

System programming

Assignment #3-1 (proxy server)



학 과 : 컴퓨터 공학과

담당교수 : 황호영

분 반 : 목34

학 번 : 2016722092

성 명 : 정동호

제출날짜 : 2018-06-01

목차

1 INTRODUCTION.....	3
2 FLOWCHART.....	3
3 PSEUDO CODE	10
4 결과화면	18
5 결론 및 고찰	19
6 참고 레퍼런스.....	19

1 Introduction

지금껏 프록시 서버의 기본적인 요구사항과 Origin서버에서 Timeout이 일어나는 상황까지 고려하여 구현하였다. 이번 과제에서는 두개 이상의 프로세스가 동시에 로그파일에 접근하는 상황을 통제하고 제어하기 위해 세마포어를 사용하여 Critical section을 구현하고 한번에 하나의 접근만을 허용하도록 한다.

2 Flowchart

3-1 과제에서 변한 부분이 정말 너무 없다. 이는 나뿐만 아니라 다른 동기들도 비슷할거라 생각한다. 변한 부분은 write_log 한군데 뿐이며 세부 내용은 해당 파트에서 설명한다.

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

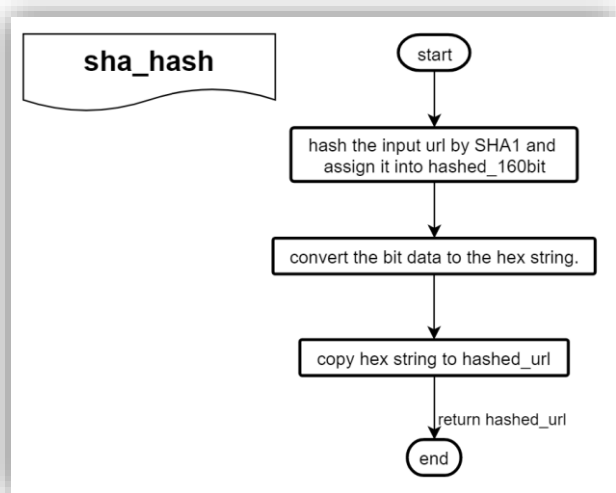


Figure 2.1 sha_hash function flowchart

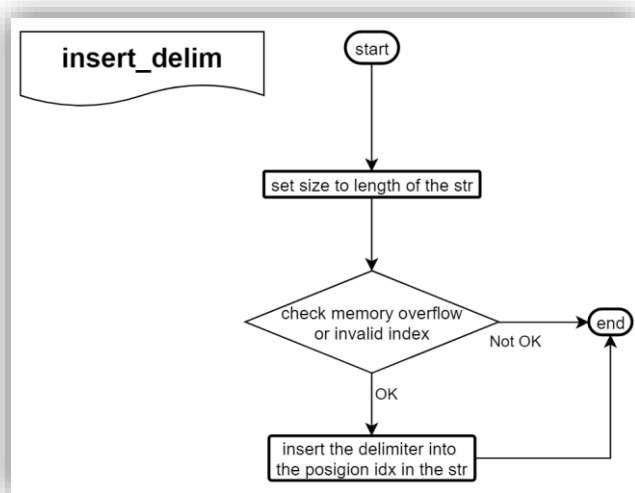


Figure 2.2 insert_delim function flowchart

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.

