

System programming

Assignment #2-1 (proxy server)



학 과 : 컴퓨터 공학과

담당교수 : 황호영

분 반 : 목34

학 번 : 2016722092

성 명 : 정동호

제출날짜 : 2018-04-27

목차

| | |
|---------------------|----|
| 1 INTRODUCTION..... | 3 |
| 2 FLOWCHART..... | 3 |
| 3 PSEUDO CODE | 8 |
| 4 결과화면..... | 14 |
| 5 결론 및 고찰 | 16 |
| 6 참고 레퍼런스..... | 17 |

1 Introduction

이전 과제에서 구현했던 멀티프로세싱을 활용하여 여러 클라이언트로부터 동시에 요청을 받을 수 있는 서버를 구현한다. 동시에 기존 proxy_cache 파일 하나로 합쳐져 있던 요청과 처리 부분을 server 와 client 파일로 나누어 구현한다.

2 Flowchart

2-1 과제에서 변경된 순서도는 main_process, sub_process, write_log이고 추가된 부분은 client의 main 함수이다. 변경되지 않은 부분은 이전 과제 순서도 그대로 사용하였다.

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

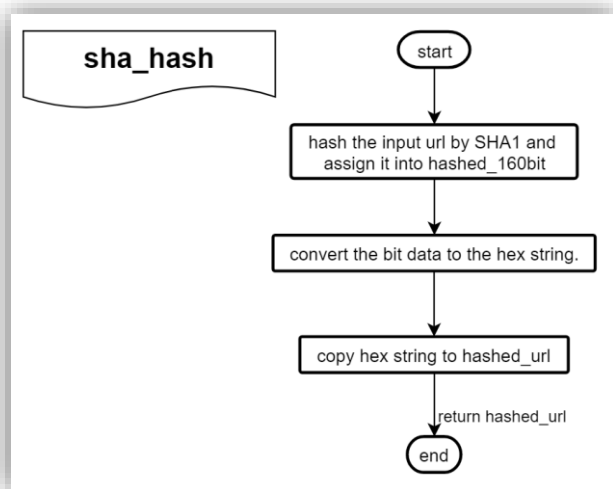


Figure 2.2 sha_hash function flowchart

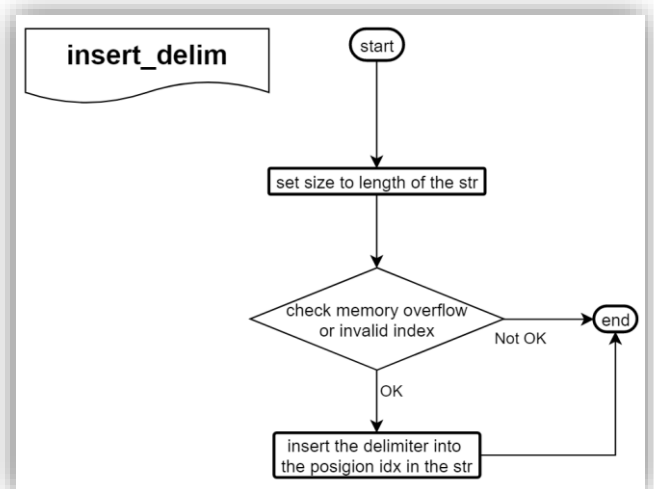


Figure 2.1 insert_delim function flowchart

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.

