

System programming

Assignment #2-2 (proxy server)



학 과 : 컴퓨터 공학과

담당교수 : 황호영

분 반 : 목34

학 번 : 2016722092

성 명 : 정동호

제출날짜 : 2018-05-04

목차

1 INTRODUCTION.....	3
2 FLOWCHART.....	3
3 PSEUDO CODE	9
4 결과화면	16
5 결론 및 고찰	18
6 참고 레퍼런스.....	18

1 Introduction

지금까지 구현한 기능들을 활용하여 브라우저 프록시 설정을 통해 좀 더 실제와 같은 환경에서 프록시 캐싱 기능을 구현해본다. 이를 위해 클라이언트가 요청한 헤더에서 method 와 url 부분을 파싱하고 이를 해싱하여 Hit/Miss를 판별한 뒤 클라이언트에게 결과값을 전송해주도록 한다.

2 Flowchart

2-2 과제에서 변경된 순서도는 main_process, sub_process이고 추가된 순서도는 클라이언트의 요청헤더를 파싱하는 parse_request 함수이다.

또한 각 순서도의 설명은 매우 짧고 간단하게 하며 자세한 내용은 Pseudo code 부분에서 설명한다.

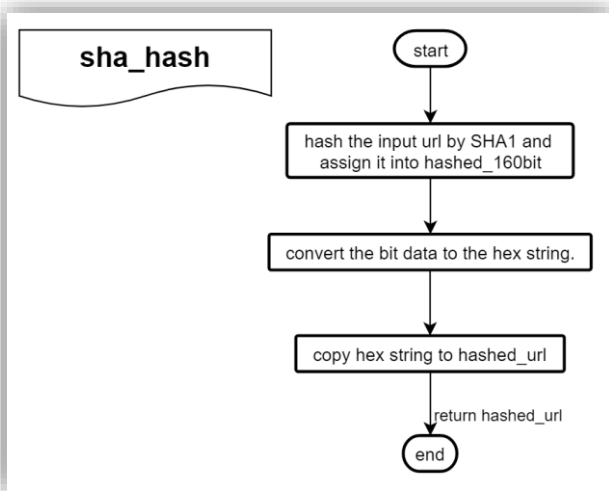


Figure 2.2 sha_hash function flowchart

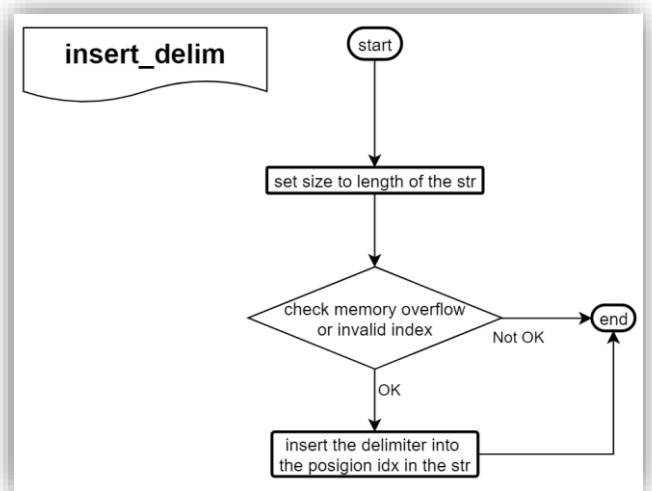


Figure 2.1 insert_delim function flowchart

Figure 2.1의 sha_hash 함수는 전달받은 문자열을 SHA1으로 해싱하여 그 값을 반환한다.

Figure 2.2의 insert_delim 함수는 인자로 전달된 문자열 포인터에 idx번째 위치에 문자 delim을 삽입하여 그 값을 반환한다.

