

System programming

2차 과제 - System Programming Assignment #1-3 (proxy server)



학 과 : 컴퓨터 공학과

담당교수 : 황호영

분 반 : 목34

학 번 : 2016722092

성 명 : 정동호

제출날짜 : 2018-04-13

목차

1 INTRODUCTION.....	3
2 FLOWCHART.....	3
3 PSEUDO CODE	7
4 결과화면.....	14
5 결론 및 고찰	17
6 참고 레퍼런스.....	18

1 Introduction

현재 만드는 프록시 서버는 여러 클라이언트로부터의 요청을 처리할 수 있어야 한다. 아직 서버/클라이언트는 구현하지 않고 대신, 멀티프로세싱을 구현한다.

2 Flowchart

1-2에서도 그랬지만 1-3은 굉장히 많은 부분이 1-2와 겹쳤다. 그래서 실습 조교님께 여쭙보았고 중복되는 부분은 생략하는 대신 복사-붙여넣기 하라는 답변을 받았다. 따라서 이번에 새로 추가되거나 변경된 함수를 제외하곤 상당 부분 이전 보고서와 중복된 내용이 있을 수 있다.

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

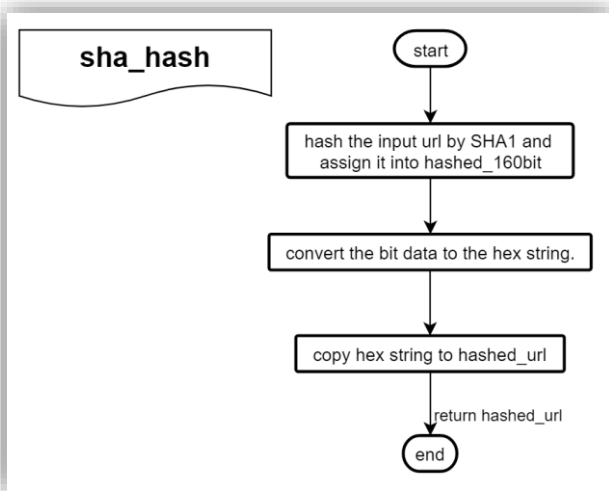


Figure 2.2 sha_hash function flowchart

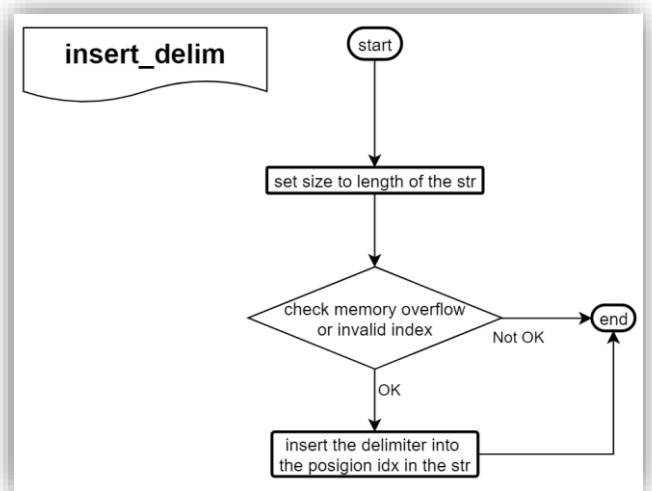


Figure 2.1 insert_delim function flowchart

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

Figure 2.2의 `insert_delim` 함수는 인자로 전달된 문자열 포인터에 `idx`번째 위치에 문자 `delim`을 삽입하여 그 값을 반환한다.