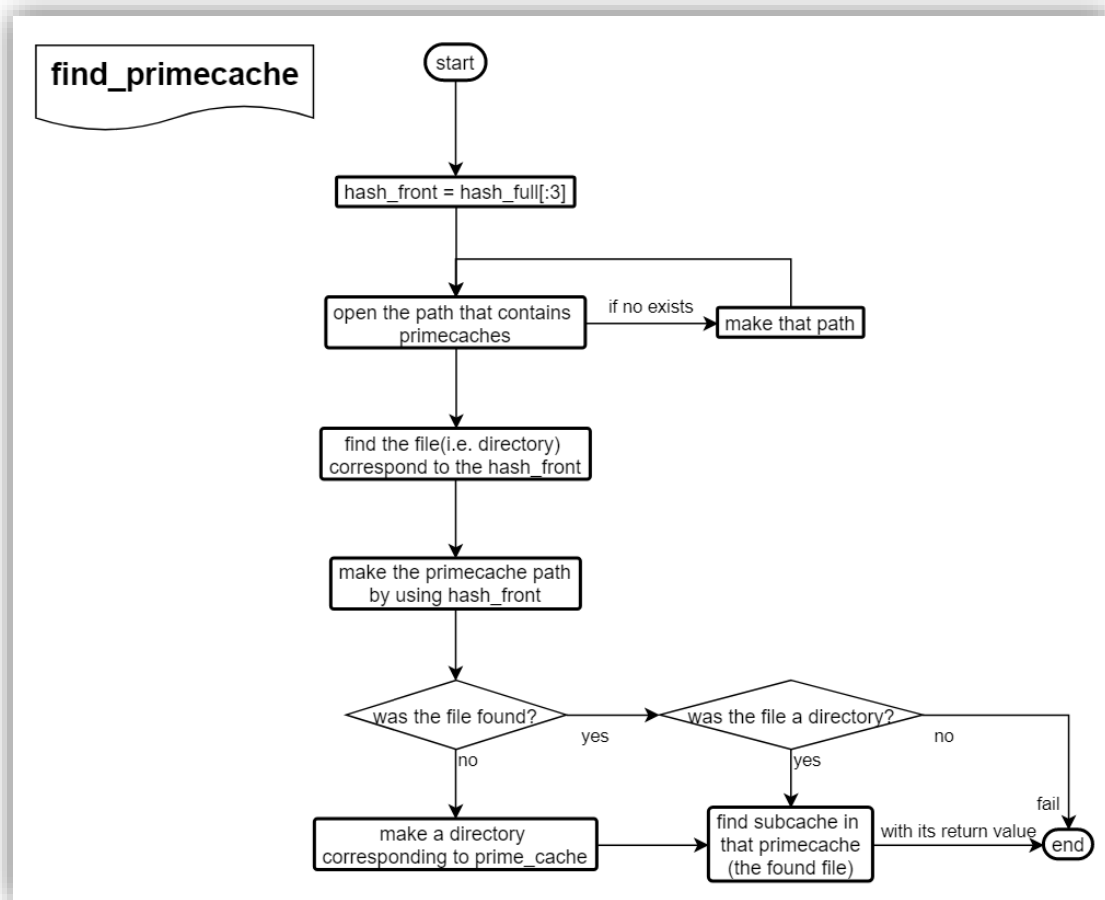


Figure 2.4 find_primecache function flowchart

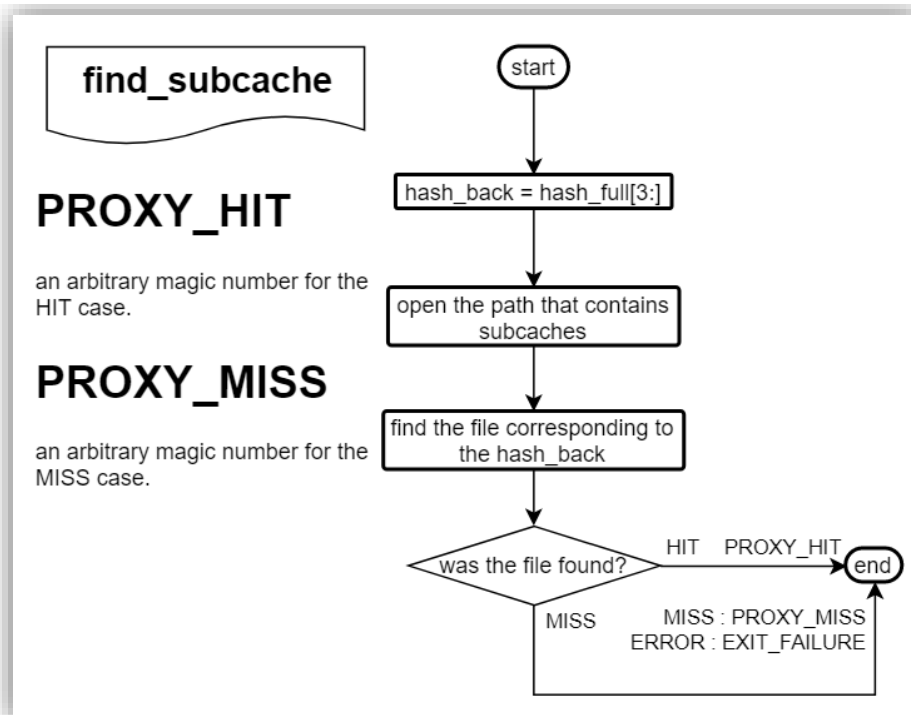


Figure 2.3 find_subcache function flowchart

Figure 2.3의 find_primecache는 해시의 앞부분(primecache)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.