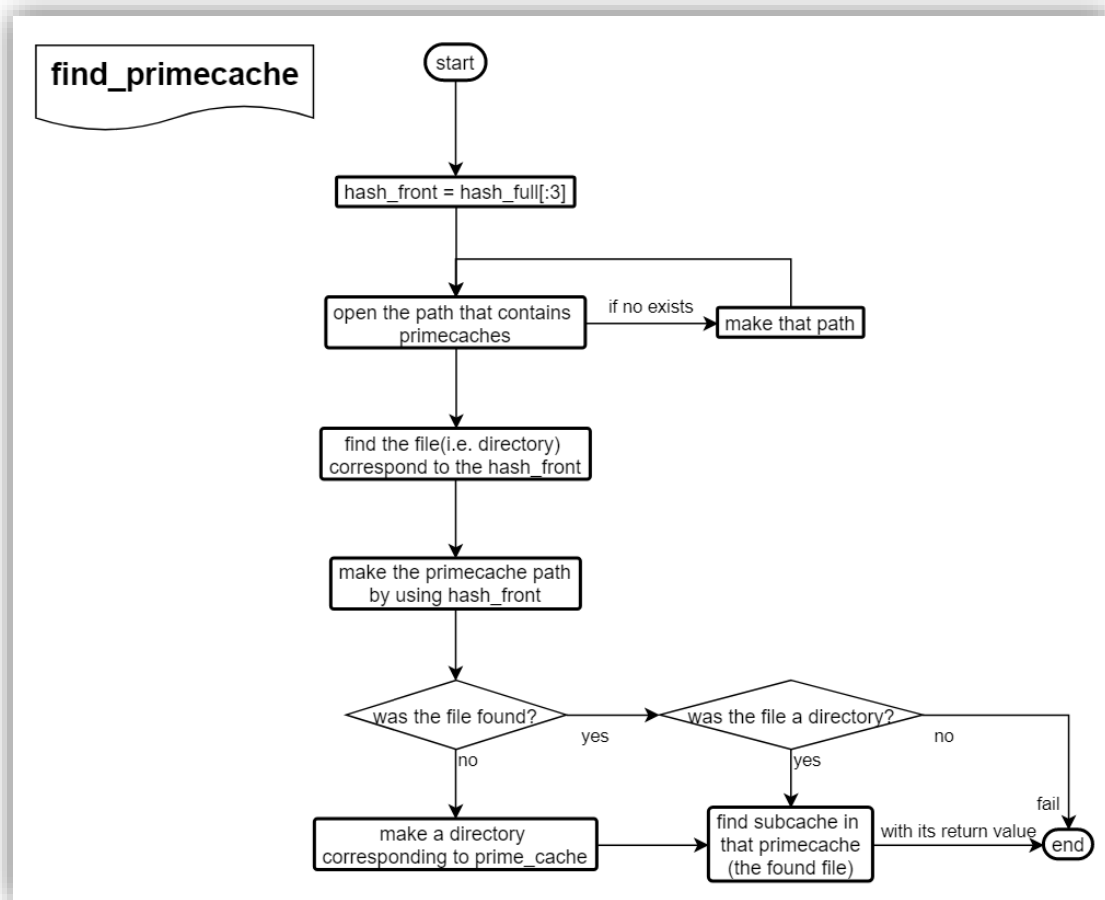


Figure 2.4 find_primecache function flowchart

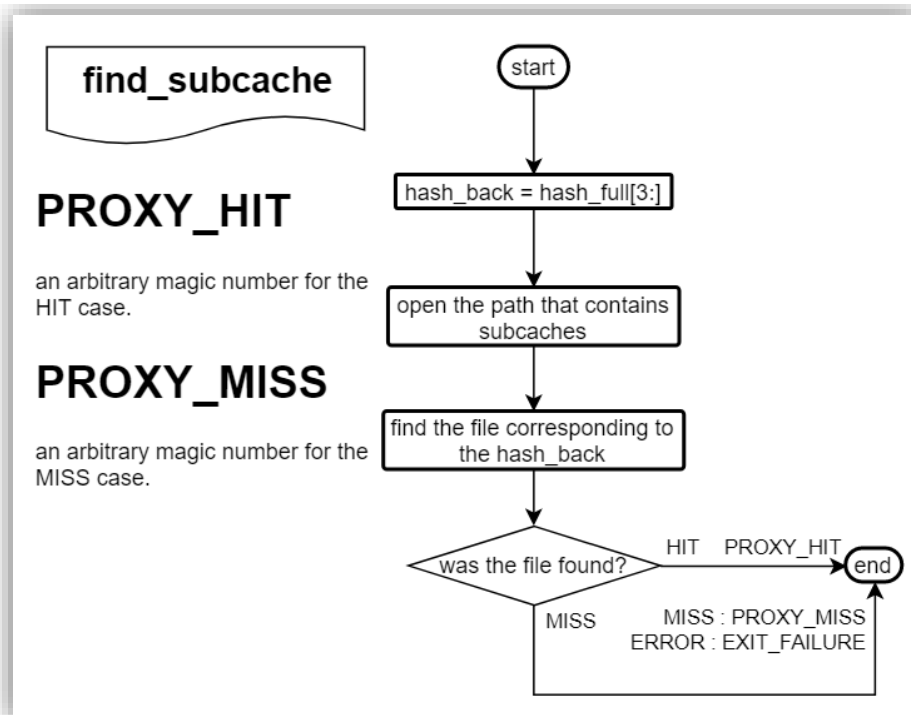


Figure 2.3 find_subcache function flowchart

Figure 2.3의 find_primecache는 해시의 앞부분(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.

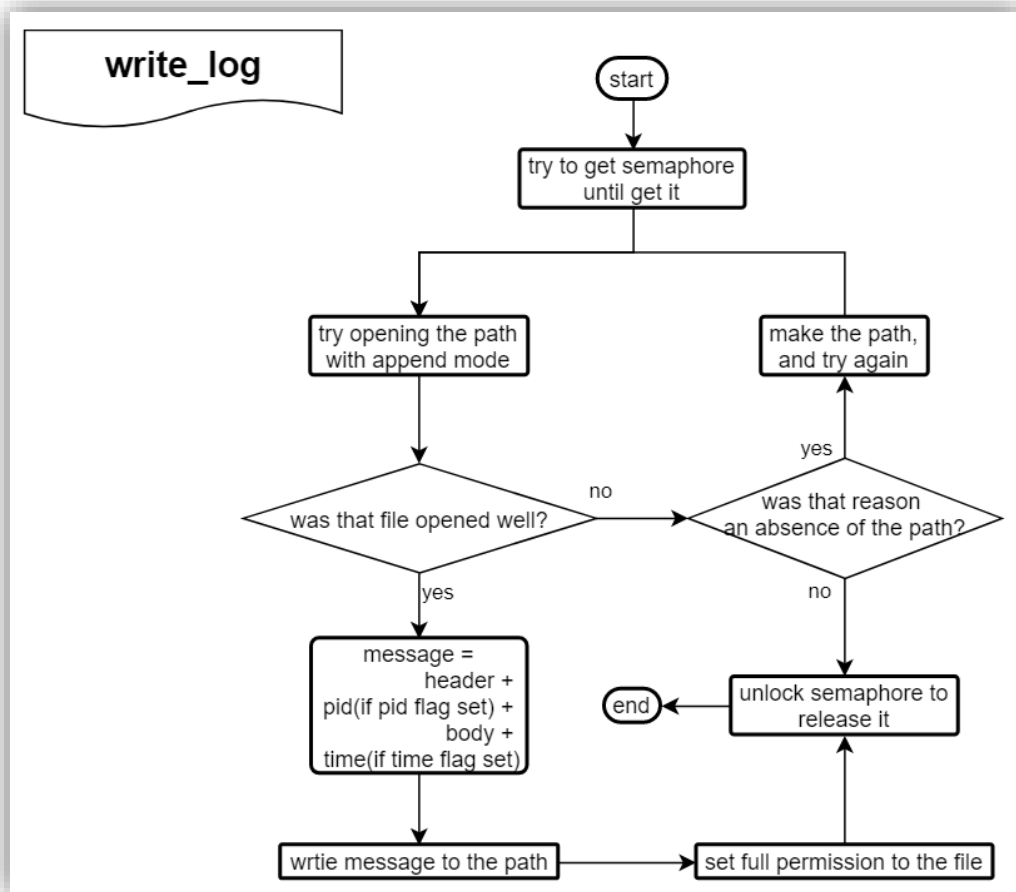


Figure 2.5 write_log function flowchart

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.