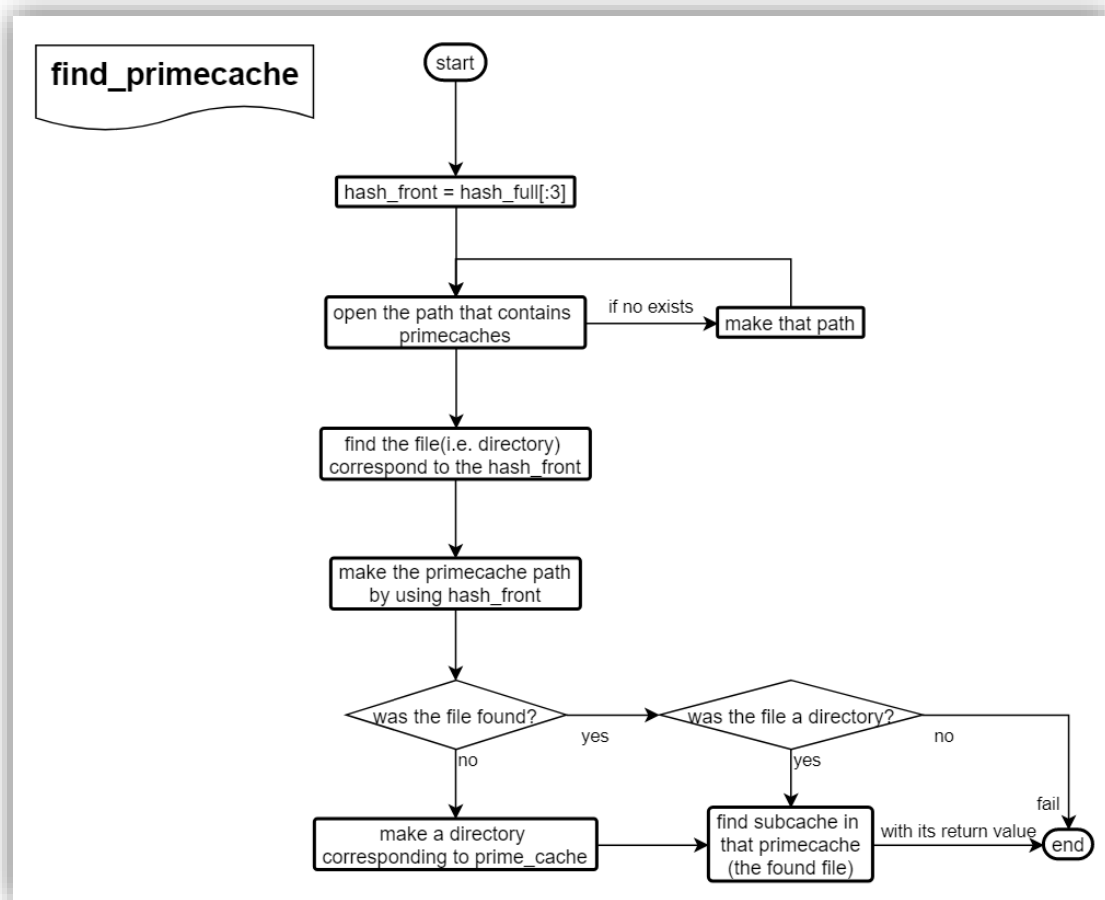


Figure 2.4 find_primecache function flowchart

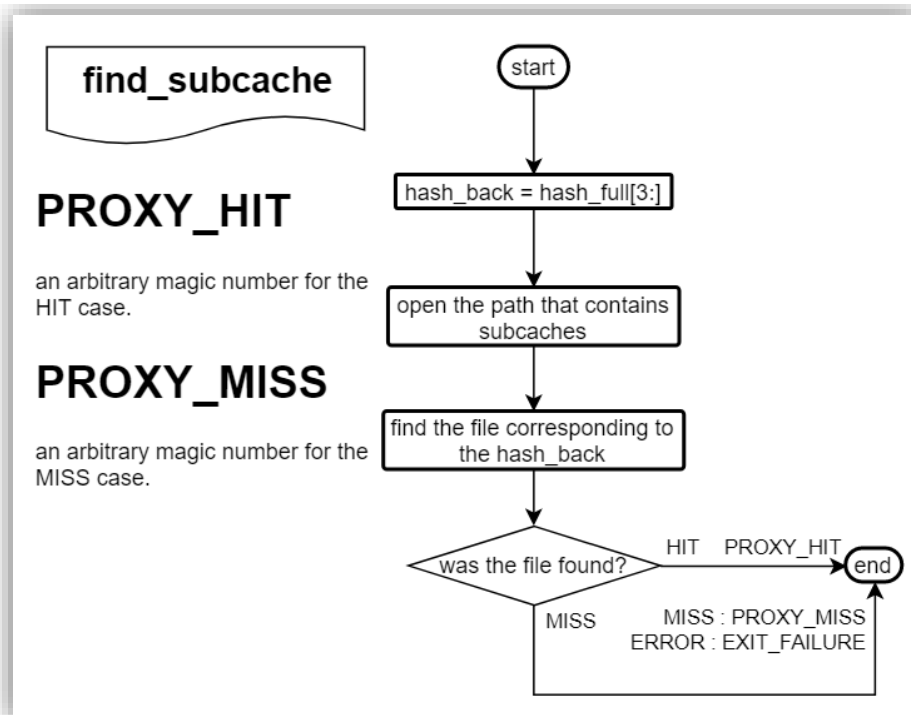


Figure 2.3 find_subcache function flowchart

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.