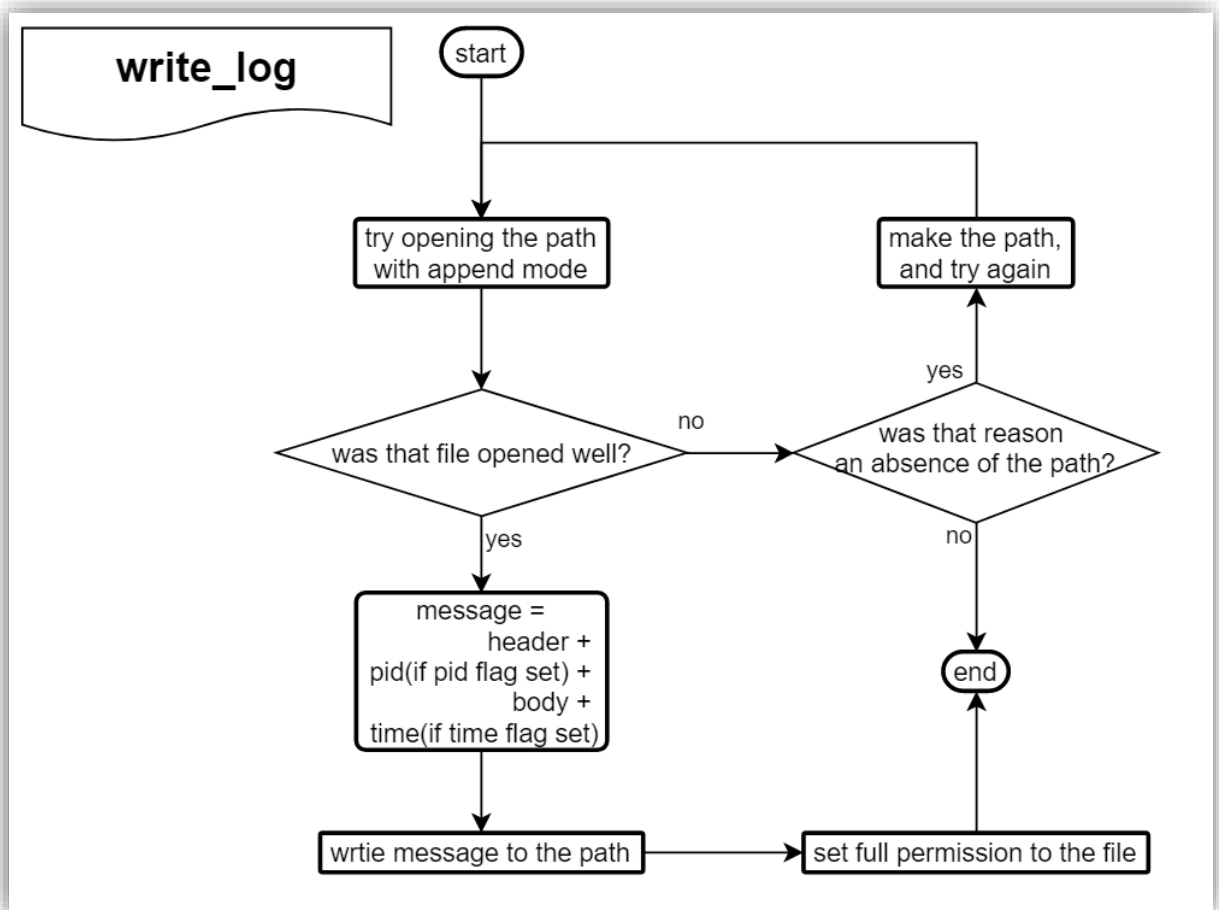


Figure 2.5 write_log function flowchart

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.

