Create worker threads */
    }

    while(1) {
        clientlen = sizeof(clientaddr);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);

        Getnameinfo((SA *) &clientaddr, clientlen, hostname, MAXLINE, port, MAXLINE, 0);
        printf("Accepted connection from (%s %s)\n", hostname, port);

        sbuf_insert(&sbuf, connfd); /* Insert connfd in buffer */
```

```
    }
    return 0;
}
```

- 주석에 언급한 바와 같이 Part 1은 Figure 11.29 The TINY Web server (`code/netp/tiny/tiny.c`)을 참고하였고, Part 2는 Figure 12.28 A prethreaded concurent echo server (`code/conc/echoservert-pre.c`)을 참고하였다.
- 이 함수의 경우 책에서 참고한 내용에서 변경할 사항이 거의 없었다.

thread()

```
/*
 * Part 2: Figure 12.28 A prethreaded concurent echo server (`code/conc/echoservert-pre.c`)
 */
void *thread(void *vargp) {
    Pthread_detach(pthread_self());
    while (1) {
        int connfd = sbuf_remove(&sbuf);    /* Remove connfd from buffer */
        doit(connfd);                   /* Service client */
        Close(connfd);
    }
}
```

- 주석의 내용과 같이 교과서의 Figure 12.28 A prethreaded concurent echo server (`code/conc/echoservert-pre.c`)를 참고해서 작성하였다.
- 기본적인 병렬프로그래밍 동작이라 변경할 사항은 따로 없었다.

doit()

```
/*
 * Part 1: Figure 11.30 TINY `doit` handles one HTTP transaction (`code/netp/tiny/tiny.c`)
 */
void doit(int fd)
{
    int clientfd;
    char buf[MAXLINE], method[MAXLINE], uri[MAXLINE], version[MAXLINE];
    char hostname[MAXLINE], path[MAXLINE], port[MAXLINE];
    char http_header[MAXLINE];
    rio_t client_rio, server_rio;

    Rio_readinitb(&client_rio, fd);
    Rio_readlineb(&client_rio, buf, MAXLINE);
    sscanf(buf, "%s %s %s", method, uri, version);

    // GET request만 handle하는 프록시이므로 GET이 아닐 경우 즉시 리턴한다.
    if (strcasecmp(method, "GET")) {
        printf("Proxy does not implement the method");
        return;
    }

    // cache에 uri에 해당하는 정보가 있으면 fd가 가리키는 곳에 적어준 후 즉시 리턴한다.
    if (check_cache(uri, fd)) { /* Part 3 */
        return;
    }

    // uri를 파싱해서 정보를 읽어온다.
    parse_uri(uri, hostname, path, port);

    // header 정보를 파싱한 후 적절한 헤더를 만들어서 출력한다.
    parse_header(http_header, hostname, path, &client_rio);

    // 해당하는 client와의 통신을 시작한다
    clientfd = Open_clientfd(hostname, port);

    if(clientfd < 0){
        printf("connection failed\n");
        return;
    }

    Rio_readinitb(&server_rio, clientfd);
```

```
    Rio_writen(clientfd, http_header, strlen(http_header));

    char obj_buf[MAX_OBJECT_SIZE];  // cache의 object를 만들기 위한 buffer이다.
    size_t n, obj_buf_size = 0;

    // 서버로부터 정보를 받아와서 buf에 저장한다. 받아온 정보의 크기는 n에 저장해준다.
    while ((n = Rio_readlineb(&server_rio, buf, MAXLINE)) != 0) {
        Rio_writen(fd, buf, n);     // 읽어온 내용을 connfd가 가리키는 곳에 적어준다.
        obj_buf_size += n;          // 읽어온 사이즈 만큼을 증가한다.
        if (obj_buf_size < MAX_OBJECT_SIZE) {       // cache를 만들기 위한 object의 크기가 충분히 작을 경우
            strncat(obj_buf, buf, strlen(buf));     // 읽어온 정보를 cache object에 적어준다.
        }
    }

    Close(clientfd);

    if (obj_buf_size < MAX_OBJECT_SIZE) {           // cache를 만들기 위한 object의 크기가 충분히 작을 경우
        cache_URL(uri, obj_buf, obj_buf_size);      // 해당 object를 캐싱한다.
    }

    return;
}
```

- 마찬가지로 교과서에서 Part 1: Figure 11.30 TINY `doit` handles one HTTP transaction (`code/netp/tiny/tiny.c`)를 참고하여 작성하였다. 그러나 앞서 설명한 함수들과 달리 Part 1, 2, 3에 걸쳐 상당히 많은 수정을 해야했다.
  - 우선 cache와 관련된 부분들은 전부 Part 3에서 추가되었다. 가령, `check_cache` 를 한 후 cache된 내역이 있으면 즉시 해당 내용을 write하고 return한다. 그렇지 않으면 서버로부터 URI 정보를 읽어오도록 요청하는데, 이 때 object 크기를 추적하면서 크기가 충분히 작은 object의 경우 캐시에 저장해준다.
- 함수가 길고 내용이 많아 구현 방식에 대한 상세한 설명은 위 코드에 주석으로 적어두었다.

check_cache()