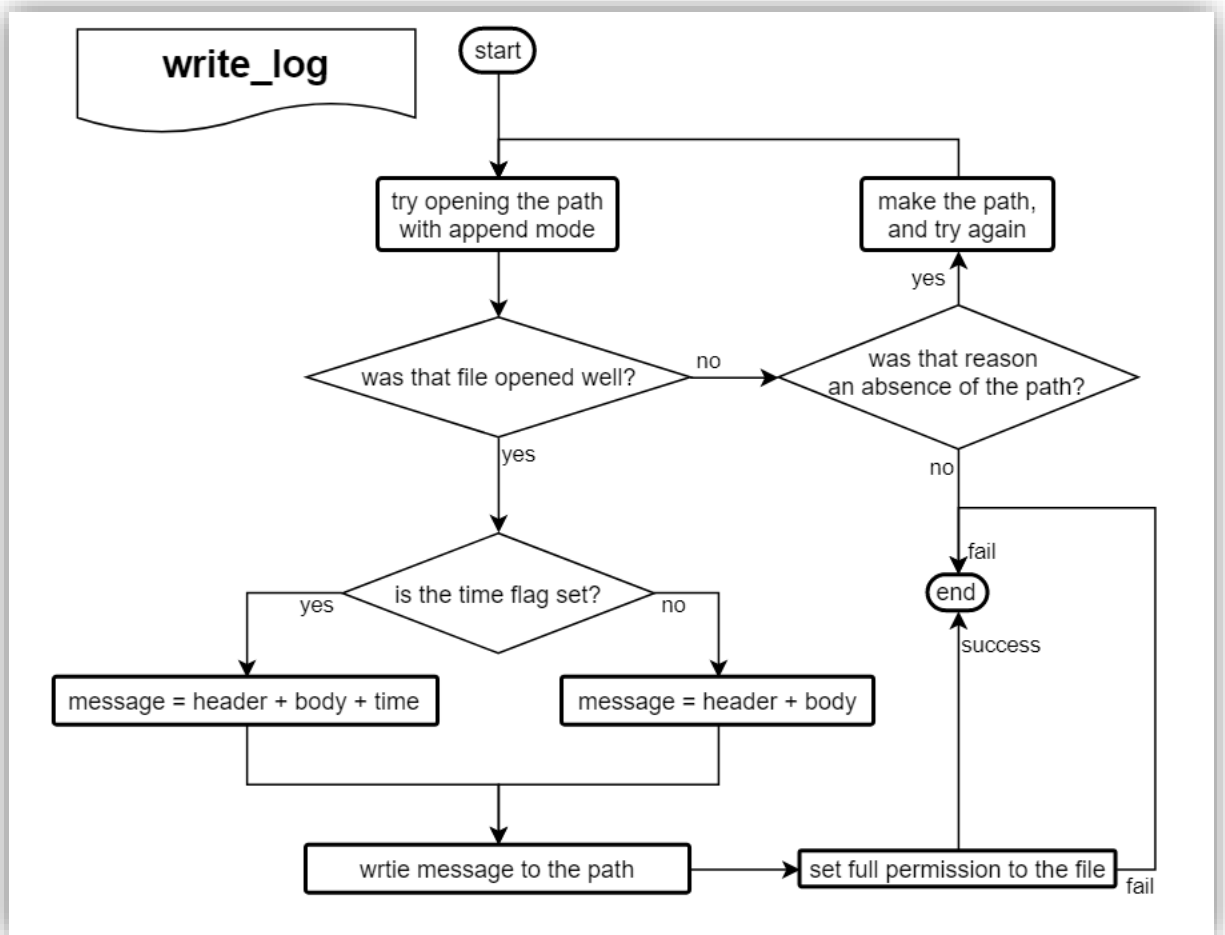


Figure 2.5 write_log function flowchart

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.