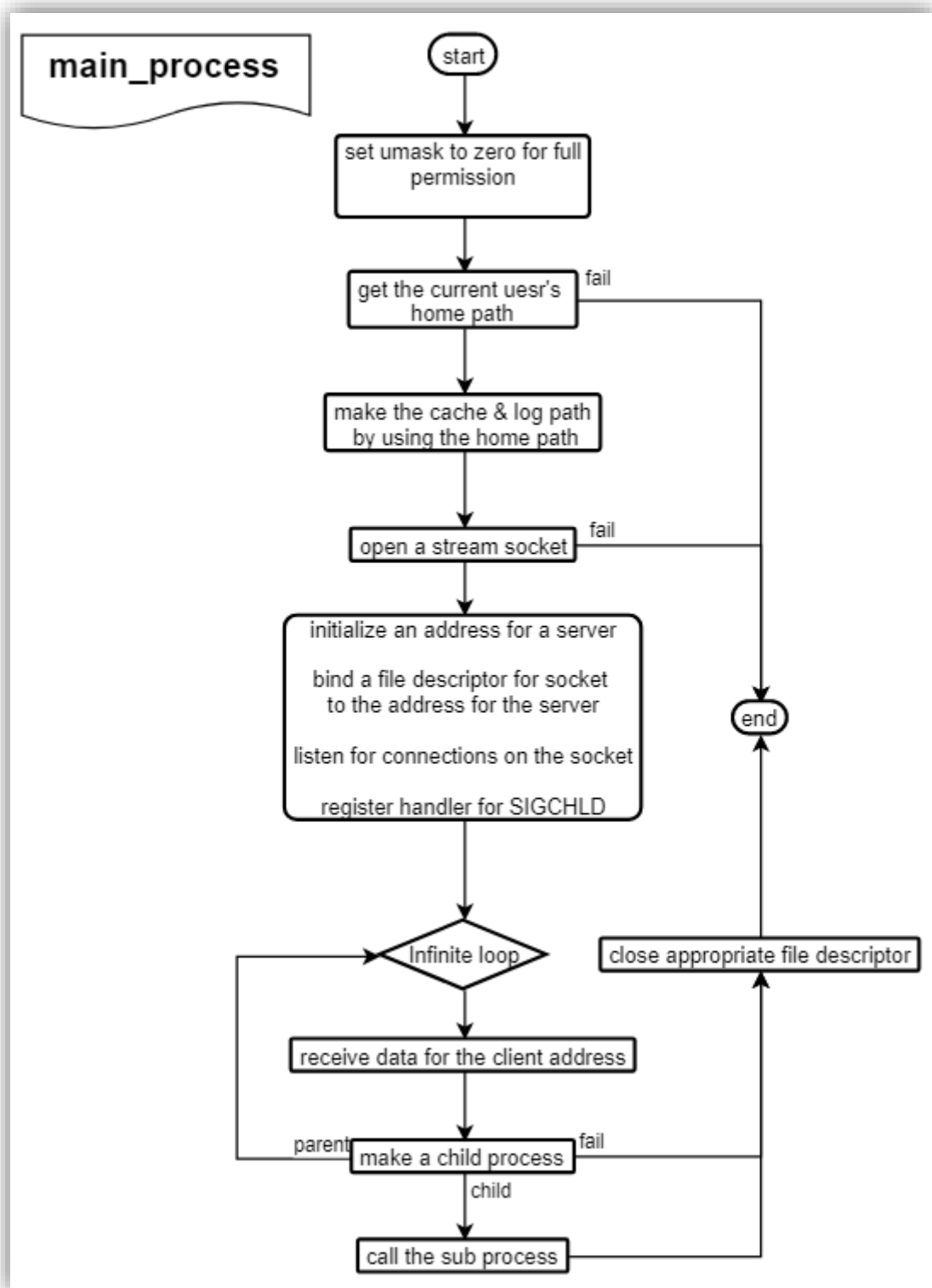


Figure 2.6 main_process function flowchart

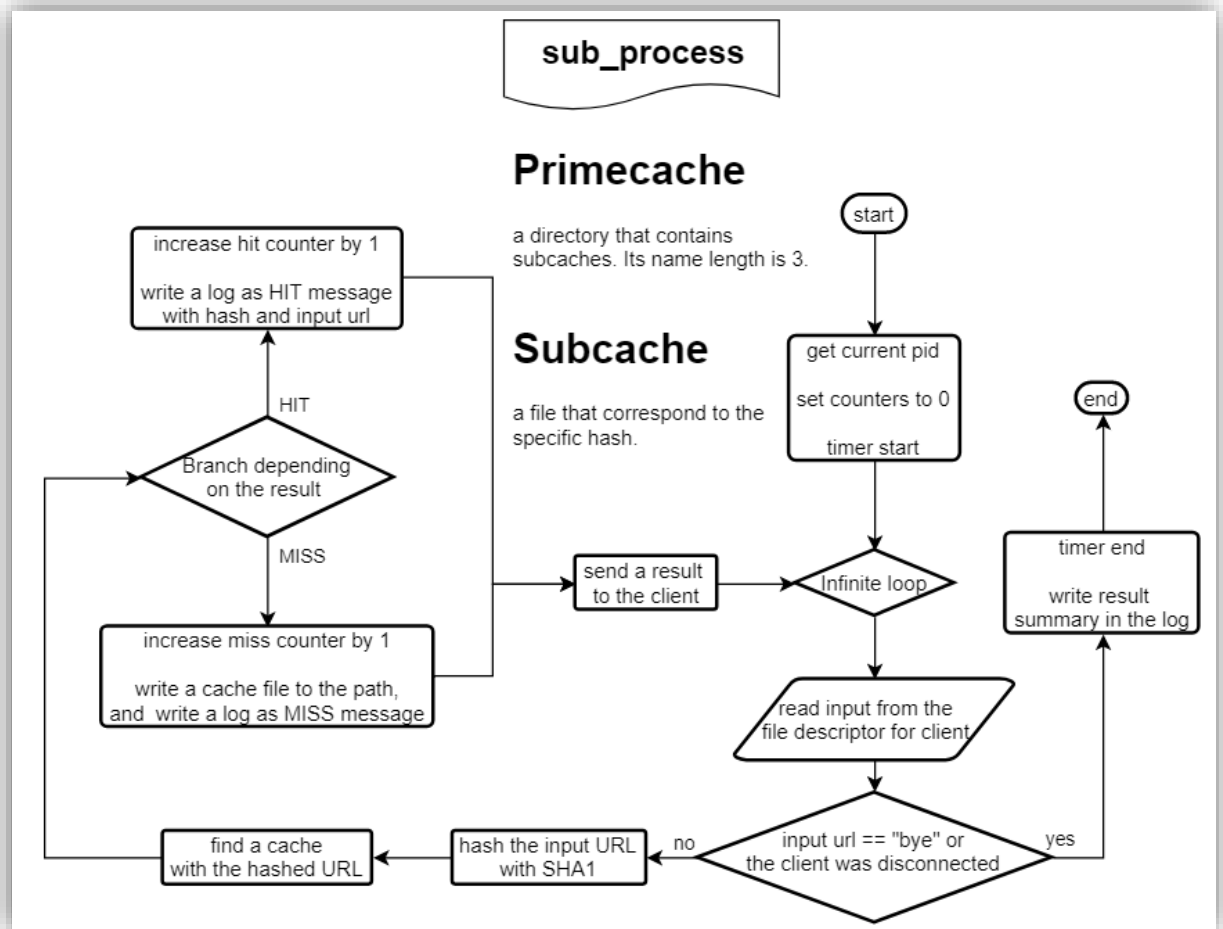


Figure 2.7 sub_process function flowchart

일단 main은 Figure 2.6의 main_process를 호출하는 역할만 하므로 이번 보고서에는 생략하였다. 특히 이번 보고서에서 main_process와 sub_process의 역할이 크게 바뀌었는데 먼저 main_process는 기존에 유저로부터 command를 입력받아 프로세스를 생성했으나 이번에는 소켓을 열고 클라이언트로부터 연결을 요청받아 그에 해당하는 프로세스를 fork하여 sub_process를 수행하도록 한다.

Figure 2.7의 sub_process는 역할이 크게 바뀌었다기보다 입출력 부분이 많이 바뀌었다고 볼 수 있다. 기존에 한 프로그램에서 유저로부터 URL을 입력받았다면, 이번에 변경된 버전에서는 인자로 전달받은 client file descriptor를 통해 URL을 받고 캐싱 결과를 반환하는 식이다.

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_, bool pid_){
```



```

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 메세지의 앞과 뒷부분, time_ 과 pid_ 는 기록할 메세지에 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.