```
int check_cache(char *uri, int fd)
{
    int flag = 0;

    P(&r_mutex);
    read_cnt++;
    if (read_cnt == 1) //first in
                P(&w_mutex);
        V(&r_mutex);

    CachedItem * cached_item = find(uri);

    if (cached_item) {
        P(&u_mutex);
        rio_writen(fd, cached_item->cache_object, cached_item->size);
        cached_item->access_time = clock();
        V(&u_mutex);
        flag = 1;
        V(&u_mutex);
    }

    P(&r_mutex);
        read_cnt--;
        if (read_cnt == 0) //last out
                V(&w_mutex);
        V(&r_mutex);

    return flag;
}
```

- 받은 request의 URI에 해당하는 정보가 캐시에 있는지를 확인하는 함수이다. 확인하는 동안에는 exclusive한 access가 필요하므로, 교과서에서 구현되었던 mutex 사용의 기본적인 코드를 활용해서 구현했다.
- 실제로 cache item을 찾는 함수는 `find()` 함수로, 이 함수에서는 해당 함수를 호출하기만 한다.
  - 호출한 결과 NULL이 아닌 값이 리턴되면, 즉 캐시에 해당되는 object가 있으면 또 다른 mutex로 block을 해준 후 cache item의 정보를 전달해주고, 해당 cache의 access time을 업데이트 해준다. LRU 원칙에 따른 캐시를 구현하기 위한 장치이다.
  - 마지막으로, cache된 아이템이 있었느지 그 여부를 리턴해준다.

parse_uri()

```
void parse_uri(char *uri, char *hostname, char *path, char *port)
{
    char *start_ptr = strstr(uri, "http://");
    start_ptr = start_ptr ? start_ptr + strlen("http://") : uri;
    char *port_ptr = strstr(start_ptr, ":");

    if (port_ptr) {
        sscanf(start_ptr, "%[^:]:%[^/]%s", hostname, port, path);
    } else {
        sscanf("80", "%s", port);
        sscanf(start_ptr, "%[^/]%s", hostname, path);
    }
}
```

- 이 함수는 http://host:port/path의 형식으로 들어있는 uri를 host, port, path 각각으로 파싱하는 역할을 한다.
- Port가 명시되어 있지 않을 경우 해당되는 포트가 80이라고 가정한다는 과제 스펙에 따라 포트 부분을 파싱한 뒤, 해당되는 내용이 NULL이면 포트에 80을 적어준다.
  - URI에서 확인해야 하는 것은 `http://` 와 `:` 의 존재 여부인데, tiny server의 예시처럼 http://가 없을 경우를 대비해 http://를 기준으로 host를 strip한다.
  - 또, URI에 `:` 가 있을 경우 포트번호가 명시되어 있다는 뜻이기 때문에, 해당 정보를 이용해 포트에 80을 저장해줘야 하는 경우와 그렇지 않은 경우를 구분한다.
  - 나머지 내용은 regex를 이용해 간단하게 파싱한다.

parse_header()

```
void parse_header(char *http_header, char *hostname, char *path, rio_t *client_rio)
{
    char buf[MAXLINE];
    char header_request[MAXLINE], header_host[MAXLINE], header_other[MAXLINE];

    sprintf(header_request, request_header, path);

    Rio_readlineb(client_rio, buf, MAXLINE);
    while (strncmp(buf, "\r\n", strlen("\r\n"))) {
        if (!strncmp(buf, "Host", strlen("Host"))) {
            strncpy(header_host, buf, strlen(buf));
            Rio_readlineb(client_rio, buf, MAXLINE);
            continue;
        }

        if (!check_header(buf)) {
            strncat(header_other, buf, strlen(buf));
        }

        Rio_readlineb(client_rio, buf, MAXLINE);
    }

    if (strlen(header_host) == 0) {
        sprintf(header_host, "Host: %s\r\n", hostname);
    }

    sprintf(http_header, "%s%s%s%s%s%s%s",
            header_request,
            header_host,
            user_agent_hdr,
            conn_hdr,
            proxy_conn_hdr,
            header_other,
            "\r\n");
}
```

- Handout 내용에 따라 구현한 함수이다. 이 함수에서는 클라이언트에서 들어오는 헤더를 통해 서버에 보낼 헤더에 필요한 내용을 선별하게 되는데, 3.2번 항목에 제시된 바를 그대로 구현하려고 했다.
  - 즉, Host가 들어있으면 헤더에 서버의 호스트 정보를 추가해주고서 `header_host` 에 저장하고, 만약 주어진 세 개 header에 해당되지 않는 헤더가 들어올 경우 `header_other` 에 따로 저장한다.

  - 이 때, 클라이언트로부터 EOF, 즉 "\r\n"이 들어오기 전까지는 계속해서 돌면서 헤더를 확인하고 필요한 헤더 정보를 내보내주어야 한다.
  - 마지막으로, Host가 들어있지 않아 `header_host` 의 크기가 0인 경우, 따로 hostname을 지정해준다.

`check_header()`