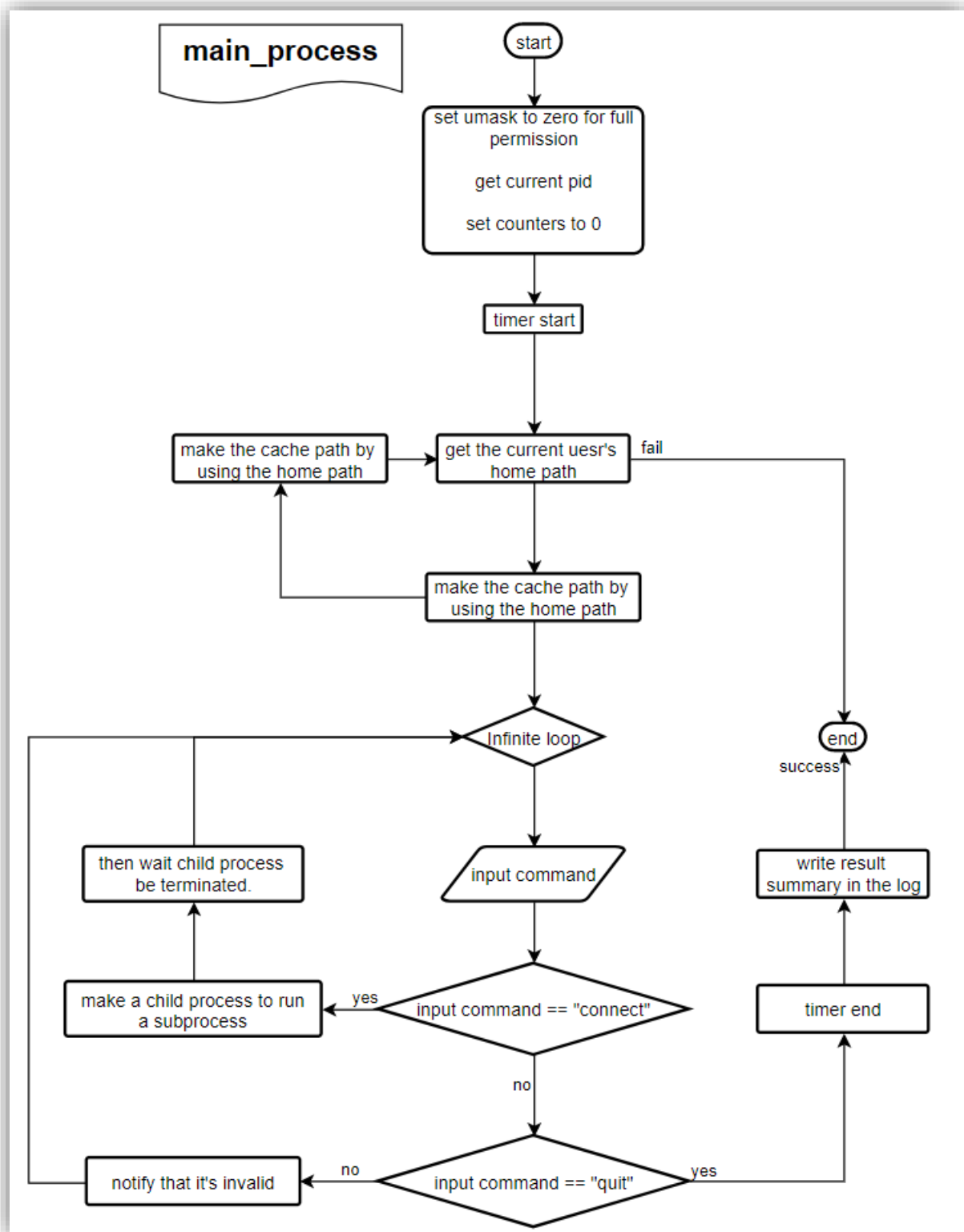


Figure 2.6 main_process function flowchart

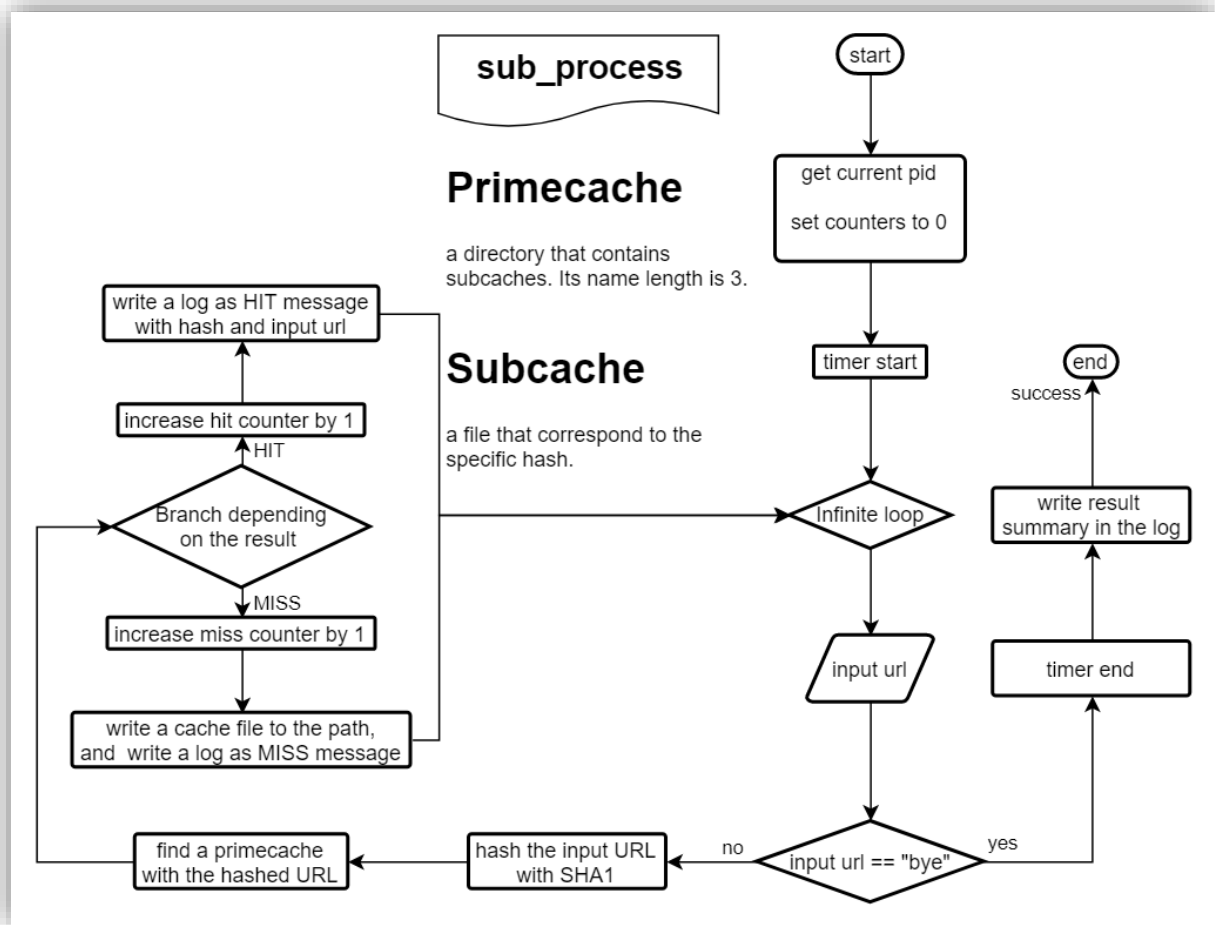


Figure 2.7 sub_process function flowchart

일단 main 은 Figure 2.6 의 main_process 를 호출하는 역할만 하므로 이번 보고서에는 생략하였다. main_process 는 사용자의 입력을 처리하며 connect 일 경우 자식 프로세스를 fork 하여 Figure 2.7 의 sub_process 를 수행하도록 한다. sub_process 는 이전과제의 main 과 상당 부분 유사한 역할을 맡고 있으며 URL 을 유저로부터 입력받아 캐시를 만들고 로그를 기록하는 역할을 한다.

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.

※ 전체적인 변경점 : 유저로부터의 입력을 scanf 대신 fgets 를 사용하도록 하였다. 혹시 모를 버퍼오버플로의 위험성을 줄이고자 하였다.

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.
* @return [int] Success:EXIT_SUCCESS, Fail:EXIT_FAILURE
int write_log(const char *path, const char *header, const char *body, bool time_){
    // 32 is a moderately large value to save time information
    char time_str[32] = {0};

    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

If time flag is true,
then time_str has format as "-[%Y/%m/%d, %T]"
※ time information is set to current local time

Join all parts of the message together

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 는 거의 바뀐게 없다. 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_ 은 메시지 뒤에 시간 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.

보면 time_str 이 32 크기인데 - dash 나 [] square brackets 또는 공백 쉼표 같은 것들과 연월일 시분초 정보를 수용할 수 있는 적당한 크기라 생각해서 결정했다.

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 도 바뀌게 없다.

먼저 인자로 받은 전체에서 primecache(해시 앞부분)을 추출한다. 그리고 인자로 받은 경로를 순회하며 해당하는 primecache를 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_FAILURE를 반환한다.