추가로 동시에 여러프로세스가 로깅하는것을 막기 위해 처음에 세마포어를 얻는 부분이 추가 되었고 마지막에 로그가 끝나고 이를 반환하는 부분도 추가 되었다.

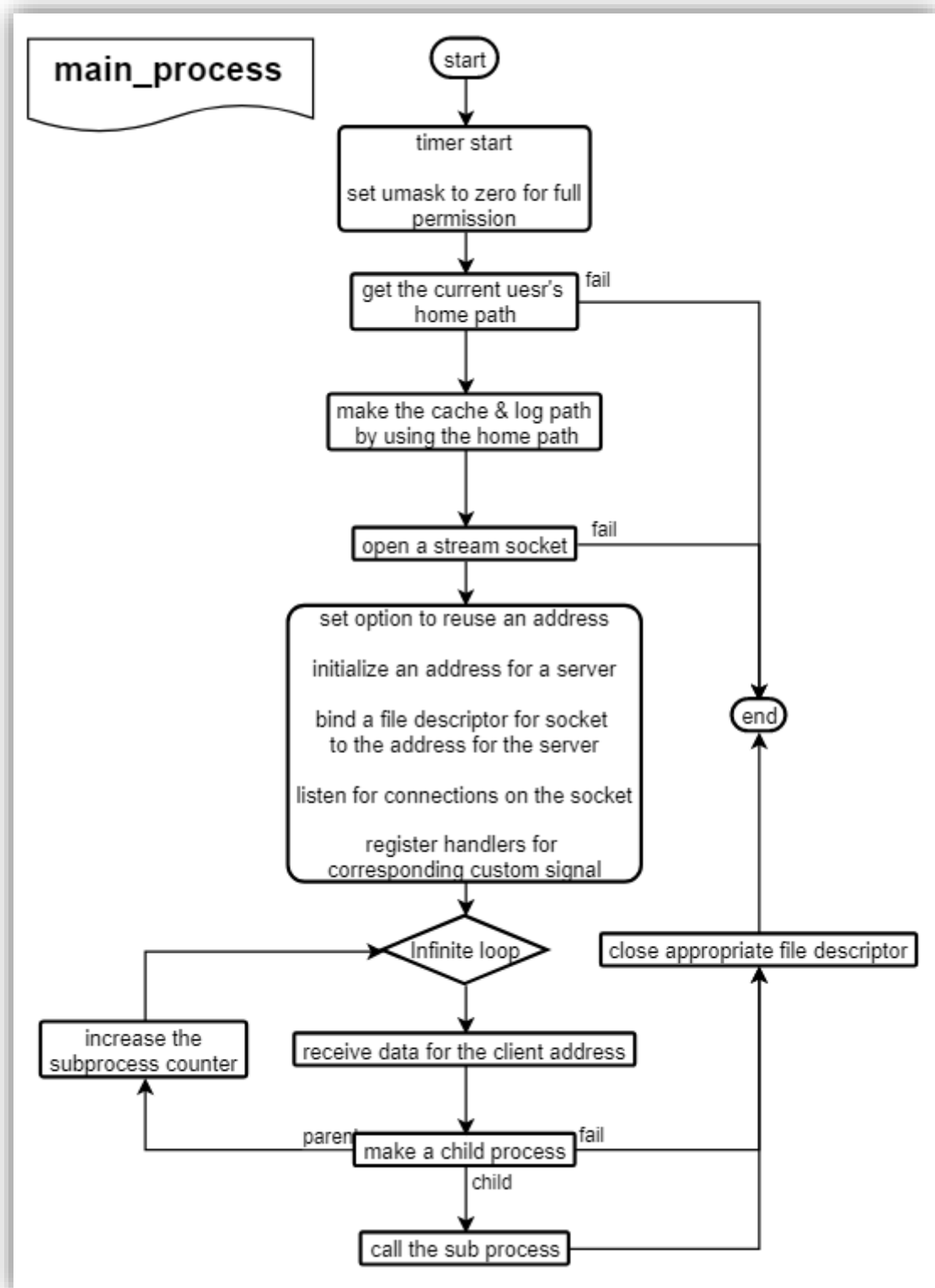


Figure 2.6 main_process function flowchart

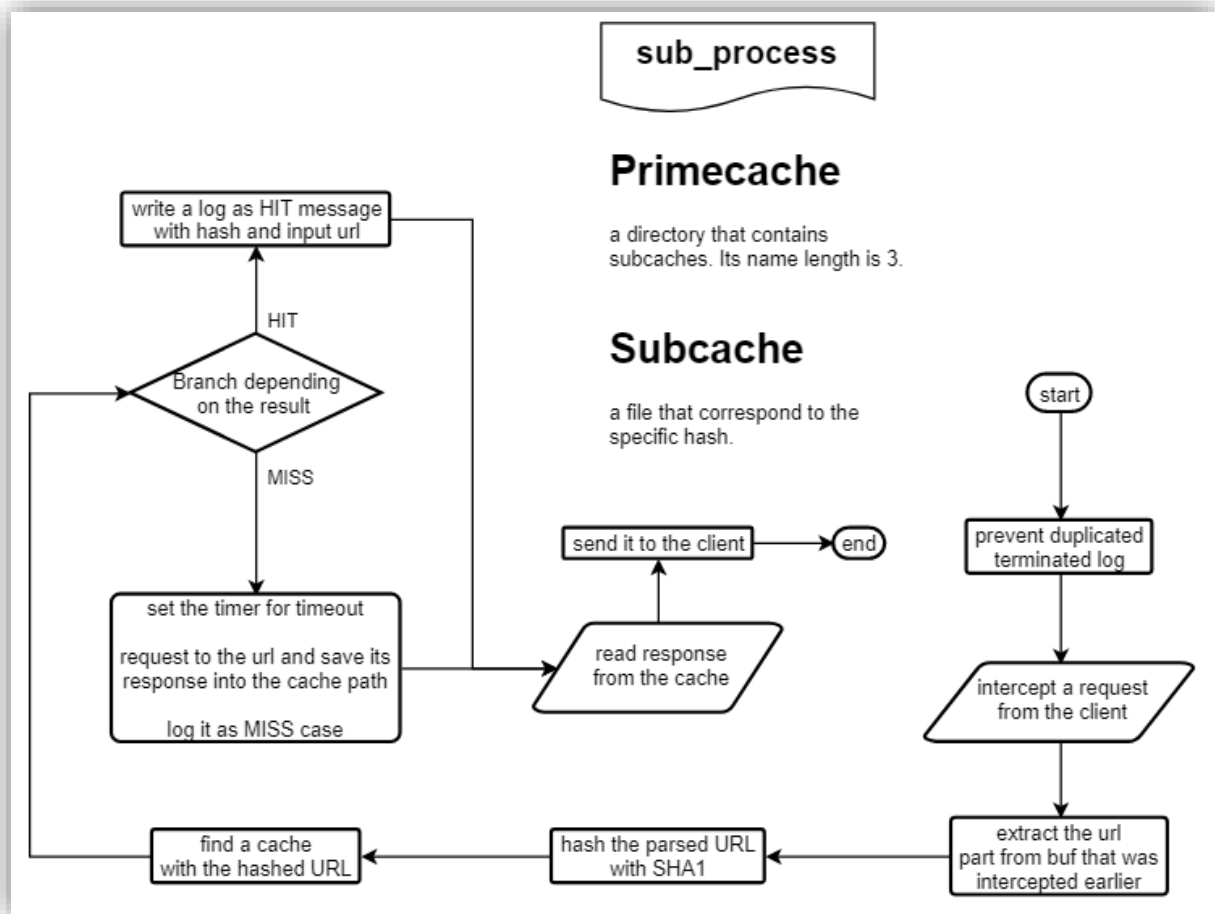


Figure 2.7 sub_process function flowchart

main 은 Figure 2.6 의 main_process 를 호출하는 역할만 하므로 이번 보고서에는 생략하였다. main_process 는 클라이언트 즉 브라우저로부터 들어오는 요청을 받아 fork 하여 자식프로세스에서 해당 요청을 처리하는 역할을 수행한다. 이전 과제와 달라진점은 과제 요구사항중 로그 기록 방식이 바뀌면서 타이머 설정을 sub_process 대신 main_process 에서 수행하는 것과 자식 프로세스를 만들때 프로세스 카운터를 증가시키고 SIGINT 의 핸들러를 등록하는 부분들이다. 타이머의 종료와 로그 기록등은 뒤에 나올 handler_int 에서 다룬다.

Figure 2.7 의 sub_process 는 가로챈 요청에서 url 부분을 뽑아 HIT/MISS 를 판별한다. HIT 일 경우 아무것도 하지않고, MISS 일 경우 10 초짜리 타이머를 설정하고 원래 대상 URL 에 요청을 보낸 뒤 그 값을 캐싱한다. 그리고 HIT/MISS 상관 없이 캐시 파일을 읽어 클라이언트에게 전달한다. HIT/MISS 일경우 모두 적절히 로그를 기록한다. 이전 과제와 비교해 실제 응답을 전달해준다는것과 요약 정보 로깅이 빠졌다는 차이가 있다.

