# <컴퓨터 네트워크 3차 과제>

학과: 컴퓨터과학과
학번: 2015147506
이름: 김기현
제출일: 2020.12.12

# 목차

# 1 Introduction

Code language: Python

version: gcc —version gcc 7.5.0

```
bill@bill-VirtualBox:~/바탕화면/project$ gcc --version
gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

linux 5.4.0 version

```
billygoat97@ubuntu:~$ uname -a
Linux ubuntu 5.4.0-52-generic #57~18.04.1-Ubuntu SMP Thu Oct 15 14:04:49 UTC 202
0 x86_64 x86_64 x86_64 GNU/Linux
```

Reference


http://itpsolver.com/java-%EC%86%8C%EC%BC%93%ED%86%B5%EC%8B%A0%EC%97%90%EC%84%9C-connection-reset-by-peer-peer%EB%A1%9C-%EC%97%B0%EA%B2%B0-%EC%9E%AC%EC%84%A4%EC%A0%95-%EC%98%A4%EB%A5%98%EA%B0%80-%EB%B0%9C%EC%83%9D/

https://realpython.com/python-sockets/

https://www.geeksforgeeks.org/creating-a-proxy-webserver-in-python-set-1/

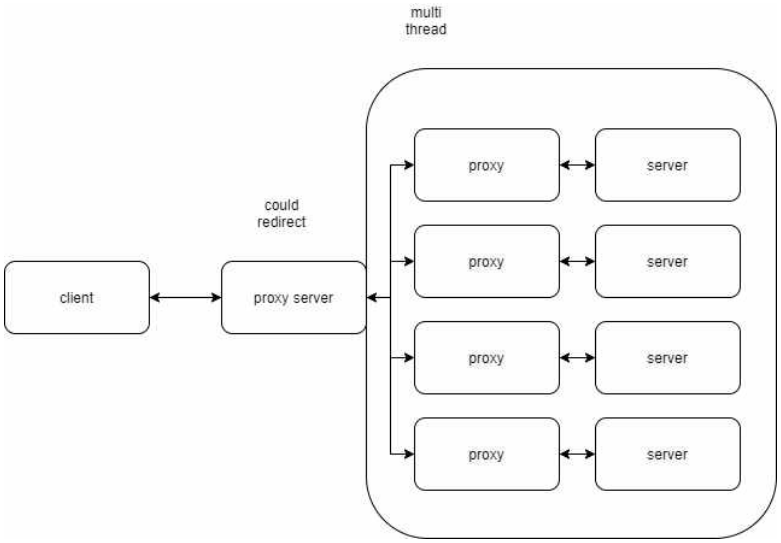
https://developer.mozilla.org/ko/docs/Web/HTTP/Basics_of_HTTP/MIME_types

Computer Network Lecture 2

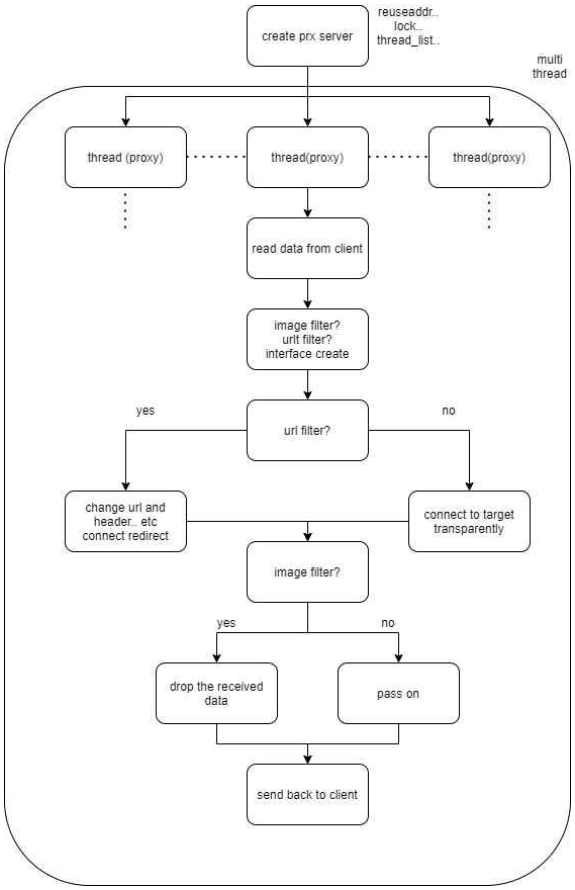
https://zzsza.github.io/development/2018/07/18/linux-top/

https://www.cyberciti.biz/faq/linux-find-process-name/

# 2 Code & Logic




전체적인 구조는 다음과 같으며, 크게 클라이언트와 소통하는 server 클래스와 proxy 클래스 두 가지로 나누어서 진행하였습니다.




전체적인 logic을 나타낸 순서도는 다음과 같습니다. 간단히 말해서 위의 구조대로 세운 이후에, client에서 정보를 얻고, image filter과 url filter에 대해서 정의 후 redirect 할지 말지 결정, 그 후에 image filter에 대해서 어떻게 할지 결정, 그 후에 data를 client로 돌려 보내주는 과정으로 크게 볼 수 있습니다.

코드 설명은 다음과 같습니다.

