System Programming Lab #5 Report

2019-18499 김준혁

시스템 환경 및 실행 결과

Environment (개인 테스트 환경, 본 환경에서도 확인하였음)

```
dragonfish@dragonfish-VirtualBox:~/Desktop/SNU-SP-kernel-lab/kernellab-handout/p
tree$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.6 LTS
Release:        20.04
Codename:       focal
dragonfish@dragonfish-VirtualBox:~/Desktop/SNU-SP-kernel-lab/kernellab-handout/p
tree$ uname -ar
Linux dragonfish-VirtualBox 5.15.0-72-generic #79~20.04.1-Ubuntu SMP Thu Apr 20
22:12:07 UTC 2023 x86_64 x86_64 x86_64 GNU/Linux
```

Result – Self test (개인 환경에서 실행)

```
dragonfish@dragonfish-VirtualBox:~/Desktop/proxylab-handout$ ./port-for-user.pl
dragonfish: 1908
dragonfish@dragonfish-VirtualBox:~/Desktop/proxylab-handout$ cd tiny
dragonfish@dragonfish-VirtualBox:~/Desktop/proxylab-handout/tiny$ ./tiny 1907 &
[1] 2483
dragonfish@dragonfish-VirtualBox:~/Desktop/proxylab-handout/tiny$ cd ..
dragonfish@dragonfish-VirtualBox:~/Desktop/proxylab-handout$ ./proxy 1908 &
[2] 2494
dragonfish@dragonfish-VirtualBox:~/Desktop/proxylab-handout$ User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:10.0.3) Gecko/20120305 F
irefox/10.0.3
curl -v --proxy http://localhost:1908 http://localhost:1907/home.html
*   Trying 127.0.0.1:1908...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 1908 (#0)
> GET http://localhost:1907/home.html HTTP/1.1
> Host: localhost:1907
> User-Agent: curl/7.68.0
> Accept: */*
> Proxy-Connection: Keep-Alive
>
Recieved request header : GET http://localhost:1907/home.html HTTP/1.1
Accepted connection from (localhost, 56620)
GET /home.html HTTP/1.0Host: localhost:1907

Response headers:
HTTP/1.0 200 OK
Server: Tiny Web Server
Connection: close
Content-length: 120
Content-type: text/html

* Mark bundle as not supporting multiuse
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: Tiny Web Server
< Connection: close
< Content-length: 120
< Content-type: text/html
<
<html>
<head><title>test</title></head>
<body>
<img align="middle" src="godzilla.gif">
Dave O'Hallaron
</body>
</html>
* Closing connection 0
```

Result – driver.sh (sp 서버 환경에서 실행)

```
stu67@sp03:~/proxytest/proxylab-handout$ ./driver.sh
*** Basic ***
Starting tiny on 26208
Starting proxy on 2686
1: home.html
   Fetching ./tiny/home.html into ./.proxy using the proxy
   Fetching ./tiny/home.html into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
2: csapp.c
   Fetching ./tiny/csapp.c into ./.proxy using the proxy
   Fetching ./tiny/csapp.c into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
3: tiny.c
   Fetching ./tiny/tiny.c into ./.proxy using the proxy
   Fetching ./tiny/tiny.c into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
4: godzilla.jpg
   Fetching ./tiny/godzilla.jpg into ./.proxy using the proxy
   Fetching ./tiny/godzilla.jpg into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
5: tiny
   Fetching ./tiny/tiny into ./.proxy using the proxy
   Fetching ./tiny/tiny into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
Killing tiny and proxy
basicScore: 40/40

*** Concurrency ***
Starting tiny on port 32613
Starting proxy on port 13561
Starting the blocking NOP server on port 12153
Trying to fetch a file from the blocking nop-server
Fetching ./tiny/home.html into ./.noproxy directly from Tiny
Fetching ./tiny/home.html into ./.proxy using the proxy
Checking whether the proxy fetch succeeded
Success: Was able to fetch tiny/home.html from the proxy.
Killing tiny, proxy, and nop-server
concurrencyScore: 15/15

*** Cache ***
Starting tiny on port 25202
Starting proxy on port 31514
Fetching ./tiny/tiny.c into ./.proxy using the proxy
Fetching ./tiny/home.html into ./.proxy using the proxy
Fetching ./tiny/csapp.c into ./.proxy using the proxy
Killing tiny
Fetching a cached copy of ./tiny/home.html into ./.noproxy
Success: Was able to fetch tiny/home.html from the cache.
Killing proxy
cacheScore: 15/15

totalScore: 70/70
```