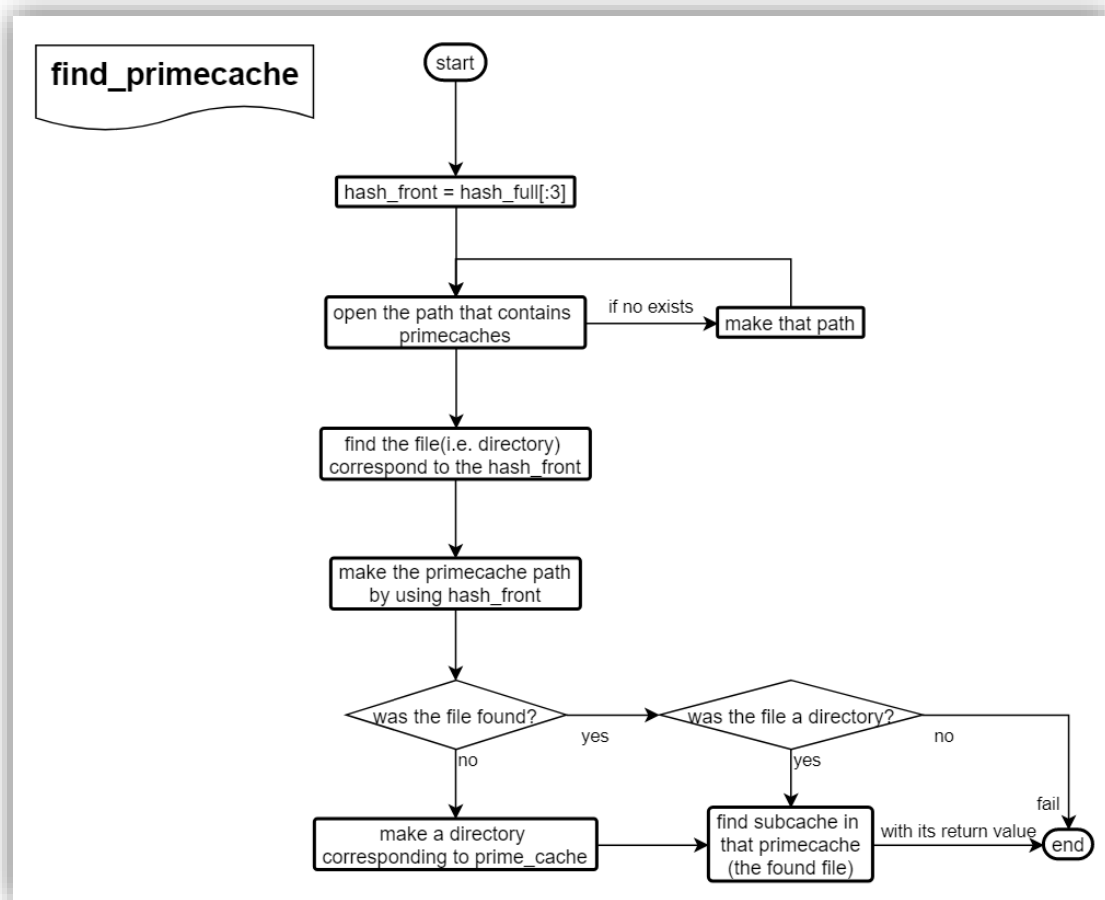


Figure 2.4 find_primecache function flowchart

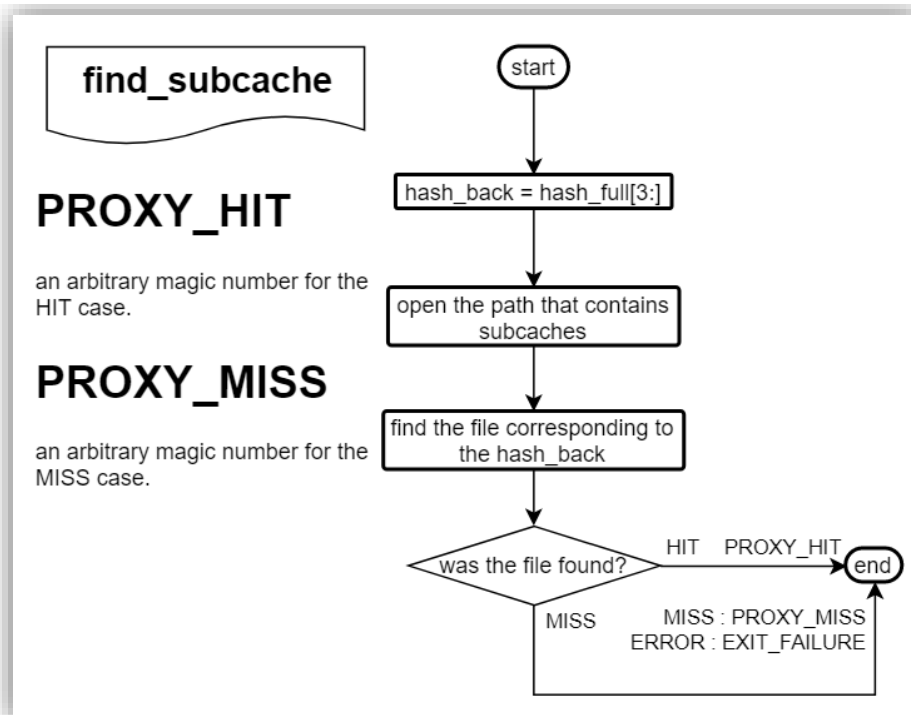


Figure 2.3 find_subcache function flowchart

Figure 2.3의 find_primecache는 해시의 앞부분(primcache)을 인자로 전달받은 경로에서 탐색한다. 만약 해당 폴더가 없다면 그 문자열로 시작하는 해시가 아직 한번도 안나온 것이므로 해당 폴더를 생성해준다. 그 다음 find_subcache를 호출한다.

Figure 2.4의 find_subcache는 위의 find_primecache로부터 호출되는 함수다. 여기서 한번 더 subcache를 탐색하고 찾았다면 HIT를 못찾았다면 MISS를 반환한다.