파일을 못 찾았으면 해당 primecache로 시작하는 캐시가 아직 생성되지 않은 것이므로 이를 생성하고 find_subcache를 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 함수도 바뀌게 없다.

그리고 primecache 와 했던것과 유사하게 인자로 전달받은 경로를 순회하며 subcache 들을 탐색하고 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.
* @return [int] Success:EXIT_SUCCESS
int sub_process(const char *path_log, const char *path_cache){

    Get pid of current process and set current_pid to it

    // pid number for current process
    pid_t current_pid = getpid();

    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'{
        printf("[%d]input URL> ", current_pid);
    }
}

```

```

    Input URL safely to avoid the buffer overflow

    If 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
            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
            break;

        default:
            break;
    }
}

Timer end

Make a string for terminating the log and write it

return EXIT_SUCCESS;
}

```

sub_process 는 새로추가된 함수지만 상당 부분이 이전 과제의 main 과 유사하다. 좀 달라진 점이라면 홈 경로를 구하고 로그와 캐시 경로를 만드는 과정이 상위 함수인 main_process 에 옮겨졌다는 것이다. 그 이유는 아래 main_process 에서 언급한다.

해당 함수의 역할은 유저로부터 URL 을 입력받아 SHA1 으로 해싱하고 앞 세자리를 1 차 캐시, 나머지 뒷 자리를 2 차캐시(본 프로그램에서는 primecache, subcache 로 명명)로 나눠 MISS 나 HIT 나에따라 적절한 처리를 수행하는 것이다.

Figure 3.6 main_process function pseudo code

```

* @return [int] Success:EXIT_SUCCESS, Fail:EXIT_FAILURE
int main_process(){
    Get pid of current process and set current_pid to it

```

```

// pid numbers of processes
pid_t current_pid = getpid();

Set full permission for the current process.