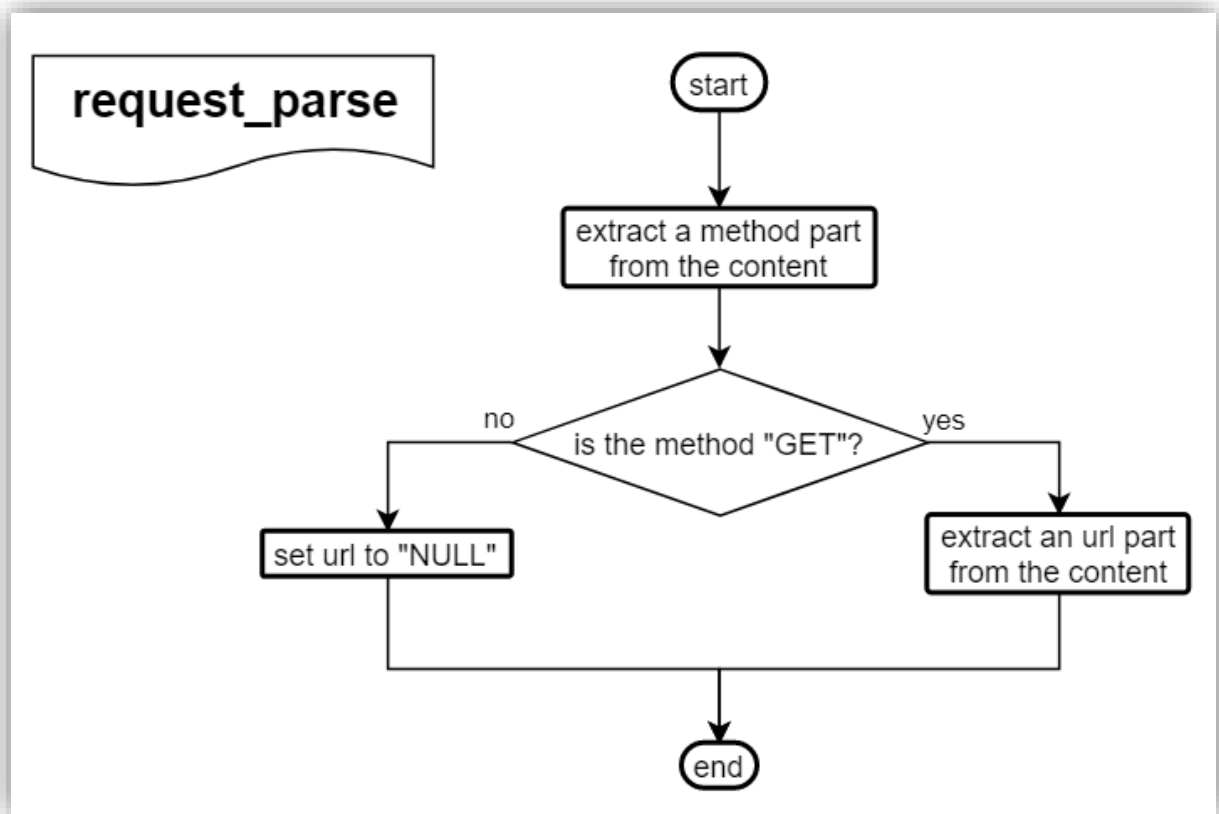


Figure 2.8 request_parse function flowchart

Figure 2.8의 request_parse 는 요청 헤더 정보가 담긴 버퍼를 받아 메서드를 추출하여 메서드가 GET 인 경우 url 을 파싱하고 이외에 경우에는 url 에 문자열 "NULL"을 넣는다.

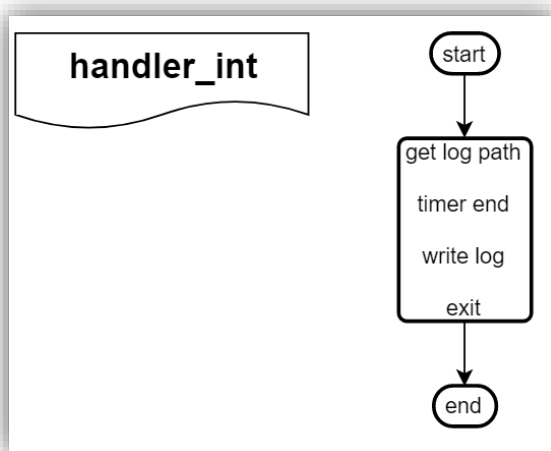


Figure 2.9 handler_int

Figure 2.9 의 handler_int 는 무척 단순하다. 이번 과제에서 아무리찾아봐도 종료조건이 보이지 않지만 main process 에서 종료시에 로그를 기록해야 했으므로 강의자료에 나온 SIGINT 를 활용하는것이라 판단했다.

프로그램 수행중 인터럽트가 발생하면 전역타이머와 카운터를 통해 적절한 로그를 기록하고 종료한다.