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 subcache.
* @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_log The path containing a logfile.
* @param path_cache The path containing primecaches.
* @param fd_client The client file descriptor.
* @return [int] Success:EXIT_SUCCESS
int sub_process(const char *path_log, const char *path_cache, int fd_client){

    char url_input[PROXY_MAX_URL] = {0};
    char url_hash[PROXY_LEN_HASH] = {0};

    // counter for HIT and MISS cases
    size_t count_hit = 0;
    size_t count_miss = 0;

    Timer start

    Receive inputs till the input is 'bye' or client is connected{

        If the input is 'bye' then break loop
```

```

    Hash the input URL and find the cache with it

    Insert a slash delimiter at the 3rd index in the url_hash

    Make a path for fullcache

    switch(result){
        case PROXY_HIT:
            count_hit += 1;
            Write a log as url_hash and url_input in log path
            Send the HIT case to hte client
            break;

        case PROXY_MISS:
            count_miss += 1;
            Create a dummy cache in the cache path
            Write a log as url_hash and url_input in log path
            Send the MISS case to hte client
            break;

        default:
            break;
    }
}

Timer end

Make a string for terminating the log and write it

return EXIT_SUCCESS;
}

```

sub_process 는 인자로 전달받은 client file descriptor 를 통해 URL 을 전달받고 find_primecache 를 수행한 결과를 다시 client file descriptor 를 통해 반환한다. 이러한 반복 작업은 해당 클라이언트로부터 bye 를 입력받거나 클라이언트와 연결이 종료되었을 때 중단된다. 그리고 반복문을 빠져나와 종료시 요약 정보(HIT & MISS 수, 실행 시간)를 인자로 전달받은 로그 경로에 기록한다.

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

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

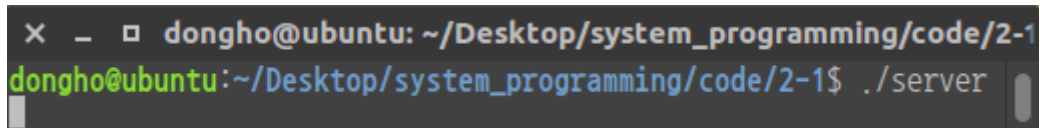
        Close appropriate file descriptors
        return EXIT_SUCCESS;
    }
    Close appropriate file descriptors
}

Close appropriate file descriptor
return EXIT_SUCCESS;
}

```

main_process 는 따로 유저로부터 입력을 받는 인터페이스를 제공하는 대신 주어진 포트 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.

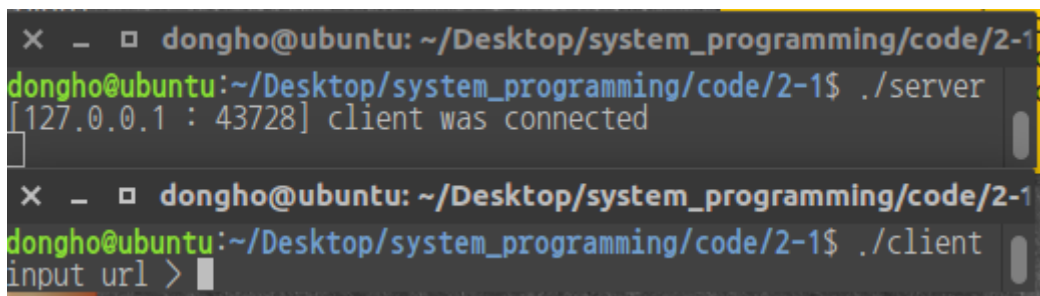
4 결과화면



```
× _ □ dongho@ubuntu: ~/Desktop/system_programming/code/2-1
dongho@ubuntu:~/Desktop/system_programming/code/2-1$ ./server
```

Figure 4.1 server exec init

처음 server를 실행시키면 유저로부터 입력을 받을 수도 없고 아무것도 나오지 않는다. 이 상황에서 client를 실행시킨 결과는 다음과 같다.