```
int check_header(char *buf) {
    char *ptr = strstr(buf, ":");
    int hdr_len = strlen(buf) - strlen(ptr);

    char *hdr = malloc(hdr_len);
    strncpy(hdr, buf, hdr_len);

    int i;
    for (i = 0; i < 3; i++) {
        if (!strncmp(hdr, headers[i], hdr_len)) {
            return 1;
        }
    }
    return 0;
}
```

- URI의 포트 번호 앞에 명시된 내용이 주어진 세 개 header 중 하나랑 일치하는지를 확인하는 함수이다. 처음에는 `strncasecmp` 함수를 사용하여 구현하였는데, 이후 언급할 디버깅 과정에서 header 파싱 과정에 문제가 있는 것 같다는 생각을 하면서 위와 같이 보수적인 방식으로 구현을 하게 되었다.

`cache_init()`

```
void cache_init()
{
    strcpy(list.head->cache_object, "");
    strcpy(list.head->uri, "");
    list.head->access_time = clock();
    list.head->size = 0;
    list.head->prev = NULL;
    list.head->next = NULL;
    Sem_init(&w_mutex, 0, 1);
    Sem_init(&r_mutex, 0, 1);
    Sem_init(&u_mutex, 0, 1);
    list.tail = list.head;
    cache_size = 0;
}
```

- Cached item들을 전부 모아둔 list, 즉 cache를 초기화하는 함수이다.
  - 단순히 `main` 함수에서 malloc 해두었던 head item에 제로값들을 설정해주면 되는데, 이 때 주의할 점은 mutex들 또한 꼭 initialize를 해주어야 한다는 점이다. 이 부분을 놓쳤다가 stack과 관련된 문제가 생겼었다.

`cache_URL`

```
void cache_URL(char *URL, char *object, size_t size)
{
    P(&w_mutex);

    if ((cache_size + size) > MAX_CACHE_SIZE) {
        while((cache_size + size) > MAX_CACHE_SIZE) {
            evict();
        }
    }

    CachedItem *new_item = malloc(sizeof(CachedItem));
    strcpy(new_item->cache_object, object);
    strcpy(new_item->uri, URL);
    new_item->access_time = clock();
    new_item->size = size;
    new_item->prev = list.tail;
    new_item->next = NULL;
    list.tail->next = new_item;
    list.tail = new_item;
```

```
    cache_size += size;

    V(&w_mutex);
}
```

- 캐싱해야 하는 request가 있을 시 해당 request를 캐시에 저장하기 위해 필요한 만큼 evict를 하고, 빈 공간에 저장을 해주는 함수이다.
  - 최대 cache size가 정해져있기 때문에 evict를 하나씩 하면서 object 전체가 저장될 수 있을 만큼의 공간을 확보한다. 이 과정은 `evict()` 함수를 통해 이루어진다.
  - 새로 만든 item은 들어온 URI와 object, 그리고 size 정보를 반영하여 저장해주고, 저장 시점의 시간으로 access_time을 지정해준다.
  - 또한, tail에 추가를 하기 때문에 prev를 tail로 지정해주고, 기존의 tail 또한 새로 만든 item을 가리키도록 해야 한다.
  - 그리고 cache_size에 새로 만들어진 cache item의 size를 꼭 추가해주어야 한다.
  - 마지막으로, 앞서 언급한 바와 같이 cache에서 evict를 하고 새로 추가를 하는 동안 cache에 변화가 일어나지 않도록 w_mutex로 꼭 블라킹을 해야 한다.

`evict()`