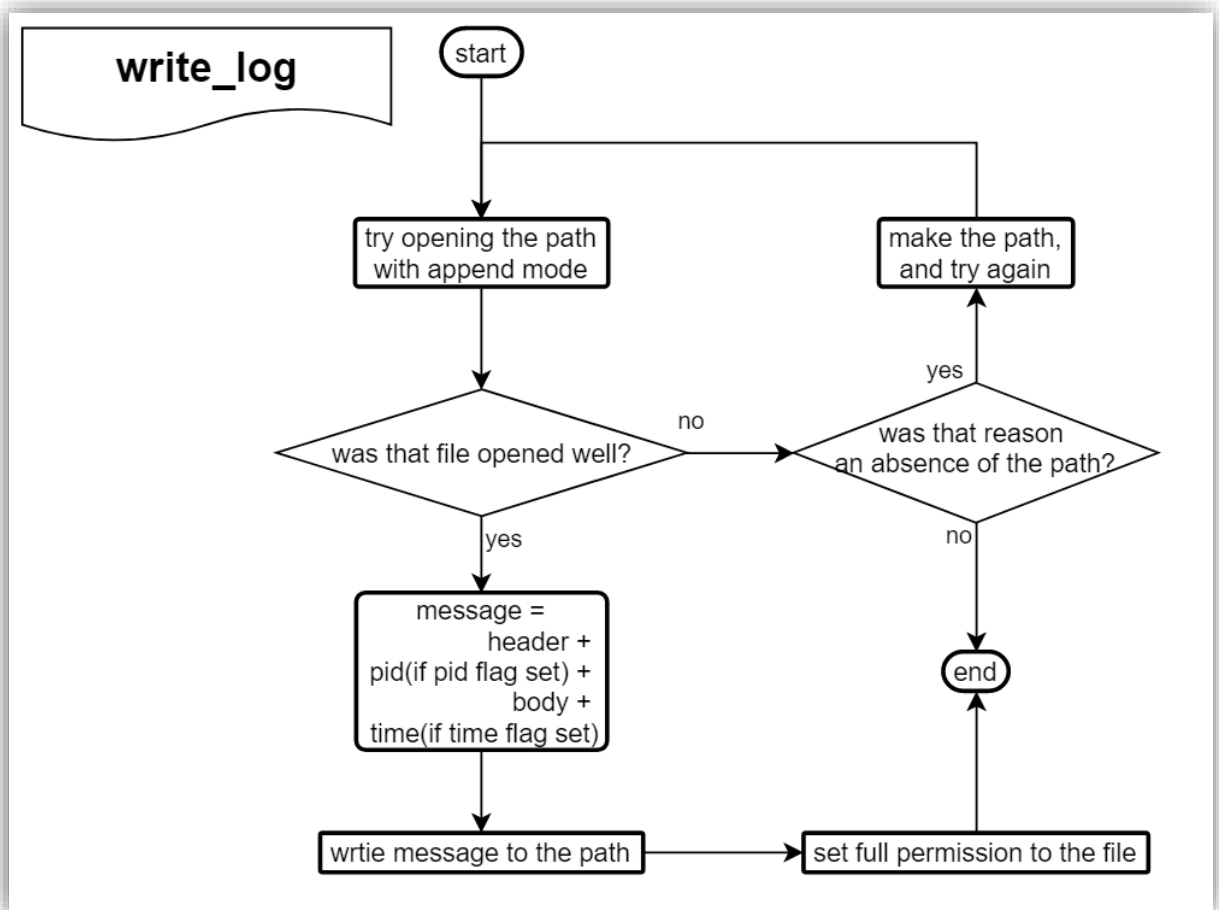


Figure 2.5 write_log function flowchart

Figure 2.5의 write_log 함수는 인자로 전달받은 경로에 메시지들을 지정된 서식에 맞게 조합하여 그 내용을 덧쓰는 역할을 한다. 주 목적은 로깅이지만 본 프로그램에서는 캐시파일을 생성하는데도 사용하였다. 이번 2-1 과제에선 로그에 PID 기능이 추가되어 pid 플래그를 추가하였다. 시간정보나 PID 정보는 해당 값이 set 되어 있을때만 기록된다.