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

Receive inputs till the input is 'quit'{
    printf("[%d]input CMD> ", current_pid);
    Input URL safely to avoid the buffer overflow

    If input is 'connect'{
        Make a child process{
            If it goes fail then notify it and continue
            In a child process, start a sub_process
            In a original process, increase subprocess conter by 1 and wait
        }
    } else if input is 'quit' then escape loop
    else then notify that it's invalid input
}

Timer end

Make a string for terminating the log and write it

return EXIT_SUCCESS;
}

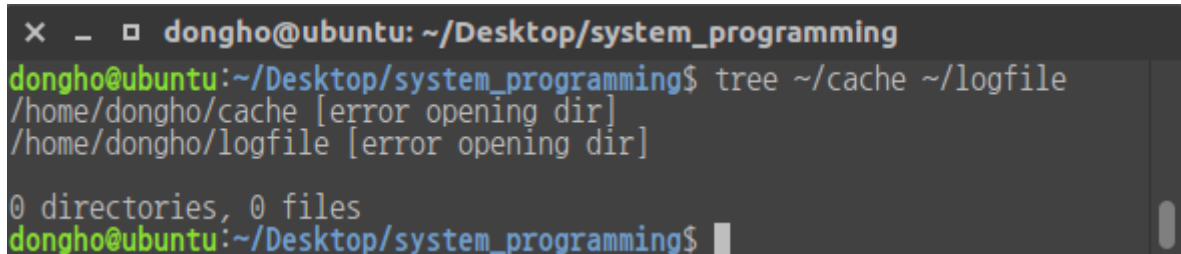
```

main_process 는 이번 과제에서 온전히 새로 추가된 함수이다. 이번에 새로 배운 fork() 함수를 사용하여 이전 과제의 main 인 sub_process 를 수행하는 자식 프로세스를 생성하는 역할이다. 전체적인 루틴은 sub_process 와 유사하며 connect 또는 quit 을 명령어로 받는다. connect 면 자식 프로세스를 생성하고 기다린다. 자식 프로세스가 할 일은 위 sub_process 에서 설명했다.

그리고 기존 main(현재 sub_process)의 역할이었던 캐시&로그 경로 생성은 여기 main_process 로 옮겨졌다. 왜냐하면 main_process 도 종료시에 로그에 경과 시간과 서버 프로세스 갯수등의 정보등을 기록해야 하기 때문이다. 아무래도 부모 -> 자식 관계에서 부모가 자식의 정보에 의존하여 코드를 수행하는 것은 구조상 좋지 않아보였다.

sub_process 와 다르게 main_process 는 정해진 명령어만을 입력받으며 그 외에 값이 들어오면 유효하지 않은 명령어임을 알리도록 했다.

4 결과화면

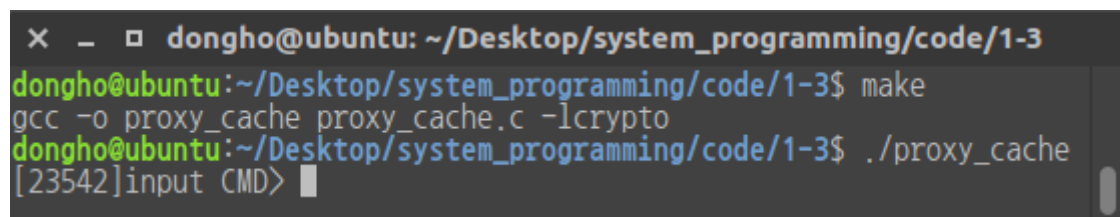


```
× _ □ dongho@ubuntu: ~/Desktop/system_programming
dongho@ubuntu:~/Desktop/system_programming$ tree ~/cache ~/logfile
/home/dongho/cache [error opening dir]
/home/dongho/logfile [error opening dir]

0 directories, 0 files
dongho@ubuntu:~/Desktop/system_programming$
```

Figure 4.1 init state

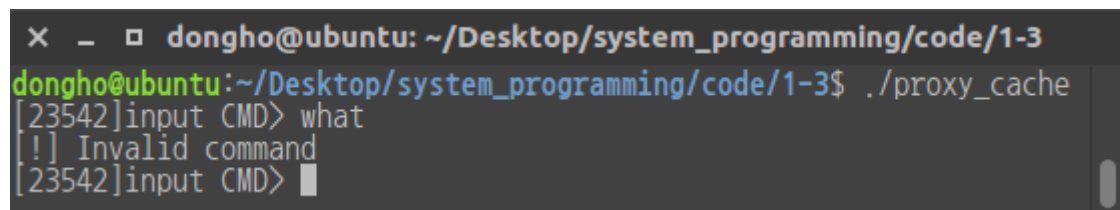
처음에 두 디렉터리 ~/logfile 과 ~/cache를 tree로 조회해 본 결과이다. 아직 아무것도 수행하지 않았으므로 아무것도 없다.