함수(구조체)별 구현 방법 (proxy.c 수정만 존재)

Cache 관련

struct cache_data

: 리퀘스트를 받은 url, 서버의 response, content 실제 데이터 및 각각의 길이, doubly linked list 구현을 위한 cache_data 포인터 prev, next를 가지는 구조체를 구성하였다.

void delete(struct cache_data *cache)
: 현재 'cache'가 가리키는 cache data를 linked list에서 단순히 제거한다. free는 실행하지 않는다. 이 delete는, 'cache'로 들어갈 argument가 find되거나 MAX_CACHE_SIZE를 넘어서는 경우에 LRU Policy에 의한, tail이 가리키는 데이터가 들어가며, NULL이나 허용되지 않은 값이 들어가지 않도록 하였다.

void insert_head(struct cache_data *cache)
: 들어온 'cache'를 linked list의 맨 앞에 넣는, 즉 LRU Policy에 따르는 캐싱을 위한 함수다.

struct cache_data *find(char *input_url)
: 들어온 url과 일치하는 cache data가 있는지 찾아서 반환하는 함수다. 'mutex' 및 'w' semaphore를 이용하여 concurrency를 구성하였다. head부터 list에 존재하는지 확인하는 절차를 거치며, cache에 없는 경우, NULL을 반환한다. 있는 경우에는 해당 cache data를 list에서 제거했다가(delete), insert_head를 통해 head쪽에 다시 삽입하여 LRU Policy를 적용하고, 반환한다.

void *add_cache(char *url, char *response, int response_length, char *content, int content_length)
: 새로운 데이터를 caching하는 함수다. 해당 데이터(response + content)가 MAX_OBJECT_SIZE를 넘어가는지를 체크하고 나서(넘으면 return), 기존 캐시와 합이 MAX_CACHE_SIZE를 넘지 않는지 확인하는 작업을 거친다. MAX_CACHE_SIZE를 넘길 경우, 넘기지 않을 때까지 tail의 cache data를 list에서 delete해주고, 해당 data를 free시켜준다. 이후 argument로 들어온 data를 cache_data 형태에 맞게 allocate 및 구성해주고, 완성된 cache_data를 insert_head로 list의 맨 앞에 삽입한다.

Proxy & Concurrency 관련

void transmit(int connect_fd)
: 이 함수는 크게 client로부터 request header 받기, server로 request header 전달하기, server로부터 response 및 content 받아 client에 전달하기로 이루어져있다. 이 3가지 부분으로 나누어 설명하고자 한다.

1. Client로부터 request header 받기
: client와 연결된 connect_fd와 rio를 통해 request header를 받고, 이를 명령어, url, http 버전으로 파싱한다. 이후의 입력 데이터는 무시해준다. 명령어가 'GET'인지를 체크, 아닌 경우 return해준다. url은 host(& port)와 filename으로 분리, host는 가능한 경우, port를 따로 분리시켜준다. 이 port는 1 이상의 적합한 값인지를 확인해준다.

2. Server로 request header 전달하기
: Server에 전달하기 전에, url이 기존에 들어와서 캐싱되었는지를 먼저 확인해주어, 캐

싱된 데이터가 있는 경우(일치), 해당 데이터를 바로 반환해주고 return 한다.
그 외의 경우, Server와 open_clientfd로 연결 시도, 이는 최대 20번까지 시도를 한다. 그리고, 들어왔던 request header를 specification에 맞게, 'GET /(filename) HTTP/1.0' & 'Host: (host):(port)₩r₩n₩r₩n' 형태로 전달한다.