```
void evict()
{
    CachedItem * temp = list.head;
    temp = temp->next;
    CachedItem * evict_item = temp;
    clock_t oldest_access = temp->access_time;
    while(temp) {
        if (oldest_access > temp->access_time) {
            oldest_access = temp->access_time;
            evict_item = temp;
        }
        temp = temp->next;
    }

    if (!evict_item->next) {
        list.tail = evict_item->prev;
    }
    if (!evict_item->prev) {
        list.head = evict_item->next;
    }
    evict_item->prev->next = evict_item->next;
    evict_item->next->prev = evict_item->prev;

    free(evict_item->cache_object);
    free(evict_item->uri);
    free(evict_item);
    cache_size -= evict_item->size;
}
```

- cache에 공간이 부족할 경우 LRU policy에 따라 eviction을 해주는 함수이다.
- 우선 access_time을 기준으로 가장 적은 숫자, 즉 가장 오래 전에 access된 item을 찾아낸다.
- 그리고 기본적인 linked list의 deletion operation을 실행해준다.
  - 가령, 해당 item이 tail일 경우 해당 item의 prev를 tail로 새롭게 지정해주고, head일 경우 next를 head로 새로 지정해준다.
- 마지막으로, evict된 item의 크기만큼을 cache_size에서부터 꼭 빼주어야 한다.

`find()`

```
CachedItem *find(char *URL)
{
    CachedItem *temp = list.head;
    while (temp != NULL) {
        if (!strcmp(URL, temp->uri)) {  // found matching cache line
            return temp;
        }
        temp = temp->next;
    }
    return temp;
}
```

- request를 받은 uri에 해당하는 resource가 캐시에 저장되어 있는지를 URI를 기준으로 판단해주는 함수이다.
  - linked list traversal과 strcmp로 간단하게 구현할 수 있었다.

Concluding Remarks

- 과제 안내문을 봤을 때는 Part 2와 3가 가장 어려울 것으로 예상했는데, 실제로 구현을 해보니 그렇지 않았다. 과제에 대한 파악을 하는 데에는 Part 2와 3가 가장 오래 걸렸지만 Part2는 CSAPP 책에 나와있는 코드를 거의 그대로 사용할 수 있어서 금방 구현을 할 수 있었다. Part3 또한 뼈대 코드 예시가 주어져서 해당 내용을 참고하고 나니 (C string manipulation으로 인한 어려움을 제외하면) cache 자체와 관련된 부분은 linked list 구조와 뼈대 코드에 제시된 구조체를 이용해서 비교적 쉽게 구현할 수 있었다. 또 과제를 할 당시 매우 어렵게 느껴졌던 semaphore 관련된 내용이, 기말고사 공부를 하는 과정을 통해 훨씬 잘 이해되었고, 그런 이해를 바탕으로 비교적 쉽게 concurrent한 server을 구현할 수 있었다.
- Part1의 경우 CSAPP 교과서 내용을 대부분 참고할 수 있다는 점에서는 비교적 쉬웠지만, header이나 URI parsing과 같은 내용이 처음에는 상당히 헷갈렸다. 웹에 대한 이해가 어느 정도 필요했기 때문인 것 같다. 특히 환경부 사이트를 요청할 시 `curl: (18) transfer closed with 309759 bytes remaining to read` 라는 오류가 발생했는데, 해당 오류에 대해 검색을 하다 보니 header에 문제가 있다고 - 그래서 `Accept-encoding` header을 추가해줘야 한다는 내용을 보고 해당 header와 `Accept` header에 대해서도 구현을 했었다. 이 내용은 handout에 명시되어있지 않아 어떻게 알아내야 할지 막막하게 느껴졌었다. 결국 위 오류는 header 문제 때문이 아닌 것을 알게 되었지만, 해당 문제를 해결하는 과정에서 생소한 내용들을 다루게 돼서 좀 더 어렵게 느껴졌던 것 같다.
- Proxy lab이지만 공교롭게도 string에 대해 더 많이 배운 것 같다. C로 코딩을 많이 안 해봐서 string과 관련된 다양한 함수들을 사용해보는 기회가 되었다. 한 가지 불편했던 점은, string 간에 비교를 할 때 단순비교를 할 수 없을 뿐 아니라 `strcmp` 와 같은 함수를 사용하게 되면 리턴값이 -1, 0, 1로 나뉜다는 점이다. 일반적으로(?)는 두 문자열이 서로 동일할 때 1이 반환되고 그렇지 않을 때 0이 반환될 것 같은데, C에서는 그 반대라는 점을 잘못 기억해서 발생한 오류들을 디버깅하는 데에 한참이 걸렸다. 또 buffer을 읽어들일 때에도 null character이 붙거나 붙지 않거나 남은 공간이 복사되거나 복사되지 않는 현상들이 일어났는데, 자잘한 오류가 너무 많이 발생하다 보니 헷갈려서 초기 구현과 달리 최대한 보수적으로 - 즉 원하는 바를 지정할 수 있는 방식 (e.g. copy하고자 하는 크기를 지정)으로 - 구현을 해나가게 되었다.
- 신기했던 점은 이 과제에서 구현한 내용이 실제로 인터넷 페이지 내용을 가져올 수 있었다는 점이다. 드라이한 느낌의 코드를 작성했는데, 그것이 우리가 평소에 사용하는 인터넷 동작의 원리가 된다는 것이 신기하게 느껴졌다.