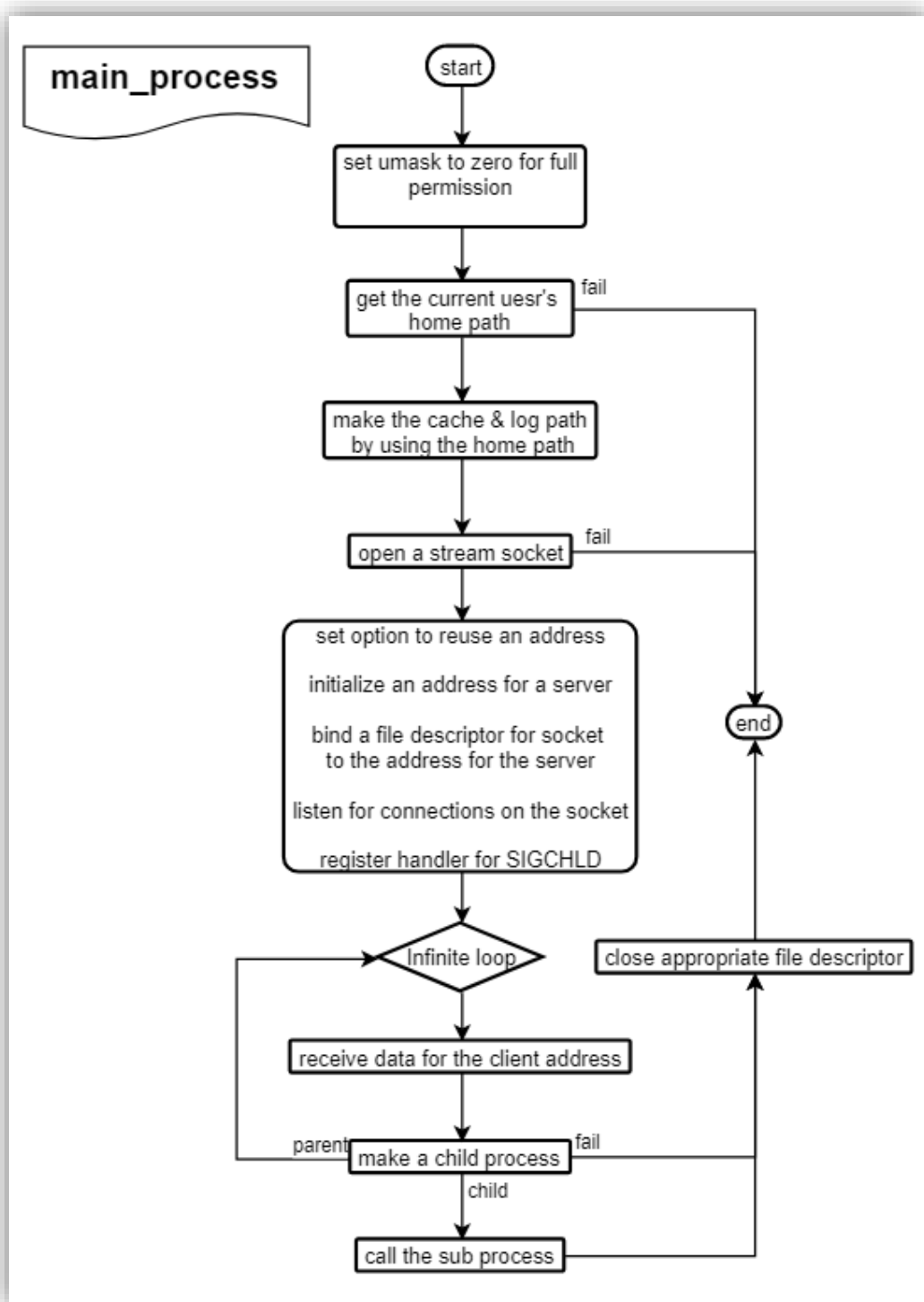


Figure 2.6 main_process function flowchart

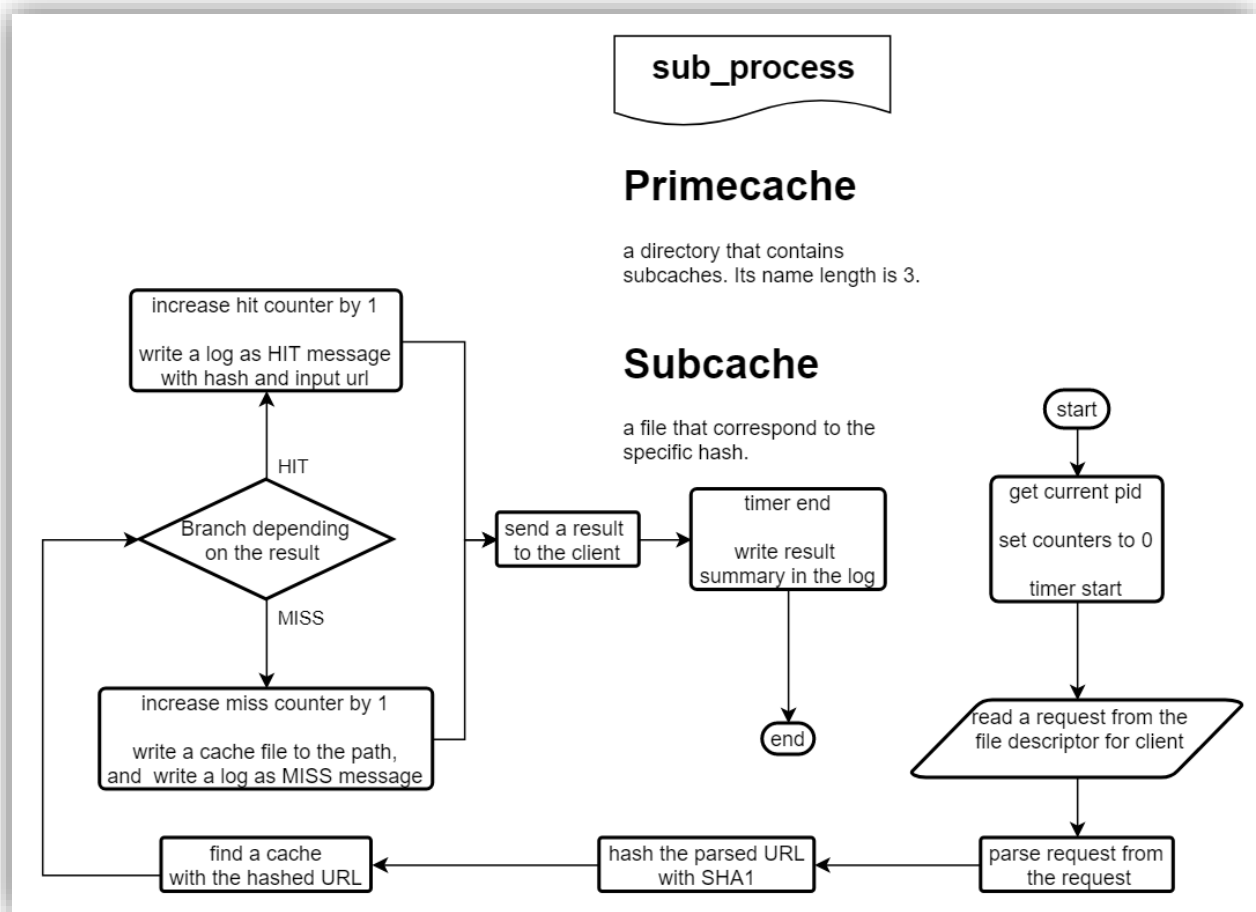


Figure 2.7 sub_process function flowchart

main 은 Figure 2.6 의 main_process 를 호출하는 역할만 하므로 이번 보고서에는 생략하였다. main_process 의 변경되었다고 하긴 했지만 실제 변경된 부분은 stream socket 을 여는 부분에 reuse address 옵션이 추가된 것 뿐이다. 그 외의 동작 방식은 모두 동일하며 클라이언트의 요청을 받아 fork 하여 자식 프로세스를 만들고 그 안에서 sub_process 를 수행하게 된다.

Figure 2.7 의 sub_process 는 이번에 상당 부분 변경되었다. 먼저 기존에 터미널에서 직접 클라이언트의 URL 을 입력받으며 bye 를 입력받을때까지 무한 루프를 돌았었는데 이번 과제에서는 직접 브라우저의 요청 헤더를 파싱하여 추출한 URL 을 해싱에 사용한다. 따라서 순서도의 구조 또한 반복 구조에서 순차 구조로 변경되었다. 또 로그를 기록하는 부분이 현재 의미는 없지만 제거없이 그대로 두라는 요구에 맞추어 로그 부분은 변경하지 않았다.