```
× _ □ dongho@ubuntu: ~/Desktop/system_programming/code/1-3
dongho@ubuntu:~/Desktop/system_programming/code/1-3$ make
gcc -o proxy_cache proxy_cache.c -lcrypto
dongho@ubuntu:~/Desktop/system_programming/code/1-3$ ./proxy_cache
[23542]input CMD>
```

Figure 4.2 init screen

make로 컴파일을 하고 실행시킨 뒤 화면이다. 이전 과제들과 달리 URL 대신 CMD(명령어)를 입력받으며 대괄호로 현재 PID를 출력하는 것을 볼 수 있다.



```
× _ □ dongho@ubuntu: ~/Desktop/system_programming/code/1-3
dongho@ubuntu:~/Desktop/system_programming/code/1-3$ ./proxy_cache
[23542]input CMD> what
[!] Invalid command
[23542]input CMD>
```

Figure 4.3 invalid command

유효하지 않은 명령어('connect' | 'quit' 외)를 입력해본 결과다.

```
× - □ dongho@ubuntu: ~/Desktop/system_programming/code/1-3
[!] Invalid command
[23542]input CMD> connect
[23590]input URL> www.naver.com
[23590]input URL>

× - □ dongho@ubuntu: ~/Desktop/system_programming
dongho@ubuntu:~/Desktop/system_programming$ tree ~/cache ~/logfile
/home/dongho/cache
├── fed
│   └── 818da7395e30442b1dcf45c9b6669d1c0ff6b
/home/dongho/logfile
└── logfile.txt

1 directory, 2 files
dongho@ubuntu:~/Desktop/system_programming$ cat ~/logfile/logfile.txt
[Miss]www.naver.com-[2018/04/13, 01:46:15]
dongho@ubuntu:~/Desktop/system_programming$
```

Figure 4.5 input url 1

connect를 사용해 자식 프로세스를 생성한뒤 sub_process에 진입하여 URL을 입력받았다.
cache 파일과 log파일 모두 잘 생성된 것을 확인하였다.

```
× - □ dongho@ubuntu: ~/Desktop/system_programming/code/1-3
[23590]input URL> www.naver.com
[23590]input URL> www.kw.ac.kr
[23590]input URL> www.naver.com
[23590]input URL>

× - □ dongho@ubuntu: ~/Desktop/system_programming
dongho@ubuntu:~/Desktop/system_programming$ tree ~/cache ~/logfile
/home/dongho/cache
├── e00
│   └── 0f293fe62e97369e4b716bb3e78fababf8f90
├── fed
│   └── 818da7395e30442b1dcf45c9b6669d1c0ff6b
/home/dongho/logfile
└── logfile.txt

2 directories, 3 files
dongho@ubuntu:~/Desktop/system_programming$ cat ~/logfile/logfile.txt
[Miss]www.naver.com-[2018/04/13, 01:46:15]
[Miss]www.kw.ac.kr-[2018/04/13, 01:49:11]
[Hit]fed/818da7395e30442b1dcf45c9b6669d1c0ff6b-[2018/04/13, 01:49:13]
[Hit]www.naver.com
dongho@ubuntu:~/Desktop/system_programming$
```

Figure 4.4 input URL 2

이번엔 HIT와 MISS 둘다 테스트 해보았다. 로그가 잘 찍혔음을 확인하였다.

```
× - □ dongho@ubuntu: ~/Desktop/system_programming/code/1-3
[23590]input URL> www.kw.ac.kr
[23590]input URL> www.naver.com
[23590]input URL> bye
[23542]input CMD>

× - □ dongho@ubuntu: ~/Desktop/system_programming
dongho@ubuntu:~/Desktop/system_programming$ cat ~/logfile/logfile.txt
[Miss]www.naver.com-[2018/04/13, 01:46:15]
[Miss]www.kw.ac.kr-[2018/04/13, 01:49:11]
[Hit]fed/818da7395e30442b1dcf45c9b6669d1c0ff6b-[2018/04/13, 01:49:13]
[Hit]www.naver.com
[Terminated] run time: 267 sec. #request hit : 1, miss : 2
dongho@ubuntu:~/Desktop/system_programming$
```

Figure 4.6 bye

첫번째 서브프로세스를 종료한 결과이다. 해당 프로세스의 수행시간과 간단한 HIT/MISS 요약 통계 정보가 기록되었음을 확인하였다.