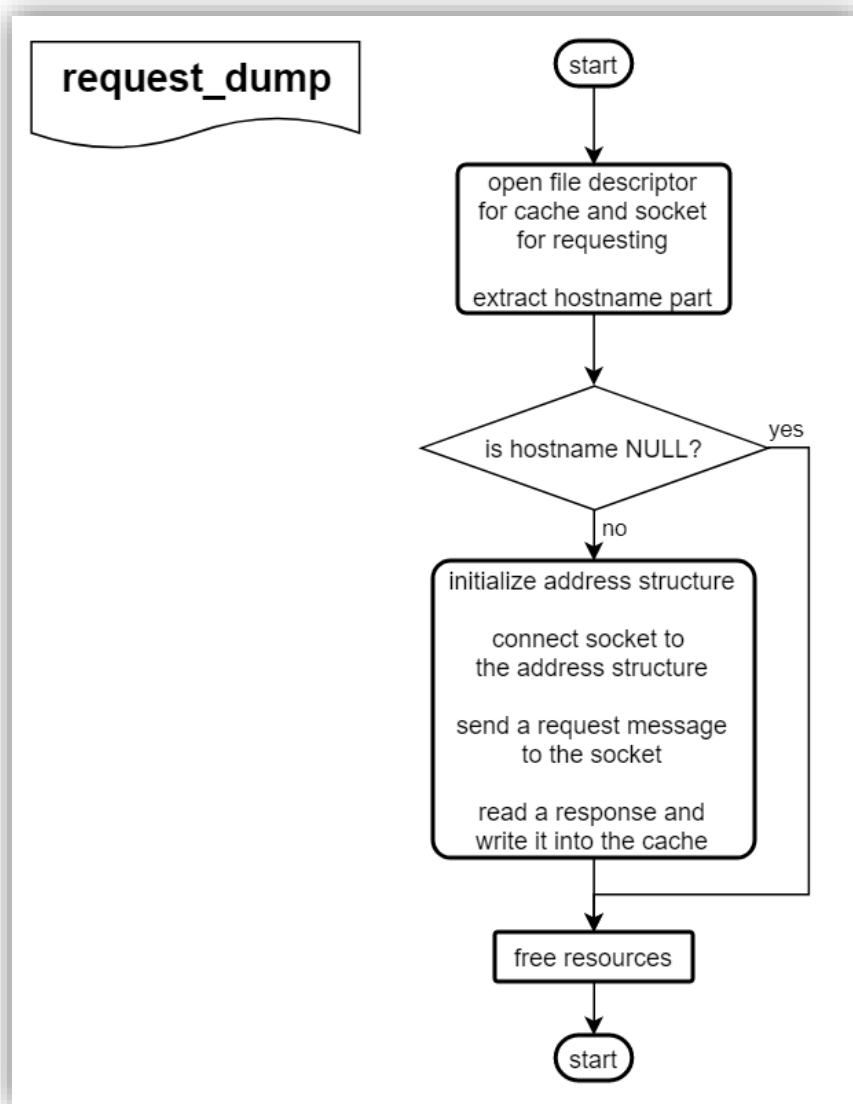


Figure 2.10 request_dump

Figure 2.10 의 request_dump 는 서버에 요청을 날리고 그 응답 결과를 파일에 저장하는 역할을 수행한다. 그러기 위해 먼저 파일 디스크립터들을 열고 요청할 서버의 hostname 을 추출한뒤 hostname 이 유효한 경우 작업을 수행한다. 마지막 검증단계 때 valgrind 로 테스트를 진행하다 가끔 hostname 에 null 이 뽑히는 문제가 발생하여 추가로 분기문을 넣었으나 없어도 가시적인 문제가 나타나진 않았다. 그래도 혹시몰라 넣었다.

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 to get semaphore until get it

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 {
        Unlock semaphore to release it
        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 then unlock semaphore to release it and return FAIL

return EXIT_SUCCESS;
}

```

write_log의 인자는 path, header, body, time_ 으로 이루어져 있고 이름과 자료형에서 알 수 있다시피 path는 로그를 작성할 위치, header와 body는 메시지의 앞과 뒷부분, time_flag와 pid_flag는 기록할 메시지에 특정 정보를 추가할지 말지 결정하는 플래그이다. 굳이 header와 body로 나눈 이유는 호출측에서 굳이 추가적인 문자열 연산 없이도 Literal string과 동적으로 변하는 문자열을 그대로 인자로 넘길 수 있게끔 하기 위해서였다. 그리고 이번 과제에서 비중은 많이 줄었지만 코드 전체 일관성을 위해 플래그를 _접미사 대신 _flag로 좀 더 명시적으로 변경하였다.

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

그리고 과제 요구사항에 맞춰 해당 파일의 접근권한을 777로 설정한다.

또한 로그파일의 동시접근으로 인해 충돌이 일어나는것을 방지하고자 세마포어를 사용하여 동시접근을 제어하였다.

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){

    Prevent duplicated terminated log

    Intercept a request from the client
```

```

Extract the url part from buf that was intercepted earlier

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:
        Write a log as hashed url and parsed url in log path
        Write the HIT message to a response message
        break;

    case PROXY_MISS:
        Set the timer for timeout

        Request to the url and save its response into fullcache path

        Write a log as hashed url and parsed url in log path
        Write the MISS message to a response message
        break;

    default:
        break;
}

Read from the cache
and send its content to the client

return EXIT_SUCCESS;
}

```

sub_process 는 클라이언트의 요청을 가로채서 해당 url 을 추출하여 HIT/MISS 를 판별한뒤 적절한 작업을 수행하고 캐시로부터 내용을 읽어 클라이언트에 전달한다. 이 때 처음 문장을 보면 중복된 종료로그를 방지하는 부분이 있는데 이는 동시에 여러요청을 받을때 인터럽트를 받으면 각각의 자식프로세스들의 handler_int 가 호출되며 종료 기록이 중복되는 부분을 방지하고자 한것이다. 그리고 result 스위치문 내 MISS 인 경우에 타이머를 설정하는 부분이 있는데 이는 request_dump 를 호출할 때 생길 수 있는 시간초과 문제를 해결하고자 한것이다. 마지막으로 HIT/MISS 결과에 상관없이 해당하는 캐시로부터 값을 읽어 클라이언트에 전달한다.

Figure 3.6 main_process function pseudo code

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

    Timer start

    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 and 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

    Call an appropriate handler when corresponding occurs

    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;
        }
        Increase subprocess counter by 1

        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. 이 때 자식프로세스가 생성될 때마다 카운터값을 증가시킨다.

Figure 3.7 request_parse function pseudo code

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

    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. 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.

Figure 3.8 getIPAddr function flowchart

```
* @param addr The URL of the host
* @return [char *] SUCCESS:statically allocated char array ,FAIL:NULL
char *getIPAddr(const char *addr){
    char *haddr = NULL;
    Get network host entry from the addr

    If the entry is not NULL {
        Set haddr to dotted decimal string of the addr
    }

    return haddr;
}
```

인자로 넘어온 hostname에서 dotted decimal string을 추출해 반환한다. 이 때 의문이 생긴게 동적으로 할당된 버퍼가 반환되는게 유효한건가 싶었는데 검색해보니 이는 statically allocated buffer기 때문에 괜찮다는걸 알게되었다. 물론 이후 호출에 의해 값이 변경될 수 있으니 멀티쓰레드 환경에선 유의해야 할것 같다.

Figure 3.9 handler_int function flowchart

```
void handler_int(){
    Get log path

    Timer end

    Make a string for terminating the log and write it

    Exit with success status code
}
```

handler_int는 프로그램에 인터럽트가 발생되면 호출되는 함수다. 메인프로세스 종료시 로그를 기록하기 위해 구현되었다. 로그 경로를 얻고, 타이머를 종료하고, 수행 시간과 자식 프로세스 카운터를 로깅한다. 이 때 시그널에서 외부 값을 사용하는 다양한 방법을 생각해보았는데 메세지 큐를 이용하거나 sigaction을 이용하거나 전역변수를 이용하는 방법이 있었다. 나는 편의상 마지막을 선택했으나 이후에 변경될 수 있다.