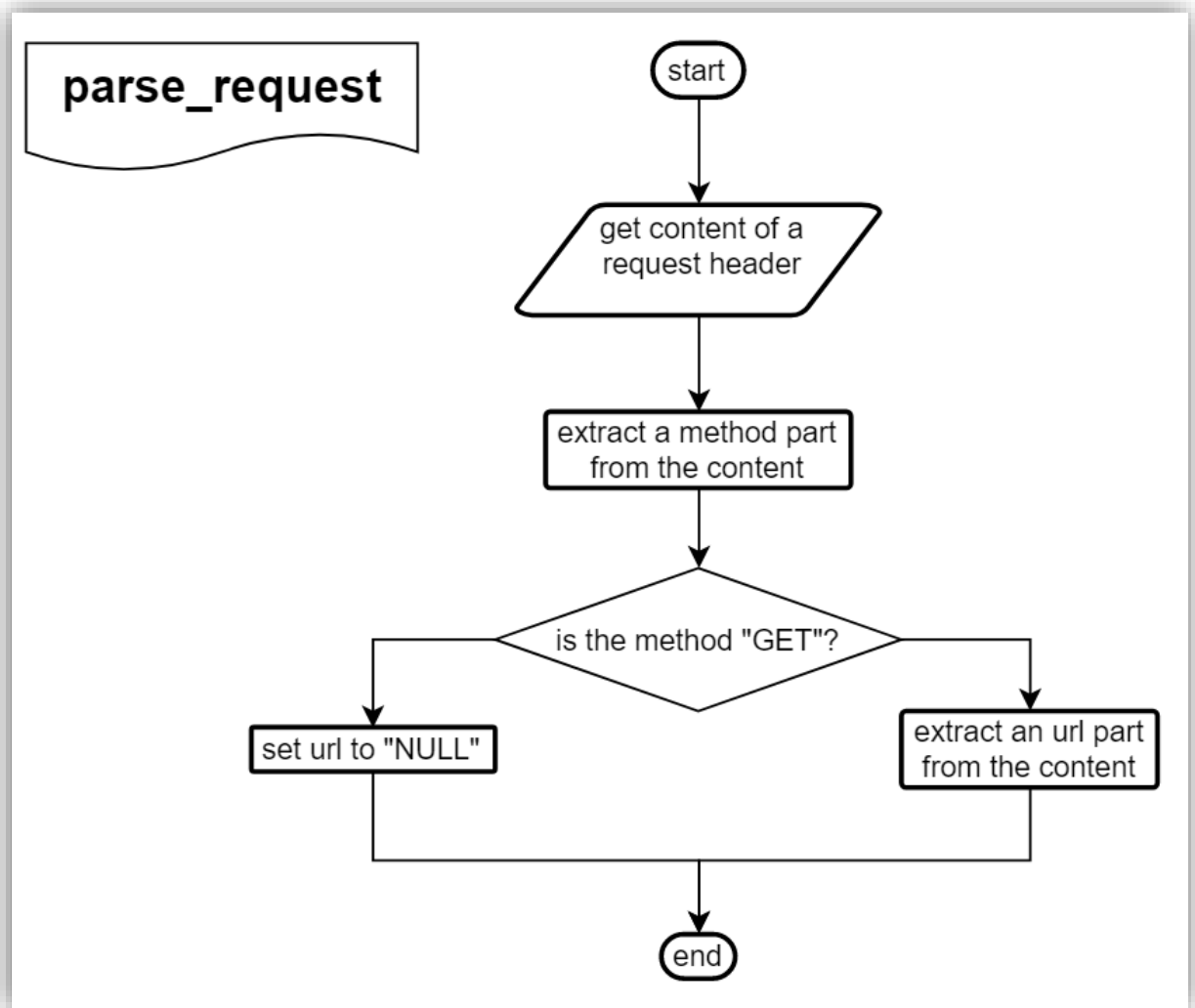


Figure 2.8 parse_request function flowchart

Figure 2.8의 `parse_request`는 요청 헤더 정보가 담긴 버퍼를 받아 메서드와 url을 추출하여 메서드가 GET인 경우 url을 전달받은 인자들이 가리키는 각 파싱 결과 배열에 저장된다. 이외에 경우에는 url에 문자열 "NULL"을 넣는다.

3 Pseudo code

본 report에서 사용된 Psuedo code style은 다음과 같다.

1. 코드를 그대로 갖고온다.
2. 문장으로 치환할 수 있는 부분은 치환하고 굵은 노란색으로 표시한다.
3. 2번을 거친 뒤 나머지 1번에서 가져온 부분 중 언급되지 않은 것들은 제거한다.
4. 코드 블록 밑에서 덧붙일 부가 설명과 중복된 부분도 제거한다.

그리고 2.의 경우 슬라이싱이나 집합등 기타 부분에서 python style을 사용할 것이다.

Figure 3.1 insert_delim function pseudo code

```
* @param str A char array that a delimiter'll be inserted into.
* @param size_max The size of str buffer.
* @param idx The location where the delimiter to be inserted into.
* @param delim A delimiter character.
* @return [int] Success:EXIT_SUCCESS, Fail:EXIT_FAILURE
int insert_delim(char *str, size_t size_max, size_t idx, char delim){

    Check buffer overflow or invalid index

    Insert the delimiter into the position idx in the str.

    return EXIT_SUCCESS;
}
```

insert_delim은 하는 역할만큼이나 단순한 함수로 전달받은 str의 idx 위치에 문자 delim을 추가한다. 딱히 실패할 일은 없지만 이미 str이 할당받은 크기만큼 꽉 차있거나 idx가 str의 길이 보다 큰 경우엔 EXIT_FAILURE를 반환한다.

Figure 3.2 write_log function pseudo code

```
* @param path A const char pointer pointing the path to write the log.
* @param header The header of a log message.
* @param body The body of a log message.
* @param time_ If it is true, write the log with current time. otherwise, don't.
* @param pid_ Append PID information into the log when it is set.
* @return [int] Success:EXIT_SUCCESS, Fail:EXIT_FAILURE
int write_log(const char *path, const char *header, const char *body, bool time_flag, bool pid_flag){
```

```

Get pid of current process and set current_pid to it
// pid number for current process
pid_t pid_current = getpid();

Try opening the path with the append mode
If the try goes fail {
    If its reason is absence of the log path {
        Make the path, and try again
    } else {
        Throw an error
    }
}

Assign new memory block to msg_total by enough space

Message consists of header, PID(if pid_ is set), body, time(if time_ is set)
※ time information is set to current local time

Write message to the path

Make that file have a full permission
If it goes fail, notify it and return FAIL

return EXIT_SUCCESS;
}

```

write_log 의 인자는 path, header, body, time_ 으로 이루어져 있고 이름과 자료형에서 알 수 있다시피 path 는 로그를 작성할 위치, header 와 body 는 메시지의 앞과 뒷부분, time_flag 와 pid_flag 는 기록할 메시지에 특정 정보를 추가할지 말지 결정하는 플래그이다. 굳이 header 와 body 로 나눈 이유는 호출측에서 굳이 추가적인 문자열 연산 없이도 Literal string 과 동적으로 변하는 문자열을 그대로 인자로 넘길 수 있게끔 하기 위해서였다. 그리고 이번 과제에서 비중은 많이 줄었지만 코드 전체 일관성을 위해 플래그를 _접미사 대신 _flag 로 좀 더 명시적으로 변경하였다.

로그 파일을 열때 해당 경로의 부모 디렉터리가 없으면 생성하고 다시 시도해본다. 근데 디렉터리의 부재같은 문제가 아니면 해당하는 에러를 알리고 EXIT_FAILURE 를 반환한다. 그리고 time_ 플래그에 따라 header + body 에 시간 정보를 추가할지 말지 결정하고 그 결과를 log 파일에 작성한다.

그리고 과제 요구사항에 맞춰 해당 파일의 접근권한을 777 로 설정한다.

Figure 3.3 find_primecache function pseudo code

```

* @param path_primecache A const char pointer to the path containing primecaches.