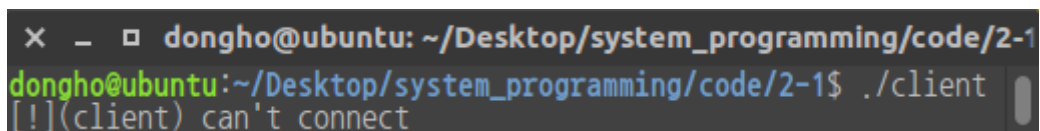


```
× _ □ dongho@ubuntu: ~/Desktop/system_programming/code/2-1
dongho@ubuntu:~/Desktop/system_programming/code/2-1$ ./server
[127.0.0.1 : 43728] client was connected

× _ □ dongho@ubuntu: ~/Desktop/system_programming/code/2-1
dongho@ubuntu:~/Desktop/system_programming/code/2-1$ ./client
input url >
```

Figure 4.2 client connects server

client에서 URL을 입력받을 수 있고 server 프로세스에서 클라이언트에 대한 주소 : 포트번호로 이루어진 정보와 함께 연결이 완료되었음을 알려준다.



```
× _ □ dongho@ubuntu: ~/Desktop/system_programming/code/2-1
dongho@ubuntu:~/Desktop/system_programming/code/2-1$ ./client
[!](client) can't connect
```

Figure 4.3 client connect fail

만약 server가 실행중이지 않을 때 client를 실행시키면 위와 같은 예외처리를 하였다.

```
× - □ dongho@ubuntu: ~/Desktop/system_programming/code/2-1
dongho@ubuntu:~/Desktop/system_programming/code/2-1$ ./server
[127.0.0.1 : 43728] client was connected
[127.0.0.1 : 43730] client was connected
□

× - □ dongho@ubuntu: ~/Desktop/system_programming/code/2-1
dongho@ubuntu:~/Desktop/system_programming/code/2-1$ ./client
input url > www.naver.com
MISS
input url > □

× - □ dongho@ubuntu: ~/Desktop/system_programming/code/2-1
dongho@ubuntu:~/Desktop/system_programming/code/2-1$ ./client
input url > www.naver.com
HIT
input url > www.google.co.kr
MISS
input url > █
```

Figure 4.4 input url from multiple clients

client를 동시에 하나 더 열어 테스트를 진행하였다. 아래 client의 경우 www.naver.com의 주소를 처음 입력함에도 불구하고 이전 client에서 입력한 www.naver.com 덕분에 HIT를 반환하는 것을 볼 수 있다.

```
× - □ dongho@ubuntu: ~
dongho@ubuntu:~$ cat ~/logfile/logfile.txt
[MISS] ServerPID : 44995 | www.naver.com - [2018/04/27, 21:51:36]
[HIT] ServerPID : 45063 | fed/818da7395e30442b1dcf45c9b6669d1c0ff6b - [2018/04/27, 21:52:01]
[HIT]www.naver.com
[MISS] ServerPID : 45063 | www.google.co.kr - [2018/04/27, 21:54:07]
dongho@ubuntu:~$ █
```

Figure 4.5 current log content

4.4 까지의 상황을 로그로 확인해보면 위와 같다.

```
dongho@ubuntu: ~/Desktop/system_programming/code/2-1
dongho@ubuntu:~/Desktop/system_programming/code/2-1$ ./server
[127.0.0.1 : 43728] client was connected
[127.0.0.1 : 43730] client was connected
[127.0.0.1 : 43728] client was disconnected

dongho@ubuntu:~/Desktop/system_programming/code/2-1$ ./client
input url > www.naver.com
MISS
input url > bye
dongho@ubuntu:~/Desktop/system_programming/code/2-1$
```

Figure 4.6 input bye to disconnect

client에서 bye를 입력하여 종료하였고 server에서도 disconnect를 알렸다. 근데 실제 코드를 보면 알겠지만 disconnect를 명시적으로 처리하는 부분이 없다. 이 부분은 아래 고찰에서 얘기하겠다.

```
dongho@ubuntu:~$ cat ~/logfile/logfile.txt
[MISS] ServerPID : 44995 | www.naver.com - [2018/04/27, 21:51:36]
[HIT] ServerPID : 45063 | fed/818da7395e30442b1dcf45c9b6669d1c0ff6b - [2018/04/27, 21:52:01]
[HIT] www.naver.com
[MISS] ServerPID : 45063 | www.google.co.kr - [2018/04/27, 21:54:07]
[Terminated] ServerPID : 44995 | run time: 968 sec. #request hit : 0, miss : 1
[Terminated] ServerPID : 45063 | run time: 1393 sec. #request hit : 1, miss : 1
dongho@ubuntu:~$
```

Figure 4.7 log when terminate all client

나머지 client 하나도 종료한 뒤 로그 내용을 출력해봤다. 각 클라이언트의 요청을 처리하기 위해 fork된 프로세스의 PID와 각각의 요약정보들이 같이 기록된 것을 볼 수 있다.