Figure 3.10 request_dump function flowchart

```
* @param buf A buffer containing response
* @param url A char array containing the url
* @param filepath A char array containing the filepath
* @return [int] SUCCESS:EXIT_SUCCESS, FAIL:EXIT_FAILURE
int request_dump(const char *buf, const char *url, const char *filepath){

    Get a file descriptor at filepath to write with full permission

    Open a socket as TCP

    Extract a hostname

    If the hostname is not NULL {

        Initialize an address structure

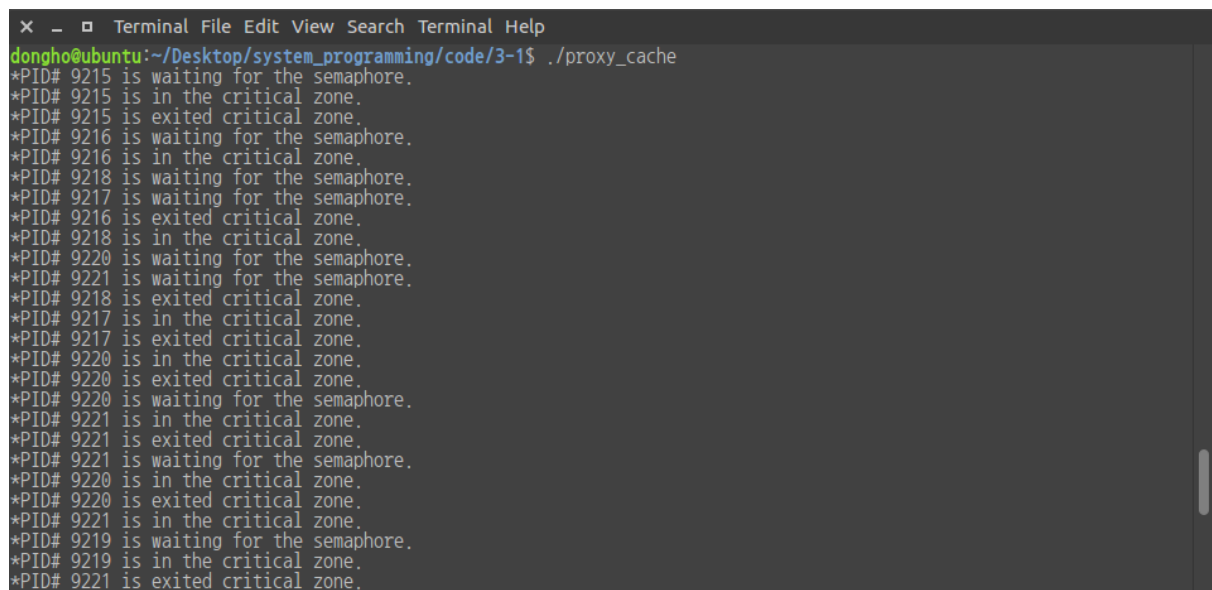
        Connect socket to the address structure

        Send the buf which is request message

        Read the response and write it into file descriptor for cache
    }
    Close file descriptors
    return EXIT_SUCCESS;
}
```

request_dump는 요청 정보가 담긴 buf와 실제 원본 파일이 있는 url 그리고 응답을 기록할 filepath를 인자로 받는다. url과 hostname부분에 다소 차이가 있어 따로 파싱과정을 중간에 거친 뒤 hostname이 유효하다면 요청정보를 해당 주소로 보낸뒤 그 응답값을 filepath에 기록한다.

4 결과화면

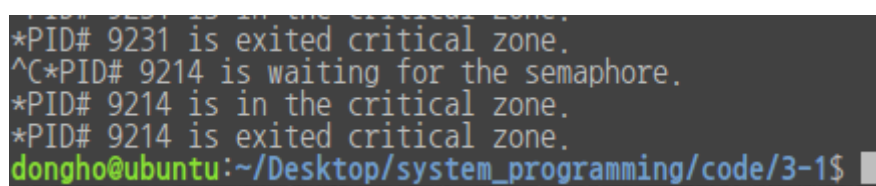


```
dongho@ubuntu:~/Desktop/system_programming/code/3-1$ ./proxy_cache
*PID# 9215 is waiting for the semaphore.
*PID# 9215 is in the critical zone.
*PID# 9215 is exited critical zone.
*PID# 9216 is waiting for the semaphore.
*PID# 9216 is in the critical zone.
*PID# 9218 is waiting for the semaphore.
*PID# 9217 is waiting for the semaphore.
*PID# 9216 is exited critical zone.
*PID# 9218 is in the critical zone.
*PID# 9220 is waiting for the semaphore.
*PID# 9221 is waiting for the semaphore.
*PID# 9218 is exited critical zone.
*PID# 9217 is in the critical zone.
*PID# 9217 is exited critical zone.
*PID# 9220 is in the critical zone.
*PID# 9220 is exited critical zone.
*PID# 9220 is waiting for the semaphore.
*PID# 9221 is in the critical zone.
*PID# 9221 is exited critical zone.
*PID# 9221 is waiting for the semaphore.
*PID# 9220 is in the critical zone.
*PID# 9220 is exited critical zone.
*PID# 9221 is in the critical zone.
*PID# 9219 is waiting for the semaphore.
*PID# 9219 is in the critical zone.
*PID# 9221 is exited critical zone.
```

Figure 3.1 critical section trace

이번 과제에서는 동시 접근 제어를 구현하고자 세마포어를 사용했고 조교님이 보시기에 불편함이 없도록 sleep을 사용하여 고의로 동시접근을 발생시켰다. Figure 3.1에서 접근한 두 URL은 <http://www.man7.org/linux/man-pages/> 와 <http://www.man7.org/linux/man-pages/man3/intro.3.html> 이다.