```
× - □ dongho@ubuntu: ~/Desktop/system_programming/code/1-3
[23542]input CMD> connect
[26919]input URL> www.google.com
[26919]input URL> www.kw.ac.kr
[26919]input URL> www.helloworld.org
[26919]input URL> bye
[23542]input CMD>

× - □ dongho@ubuntu: ~/Desktop/system_programming
dongho@ubuntu:~/Desktop/system_programming$ cat ~/logfile/logfile.txt
[Miss]www.naver.com-[2018/04/13, 01:46:15]
[Miss]www.kw.ac.kr-[2018/04/13, 01:49:11]
[Hit]fed/818da7395e30442b1dcf45c9b6669d1c0ff6b-[2018/04/13, 01:49:13]
[Hit]www.naver.com
[Terminated] run time: 267 sec. #request hit : 1, miss : 2
[Miss]www.google.com-[2018/04/13, 21:01:54]
[Hit]e00/0f293fe62e97369e4b716bb3e78fababf8f90-[2018/04/13, 21:01:57]
[Hit]www.kw.ac.kr
[Miss]www.helloworld.org-[2018/04/13, 21:02:23]
[Terminated] run time: 38 sec. #request hit : 1, miss : 2
dongho@ubuntu:~/Desktop/system_programming$
```

Figure 4.7 repeat 4-6

다른 프로세스에서도 똑같이 해보았고 로그가 한곳에 잘 기록되는지 확인 하였다.


```
× - □ dongho@ubuntu: ~/Desktop/system_programming/code/1-3
[26919]input URL> www.google.com
[26919]input URL> www.kw.ac.kr
[26919]input URL> www.helloworld.org
[26919]input URL> bye
[23542]input CMD> quit
dongho@ubuntu:~/Desktop/system_programming/code/1-3$

× - □ dongho@ubuntu: ~/Desktop/system_programming
dongho@ubuntu:~/Desktop/system_programming$ cat ~/logfile/logfile.txt
[Miss]www.naver.com-[2018/04/13, 01:46:15]
[Miss]www.kw.ac.kr-[2018/04/13, 01:49:11]
[Hit]fed/818da7395e30442b1dcf45c9b6669d1c0ff6b-[2018/04/13, 01:49:13]
[Hit]www.naver.com
[Terminated] run time: 267 sec. #request hit : 1, miss : 2
[Miss]www.google.com-[2018/04/13, 21:01:54]
[Hit]e00/0f293fe62e97369e4b716bb3e78fababf8f90-[2018/04/13, 21:01:57]
[Hit]www.kw.ac.kr
[Miss]www.helloworld.org-[2018/04/13, 21:02:23]
[Terminated] run time: 38 sec. #request hit : 1, miss : 2
**SERVER** [Terminated] run time: 69751 sec. #sub process: 2
dongho@ubuntu:~/Desktop/system_programming$
```

Figure 4.8 quit

마지막으로 메인프로세스에서 quit을 입력해 종료하였다. run time을 보면 뭔가 오류가 있는것 같이 보일 수 있지만 사실 과제 결과화면을 찍는 도중 좀 길게 자버려서 전체 수행시간이 위와 같이 나왔다.

5 결론 및 고찰

뭔가 매주 조금씩 하나하나 구현해가며 실습을 진행하고 있는데 점점 프록시 서버 다운 프로그램이 만들어지고 있는것 같다. 사실 진짜 프록시서버라 부르려면 많은 작업들이 남아있겠지만 그것들은 이제 남은 과제들에서 구현할것이라 생각한다.

그건 그렇고 이번에는 멀티프로세싱을 구현했다. unix의 fork라는 시스템콜을 이용해 이를 구현할 수 있었는데 작동방식이 참 신기했다. 근데 왜 이름이 fork인가 검색해봤는데 fork에는 분기점/갈라지다 라는 뜻도 있어서 인것 같다.

과제를 하다가 메모리 문제가 난적이 있었는데 gdb로 디버깅을 하던중 막힌 적이 있었다. 바로 fork를 통해 분기를 하고 나면 gdb가 어떤 프로세스에서 디버깅을 수행할지 막막했

기 때문이었다. 그런데 구글링을 해보니 gdb에서 set follow-fork-mode child를 통해 이를 명시할 수 있음을 알게되어 요긴하게 사용하였다.

마지막으로 해당 과제를 수행하며 애매하게 알고 있었던 Zombie 프로세스와 Orphan 프로세스의 개념을 정리할 수 있었다. Zombie는 프로세스는 죽었지만 부모로부터 wait을 호출 받지 못해 자원을 반납하지 못한 경우, Orphan은 자식이 종료되기 전에 부모가 먼저 종료되어버린 경우이다. 이번 과제를 하기전엔 솔직히 많이 헷갈렸을것 같다.

6 참고 레퍼런스

<http://endic.naver.com/enkrEntry.nhn?sLn=kr&entryId=4953fa7ed20e4c5c99edf4cae21b4344&query=fork>

fork의 뜻

<https://stackoverflow.com/questions/6199270/how-do-i-debug-the-child-process-after-fork-in-gdb>

gdb에서 fork 처리

강의 자료실의 [18-1 SPLab week05 Proxy 1-3.pdf](#)

특히 이번 과제는 많은 부분을 강의 자료를 통해 학습할 수 있었다. Zombie vs Orphan 프로세스, fork 사용법 등등

<https://stackoverflow.com/questions/20688982/zombie-process-vs-orphan-process>

Zombie vs Orphan 프로세스 추가 설명