5 결론 및 고찰

이번 과제는 실제 변경하고 구현할 부분은 가장 적었던 것 같다. 그럼에도 불구하고 가장 많은 시간을 소요했는데 그 이유는 소켓 프로그래밍에 대한 지식이 전혀 없었기 때문이다. 소켓에 대한 개념과 본 과제에서 사용된 갖가지 함수들 모두 너무 생소했다.

먼저 강의자료에서 예시로 주어진 handler_child 함수내 while문도 처음에 이해가 안가서 이것저것 찾아봤다. 그 결과 자식 프로세스가 종료될 때 자동으로 자식 프로세스에 대한 처리를 해주는 작업이라는 것을 알게 되었다.

두번째로 궁금했던 점은 Figure 4.6에서도 언급했지만 disconnect 감지에 대한 부분이다. 분명 코드에는 그런 작업을 처리하는 코드가 안보이는데 실제로 돌려보면 되는게 의아했다. 코드 구조상 disconnect를 처리하는 부분은 read가 될 수 밖에 없었는데 man read로 아무리 봐도 disconnect에 대한 부분을 찾을 수 없었다. 알고보니 read에 대한 section이 2개 있는데 section 3에서 그 답을 찾을 수 있었다. read 시스템 콜과 라이브러리 함수 두개가 있는데 첫번째 인자로 소켓을 전달하면 read는 내부적으로 recv를 호출한다.(If `fd` refers to a socket, `read()` shall be equivalent to `recv()` with no flags set.) 그리고 recv는(The return value will be 0 when the peer has performed an orderly shutdown.) peer가 연결을 종료했을 때 0을 반환한다. 따라서 client 종료시 server에선 0을 전달받아 while문을 빠져나갈 수 있는 것이었다.

그리고 이번 과제 강의자료 예시에서 STDIN_FILENO란 것을 봤는데 stdin과 뭐가 다른지 궁금했는데 전자는 file descriptor에 대한 상수이고 후자는 파일 포인터의 형태라는 것을 알게 되었다.

그리고 inet_addr 함수에 대해 알아보던 중 에러 발생시 -1을 반환하는 문제에 대해 알게되었다. 왜냐하면 -1에 해당하는 255.255.255.255또한 유효한 주소이기 때문이다. 이에 대한 대안으로 inet_aton과 inet_ntop을 알게되었고 inet_pton이 좀 더 상위호환으로 보여 이를 사용하였다.

마지막으로 과제와 별 관련은 없는데 테스트 도중 ^Z로 종료시 다음 server 실행이 제대로 되지 않는 문제의 원인을 파악하던 중 ^Z, ^C, ^D 에 대한 의미와 차이를 알게되었다. 알고보니 가끔 프로세스 종료로 사용했던 ^Z가 알고보니 종료가 아니라 백그라운드로 전환하는것 뿐이었다. ^C가 원래 해야했던 인터럽트였고 ^D는 EOF라는걸 새삼 알게되었다.

이번 과제는 많은걸 배울 수 있어서 뿌듯했고 다음 과제도 기대된다.

6 참고 레퍼런스

<https://kldp.org/node/37061>

clean up child process에 대한 설명

<https://linux.die.net/man/3/read>

<https://linux.die.net/man/2/recv>

어떻게 disconnect를 감지하는가

<https://stackoverflow.com/questions/15102992/what-is-the-difference-between-stdin-and-stdin-filen>

STDIN_FILENO 과 stdin의 차이

https://linux.die.net/man/3/inet_aton

inet_addr이 -1을 반환하는 문제

http://forums.devshed.com/programming-42/inet_pton-inet_aton-438155.html

<http://scratchnotesbymia.blogspot.kr/2009/08/inetntoa-vs-inetntop.html>

http://man7.org/linux/man-pages/man3/inet_pton.3.html

inet_aton vs inet_pton

이 둘을 따로 비교한 공식 문서를 찾지 못해 부득이하게 두 링크는 포럼과 블로그 글로 대체함.