```

```

* @param hash_full A const char pointer to be used as a part of a cache.
* @return [int] HIT:PROXY_HIT, MISS:PROXY_MISS, FAIL:EXIT_FAILURE
int find_primecache(const char *path_primecache, const char *hash_full){
    char hash_front[PROXY_LEN_PREFIX + 1] = {0};

    int result = 0;

    Extract the front part of the hash
    and assign the result into hash_front

    Check whether the path of primecache exist or not
    If not exist{
        create that path, and try opening it again
    }

    Make the full path of the directory which contains subcaches

    Find the primecache that matches with hash_front
    while traversing the path{
        If the primecache was found{
            Check whether it is a directory or not
            If the file was regular, then something's wrong in the cache directory {
                Throw an error
            }
        }
    }

    If there isn't the path of subcache,
    then create that path with full permission

    Find the subcache in the path of the current primecache
    and assign the return value into the result

    return result;
}

```

find_primecache 는 먼저 인자로 받은 전체에서 primecache(해시 앞부분)을 추출한다. 그리고 인자로 받은 경로를 순회하며 해당하는 primecache 를 탐색한다. 만약 경로를 순회하려는데 해당 경로가 없으면 해당 경로를 생성하고 다시 시도한다.

이 때 찾은 파일이 디렉터리가 아니라 파일이면 cache 디렉터리 구조에 문제가 생긴것이므로 이를 알리고 EXIT_FAILURE 를 반환한다.

파일을 못 찾았으면 해당 primecache 로 시작하는 캐시가 아직 생성되지 않은 것이므로 이를 생성하고 find_subcache 를 호출한다.

그리고 이에 대한 HIT 나 MISS 나에 대한 결과를 반환한다.

Figure 3.4 find_subcache function pseudo code

```
* @param path_subcache A const char pointer to the path containing subcaches.
* @param hash_full A const char pointer to be used as a part of a cache.
* @return [int] HIT:PROXY_HIT, MISS:PROXY_MISS
int find_subcache(const char *path_subcache, const char *hash_full){
    struct dirent *pFile = NULL;
    DIR          *pDir  = NULL;

    char hash_back[PROXY_LEN_HASH - PROXY_LEN_PREFIX + 1] = {0};

    Extract the back part of the hash
    and assign the result into hash_back

    Find the subcache while traversing the path

    If the file is found {
        return PROXY_MISS;
    } else {
        return PROXY_HIT;
    }
}
```

find_subcache 함수는 find_primecache 에서 찾은 primecache(subcache 들이 들어있는 폴더)에서 주어진 subcache(hash 의 뒷부분)를 탐색한다. 그리고 캐시를 찾았냐 못찾았냐에 따라 PROXY_HIT 또는 PROXY_MISS 를 반환한다.

Figure 3.5 sub_process function pseudo code

```
* @param path_cache The path containing primecaches.
* @param path_log The path containing a logfile
* @param fd_client The client file descriptor.
* @param addr_client The address struct for the client
* @return [int] Success:EXIT_SUCCESS
int sub_process(const char *path_cache, const char *path_log, int fd_client, struct
sockaddr_in addr_client){

    // counters for hit
    size_t count_hit = 0;
    size_t count_miss = 0;

    Timer start

    Read a request header from the client

    Parse the request header and extract an url, method string
```

```

Hash the parsed url and find the cache with it

Insert a forward slash delimiter at the 3rd index in the hash_url

Make a path for fullcache

switch(result){
    case PROXY_HIT:
        count_hit += 1;
        Write a log as hashed url and parsed url in log path
        Write the HIT message to a response message
        break;

    case PROXY_MISS:
        count_miss += 1;
        Create a dummy cache in the cache path
        Write a log as hashed url and parsed url in log path
        Write the MISS message to a response message
        break;

    default:
        break;
}

Make a response header

Send the response header and the response message to the client

Timer end

Make a string for terminating the log and write it

return EXIT_SUCCESS;
}

```

sub_process 의 길이가 굉장히 길고 난해하다. 사용되지 않는 기능까지 그대로 두려다보니 다소 번잡해진것 같다. 그래도 강조 표시된 부분만 읽어보면 sub_process 의 역할은 클라이언트의 요청 헤더를 읽고 파싱하여 url 값을 뽑아낸 뒤 이를 해싱하여 이전과 같이 캐시를 찾아 HIT or MISS 판별하는 것이다.

그리고 이 정보를 가지고 다시 클라이언트에 결과값을 반환해준다.

이전 과제와 달리 반복 구조가 없어졌으며 요청을 한번 처리하면 바로 종료된다.

Figure 3.6 main_process function pseudo code

```
* @return [int] Success:EXIT_SUCCESS, Fail:EXIT_FAILURE
int main_process(){

    Set full permission for the current process.

    Try getting current user's home path
    and concatenate cache and log paths with it

    Try open a stream socket

    Set option to reuse address

    Initialize address for server

    Bind a file descriptor for the socket to the address for the server

    Listen for connections on a socket

    Register handler_child for handling SIGCHLD

    Interact with the client{
        Receive data for the client address

        Fork a process { in child process
            Call the sub_process with file decriptor for the client

            Close appropriate file descriptors
            return EXIT_SUCCESS;
        }
        Close appropriate file descriptors
    }

    Close appropriate file descriptor
    return EXIT_SUCCESS;
}
```

main_process 는 따로 유저로부터 입력을 받는 인터페이스를 제공하는 대신 주어진 포트로 소켓을 열어 클라이언트의 요청을 받을 수 있도록 하였다. 이 Interact with the client 블록 내 Receive data for the client address 부분에서 클라이언트의 요청을 받아 fork 한 자식 프로세스내에서 file decriptor for the client 를 인자로 하여 sub_process 를 호출한다. 그리고 원본 프로세스는 다시 다른 클라이언트의 요청을 받을 준비를 한다.

이번 과제에선 강의자료에 소개된 setsockopt 함수를 사용하여 TIME_WAIT 상태일때 발생하는 can't bind local address 문제를 해결하도록 옵션 설정 부분이 추가 되었다.