터미널에 찍힌 로그를 보면 한 프로세스가 critical zone에 들어가서 나올때까지 다른 프로세스는 critical zone에 들어가지 못하고 waiting 하는것을 볼 수 있다.



```
*PID# 9231 is exited critical zone.
^C*PID# 9214 is waiting for the semaphore.
*PID# 9214 is in the critical zone.
*PID# 9214 is exited critical zone.
dongho@ubuntu:~/Desktop/system_programming/code/3-1$
```

Figure 4.2 after interupt

Ctrl+C로 인터럽트를 걸어 종료할때도 세마포어를 얻는것을 볼 수 있다. 사실 별

거 아닌거긴 하지만 이번 과제는 찍을게 없어 이거라도 찍어보았다.

```
dongho@ubuntu:~/Desktop/system_programming/code/3-1$ cat ~/logfile/logfile.txt
[MISS]http://www.man7.org/linux/man-pages/man3/intro.3.html-[2018/06/01, 15:18:07]
[MISS]http://www.man7.org/linux/man-pages/-[2018/06/01, 15:18:09]
[MISS]http://www.man7.org/linux/man-pages/style.css-[2018/06/01, 15:18:10]
[MISS]http://www.man7.org/style.css-[2018/06/01, 15:18:11]
[HIT]805/5ee659b267dc07648b127d6b0afdc7323e4b-[2018/06/01, 15:18:12]
[HIT]3ea/bc73133374d1533b35e92b0ed58785a49c10a-[2018/06/01, 15:18:13]
[HIT]http://www.man7.org/style.css
[HIT]http://www.statcounter.com/counter/counter_xhtml.js
[MISS]http://www.statcounter.com/counter/counter_xhtml.js-[2018/06/01, 15:18:16]
[MISS]-[2018/06/01, 15:18:17]
[HIT]da3/9a3ee5e6b4b0d3255bfef95601890afd80709-[2018/06/01, 15:18:18]
[HIT]
[HIT]ab6/de39bff5f79baf3851d41ad30ff47564130cd-[2018/06/01, 15:18:26]
[HIT]http://c.statcounter.com/t.php?sc_project=7422636&java=1&security=9b6714ff&u1=9E77BCB0EB394F94C08B41B31C7A729A&sc_random=0.9983585815141823&jg=new&rr=1.1.1.1.1.1.1&resolution=1916&h=956&camefrom=&u=http%3A//www.man7.org/linux/man-pages/&t=Linux%20man%20pages%20online&rcat=d&rdom=d&rdomg=new&bb=1&sc_snum=1&sess=7a9eb4&p=0&invisible=1
[HIT]630/8ed1546afb4d418664bc82b7f5ea08c495182-[2018/06/01, 15:18:28]
[HIT]http://c.statcounter.com/t.php?sc_project=7422636&java=1&security=9b6714ff&u1=9E77BCB0EB394F94C08B41B31C7A729A&sc_random=0.9796144193029925&jg=1&rr=1.1.1.1.1.1.1&resolution=1916&h=956&camefrom=&u=http%3A//www.man7.org/linux/man-pages/man3/intro.3.html&t=intro(3)%20-%20Linux%20manual%20page&rcat=d&rdomo=d&rdomg=1&sc_snum=1&sess=7a9eb4&p=0&invisible=1
**SERVER** [Terminated]run time: 542 sec. #sub process: 13
dongho@ubuntu:~/Desktop/system_programming/code/3-1$
```

Figure 4.3 log content

기록된 로그 내용은 Figure 4.3 과 같다. 맨 처음

<http://www.man7.org/linux/man-pages/> 와 <http://www.man7.org/linux/man-pages/man3/intro.3.html> 에 요청을 날린뒤 이후 자잘한 요소들을 요청한것이 기록되었음을 볼 수 있다.

5 결론 및 고찰

이번 과제의 핵심은 여러 프로세스의 동시 접근을 통제 및 제어하는 일을 구현하는 것이었다. 이를 종합적으로 locking algorithm이라 부른다고 한다. 본 과제에서 사용한 locking algorithm은 세마포어였다. 세마포어의 특징은 특정 구역에 특정 개수만큼의 소유자가 동시 접근할 수 있으며 동시 접근 가능한 수가 1인 경우를 특별히 Mutex라고 한다. Mutex는 Mutually Exclusive에서 따온 말로 상호배제라는 뜻을 나타내고 있다. 하지만 이런 유형의 메커니즘은 꽤 비용이 크다. 따라서 그냥 계속 무한루프를 돌며 값을 검사하는 스핀락이라는 것도 있다는것을 알게 되었다. 이렇게 구현한 동시접근 제어를 통해 좀 더 안정성 있는 프록시 서버를 구현할 수 있었다.

6 참고 레퍼런스

[https://en.wikipedia.org/wiki/Lock_\(computer_science\)](https://en.wikipedia.org/wiki/Lock_(computer_science))

- mutex의 어원 및 의미
- locking algorithm 개요

<http://algorithm.jioh.net/2010/10/%EB%9D%BD%ED%82%B9-%EB%A7%A4%EC%BB%A4%EB%8B%88%EC%A6%98%EC%9D%98-%EC%A2%85%EB%A5%98.html>

- 슬립 락, 논 슬립 락

18-1_SPLab_week11_Proxy+3-1.pdf

- System V semaphore 사용법. 처음에 이를 통해 구현했지만 나중에 요구사항이 변경되면서 POSIX로 수정함.

9.+semaphores.pdf

- POSIX semaphore 사용법. 이를 통해 본 과제 대부분을 구현할 수 있었다.