3. Server로부터 response, content 받아 Client에 전달하기

: 먼저, response header를 받아 content가 chunk형태로 오는지 아닌지, content length는 얼마인지를 판별한다. 이 또한 Client에 전달하게 된다.
이후, chunk형태인 경우, 다음 chunk가 없을 때까지 계속 입력을 받는데, 이 입력은 char *content를 계속 realloc해주며 해당 입력을 계속 뒤에 붙여준다. content_length 또한 계속 더해준다. chunk형태가 아닌 경우, 단순히 한 번의 입력을 받아 content에 저장한다.
그렇게 저장한 content를 client와 연결된 connect_fd를 통해 전달하며, add_cache를 통해 가능한 경우, 캐싱을 시도하고 server_fd를 close한다.

void *thread(void *vargp)

: pthread_create에 사용되는 함수로, vargp는 connect_fd값이며, transmit(connect_fd)를 호출하고, 완료가 되면 connect_fd를 close하고 return 한다.

int main(int argc, char **argv)

: proxy 사용에 있어 port값을 먼저 확인하는데, specification에서는 4500 < port number < 65000이 있지만, 개인 환경에서 사용할 때, port-for-user.pl이 1908을 계속 반환해주다 보니, 1 이상이면 허용하는 식으로 그렇게 놔두었다. 그래서 주석 상으로 해당 조건을 확인하려 했다는 증거를 남긴 흔적이 있다.
이후, 해당 port로 open_listenfd를 이용하여 listen_fd와, proxy 행동을 위한 자원을 준비한다. 그리고 while문을 계속 돌며, 연결을 시도하는 Client에 대해 accpet하고 pthread_create()를 통해 스레드를 생성하고 thread()를 실행해준다. 만약 이 과정에서 오류 발생시, continue하여 다시, 또는 다른 입력을 받도록 해주었다.

공통

: csapp에 있는 wrapper function은 대부분 error-handling function을 쓰다보니 proxy가 terminate될 가능성이 존재한다. 그래서 최대한 csapp.c의 wrapper function보다는 일반적인 function을 최대한 이용하며, 에러 발생 시 fprintf(stderr,)를 통해 에러 내용을 출력하면서 return을 하는 등으로 proxy가 종료되지 않도록 최대한 보호하였다.

어려웠던 점

skeleton이 거의 없는 상태에서 specification을 이해하고, 기본 proxy 코드를 짜고, 수동으로 테스트해보는 과정이 가장 어려웠던 것 같다. proxy에 관한 내용은 이론적으로 어떻게 이루어지는지만 다루고, 기본 proxy 코드의 구성에 대해서는 알지 못하는 상태로 시작하는 것, 그리고 그 proxy를 test하는 것이 어떤 결과를 의미하는지를 깨닫는 것, specification에서 요구하는 것과 주어진 힌트를 이해하는 것이 오래걸리다보니, 시작이 가장 어렵게 되었던 것 같다. 이후 코드를 다 짜고, 테스트를 통해 70점을 받을 때까지는 생각보다 그리 오랜 시간이 걸리지 않았다. 시작이 반이라는 말이 있듯, 정말로 시작하는 것이 이해하는 데에 가장 오랜 시간이 걸렸으며, 받은 오답 결과를 찾고, 그 의미를 깨닫는 것이 많은 시행착오를 겪게 하였다. 관련한 여러 수업 pdf를 찾아보고, 인터넷을 통해 무엇이 문제였는지, 해결 방법은 어떻게 되는지 확인하면서 오랜 시간이 걸렸지만, 그 과정이 어느 정도 해결이 되니 그 뒤는 비교적 순탄하게 마무리되었다고 생각한다.

목표 및 새롭게 배운 점

proxy(networking)와 thread 및 semaphore를 모두 이용한 lab 과제였다보니, 여러 장에 걸친 지식을 실습해보면서 그것이 어떻게 사용되는지를 깨달을 수 있는 짜임새가 꽤나 괜찮은 과제였던 것 같다. 많은 시행착오를 겪으면서 각 함수가 어떤 부분에 사용되고, 어떻게 구성하면 좋은지를 알 수 있었다.

proxy부분을 통해서는 http request와 response의 format을 볼 수 있었고, 이를 어떻게 client/server에 전달할 수 있는지를 알 수 있었다. thread 부분은 직접적으로는 pthread_create/detach와 argument malloc 처리 말고는 거의 없지만, 'detach' mode가 확실히 join 없이 혼자서 terminate되는 것은 확실히 이런 구조의 프로그램에서 편하다고 느껴졌다. semaphore에서는 사실 본 수업(음성 파일)에서는 교수님이 reader-writer problem을 짜보라는 과제가 있었지만, 이미 해답 pdf가 제공되어 실제로 짜보지는 않았지만, 그런 비슷한 느낌으로 1st reader-writer problem에 대한 코드를 짜보게 되어서 은근히 실습으로 해보면서 익힐 수 있었다고 생각한다.