Figure 3.7 parse_request function pseudo code

```
* @param buf A buffer containing request
* @param method A char array to contain extracted method
* @param url A char array to contain extracted url, it'll be "NULL" when method isn't GET
int parse_request(const char *buf, char *url, char *method){

    Extract a method part from the buf

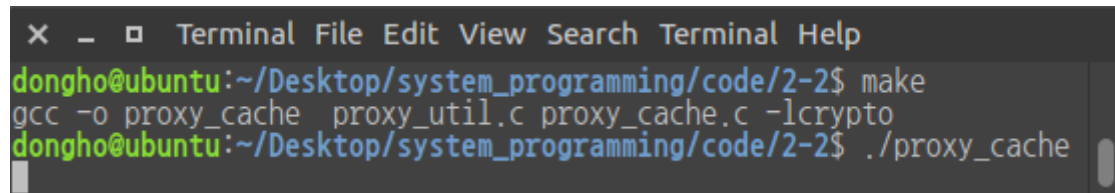
    If the method is GET then extract an url part
    Else then the url be "NULL"

    return EXIT_SUCCESS;
}
```

이번에 클라이언트의 요청 헤더를 파싱하여 url 을 추출하는 함수이다. 이 함수는 강의자료에 올라온 예제와 동일한 기능을 수행한다. 먼저 요청 헤더에서 메서드를 읽고 GET 인 경우 url 을 읽는 식이다. 반환은 포인터를 통해 수행되고 method 가 GET 이 아니면 url 은 NULL 값이 할당된다.

현재는 따로 예외처리가 없으나 후에 확장을 위해 반환형은 int 로 하였다.

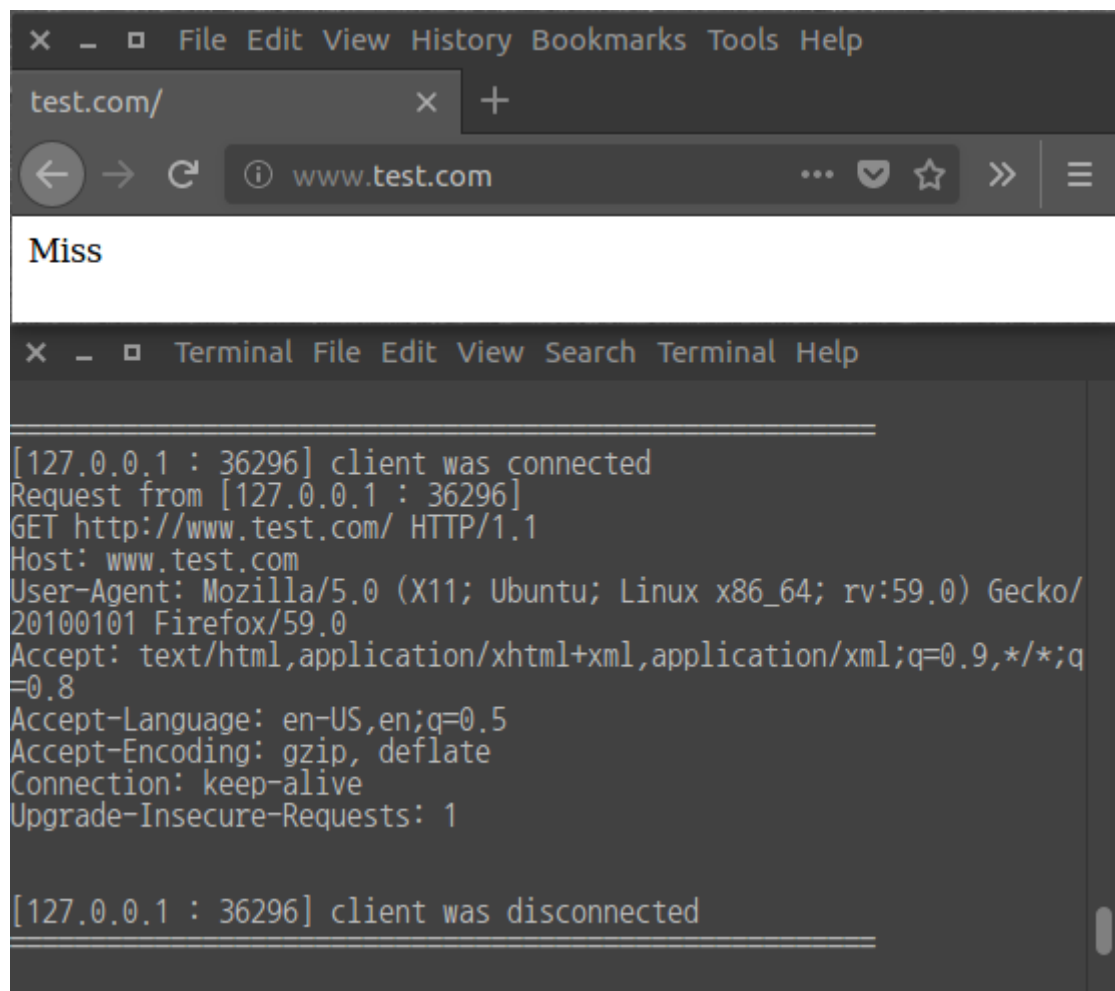
4 결과화면



```
Terminal File Edit View Search Terminal Help
dongho@ubuntu:~/Desktop/system_programming/code/2-2$ make
gcc -o proxy_cache proxy_util.c proxy_cache.c -lcrypto
dongho@ubuntu:~/Desktop/system_programming/code/2-2$ ./proxy_cache
```

Figure 4.3.1 proxy_cache exec init

이번부터 다시 server & client 대신 proxy_cache 하나로 통합되었다. 보다시피 처음 실행시엔 아무것도 뜨지 않는다.



The top part of the image shows a Firefox browser window with the address bar set to `test.com/` and the page content displaying `Miss`. The bottom part shows a terminal window with the following output:

```
Terminal File Edit View Search Terminal Help

[127.0.0.1 : 36296] client was connected
Request from [127.0.0.1 : 36296]
GET http://www.test.com/ HTTP/1.1
Host: www.test.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:59.0) Gecko/20100101 Firefox/59.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1

[127.0.0.1 : 36296] client was disconnected
```

Figure 4.2 client send request for the first time

client(firefox browser)에서 www.test.com에 대한 페이지 요청을 보냈다. 현재 firefox 브라우저 프록시 설정에서 proxy_cache에서 열어둔 소켓의 포트로 연결 해두어 Figure 4.2 아래와 같이 요청 정보가 뜨는 것을 볼 수 있다. 사실 요구 사항엔 없었는데 조교님이 출력해도 상관없다고 하셔서 디버깅으로도 쓸겸 그냥

출력하였다.

현재 www.test.com에 대한 요청은 처음이라 Miss가 반환되었으며 마지막 disconnected를 통해 프로세스가 Hit/Miss를 판별하고 바로 종료된것을 볼 수 있다.

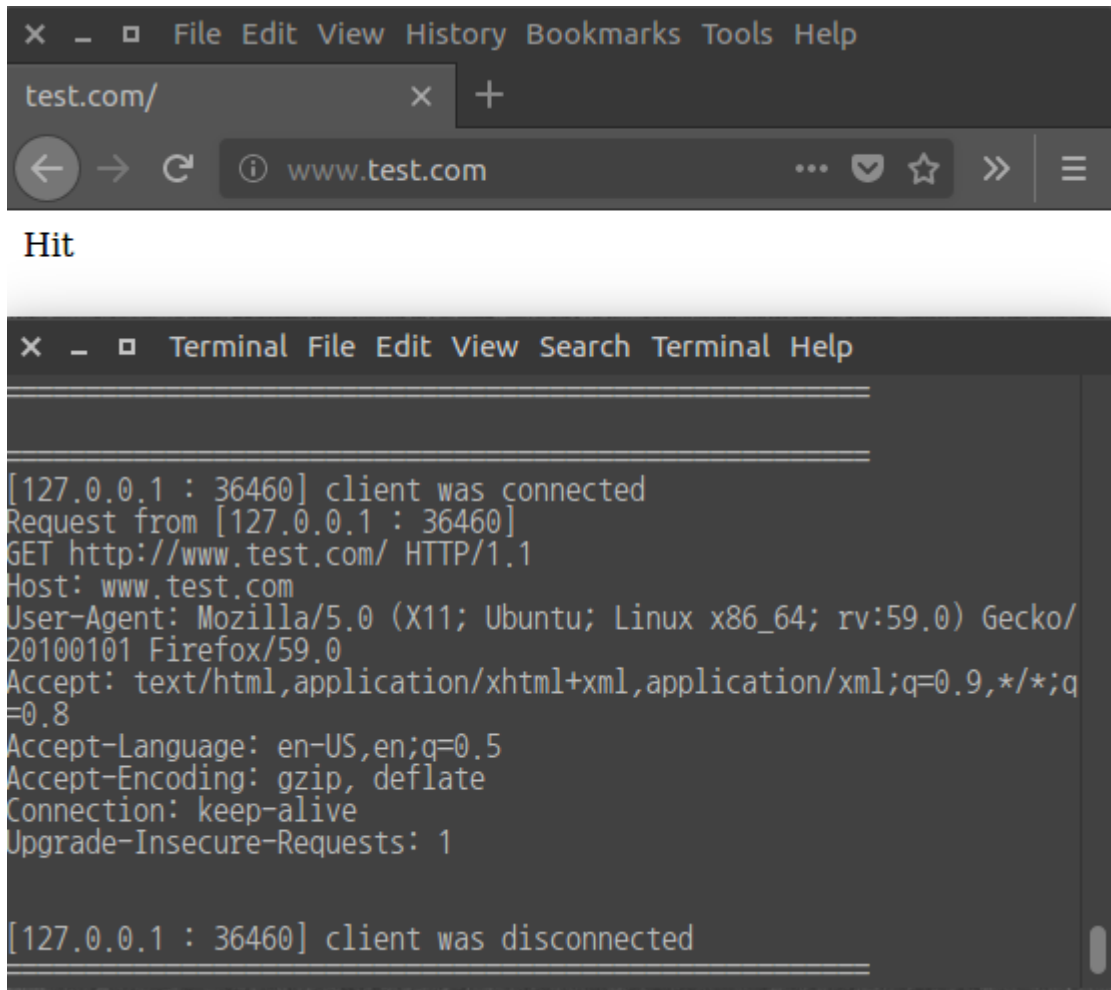


Figure 4.3 client send again

다시 같은 주소로 브라우저에서 요청을 보내니 이번에는 Hit가 뜨는 것을 볼 수 있다. 아래 출력로그를 보면 서로 다른 요청임을 확인할 수 있다.

5 결론 및 고찰

이번 과제에서는 지금까지 구현했던 내용들을 이번에는 좀 더 실전에서처럼 브라우저 요청을 중간에서 가로채 캐싱을 수행하고 HIT 또는 MISS를 반환하는 기능을 구현하였다. 그 과정에서 url을 터미널에서 수동으로 입력받는 부분이 없어지고 반복 구조 대신 그냥 한번 요청을 처리하면 자식 프로세스는 종료되게 만들었다.

이 과정에서 기존에 한 프로세스의 HIT MISS 통계 정보, 얼마나 켜져있었는지에 대한 시간이 의미가 없어져서 지웠는데 다시 그냥 넣으라 하셔서 다시 넣었다. 앞으로 쓸지 안쓸지는 모르겠다.

그리고 이번 과제부터 소스코드를 분할할 수 있게 되어 sub와 main process들을 제외하고 모두 proxy_util이라는 이름의 파일로 분할하였다. 최근 과제들은 1-때 구현한 부분을 건들지 않고 있으며 따라서 proxy_cache의 process 함수들에만 집중하고 싶었기 때문이다.

그리고 이번과제는 딱히 레퍼런스를 참고할만한게 없었다. 요구사항도 새로 추가된게 아니라 기존에 있던 내용을 조금만 수정하면 되었기 때문이다.

마지막으로 이번 과제를 하면서 느낀점은 이제 정말 프록시 캐시 서버를 만드는 느낌이 나는것 같다. 브라우저 요청을 중간에 가로채서 캐싱을 처리하는게 신기했다.

6 참고 레퍼런스

강의자료 18-1_SSLab_week08_Proxy+2-2.pdf 의 setsockopt를 참고하여 can't bind local address문제를 해결하였다.