```
import socket
import sys
import threading
from threading import Thread
import os
```

기본적으로 이용한 import들은 다음과 같습니다. 3rd 파티를 이용하지 않았음을 알 수 있습니다.

```
############################## so far so good ##########################
port = int(sys.argv[1])  # get port from input -> 1 input only
proxy_server = Server(port) # create server by using class Server
proxy_server.run("yonsei", "www.linuxhowtos.org") # redirection -> anything th
```

가장 먼저 코드를 시작하기 앞서서 여러 클래스와 함수를 선언하였지만, port를 외부에서 받아오는 것과, 서버 클래스를 선언을 proxy_server로 선언하는 것을 알 수 있습니다. 또한 redirection의 지시를 위해 yonsei가 있다면, linuxhowtos.org로 redirection하는 것을 알 수 있습니다.

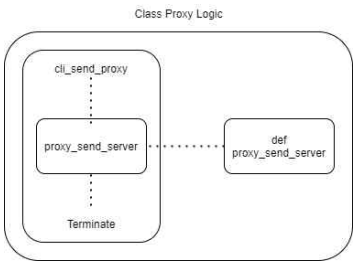
```
class Server:
    def __init__(self, port):
        self.port = port # get port number
        self.lock = threading.Lock() # to prevent interrupting thread
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # create of socket
        self.original = "" # original that should be changed
        self.redirect = "" # revised url
        self.thread_list = []
        self.image_tf = False
```

서버의 변수입니다. init으로 선언을 하자마자, 임의로 정할 수 있는 port을 저장을 하며, thread끼리 충돌을 방지하기 위한 lock, 그리고 현재 있는 서버의 소켓을 위한 소켓의 선언, redirection하는 내용들을 저장하기 위한 original과 redirect에 대해 string 형식을 준비해 놓았습니다. thread_list는 thread_id를 저장하는 모습입니다.  마지막으로 image_tf는 이미지를 쓸지 말지 정하는 과제 명세를 위하여 도입하였습니다.

```
def run(self, original = None, redirect = None): # start

        self.redirect = redirect
        self.original = original
        print("Starting proxy server on port: "+ str(self.port))
        self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,1) # in case of reusal
        self.socket.bind(("",self.port)) #bind with local port
        self.socket.listen() #wait

        while(True):
            try:
                conn, (cli_ip,cli_port) = self.socket.accept() # accept new socket (each thread)

                self.lock.acquire()
                global count # count how many times
                count += 1
                thread_id = 1
                while(thread_id in self.thread_list): # thread creation
                    thread_id += 1
                self.thread_list.append(thread_id)
                print("---------------------------------------")
                print(str(count)+ " [Conn:  "+str(thread_id)+"/   "+str(len(self.thread_list))+"]")
                self.lock.release()

                proxy = Proxy(self, thread_id, conn, cli_ip, cli_port) # make proxy server
                new = threading.Thread(target = proxy.client_send_proxy) # make start with client send proxy
                new.daemon = True
                new.start() # start

            except KeyboardInterrupt: # for exceptions
                self.socket.close()
                print("exit")
                sys.exit(1)
```

다음은 서버의 실제 run 구조에 대해서 보겠습니다. 2차과제와 마찬가지로 소켓 통신을 이용하며, 재사용을 위한 reuseaddr등이 이용이 되어습니다. original과 redirect 주소에 대해서 입력을 받기 위하여 run 자체에서 입력을 받았으며, bind후 다른 소켓의 접속을 위해 listen()을 하고 있습니다. 계속 소켓이 접속을 하려고 할 것이기 때문에 ctrl + c 가 아닌 이상 접속이 끊기지 않기 위해 while문 안에서 try except를 이용하여 실행시켰습니다. keyboardinterrupt인 ctrl +c 의 경우는 exit를 하며 try 내부에서는 소켓 접속을 받은 후 lock을 통해 thread간의 충돌을 방지하고자 하였습니다. lock.acquire을 하였으며, global 변수로 만든 count (총 몇 번의 접속이 있었는지)를 이용하였고, thread_id는 1부터 시작해 thread_list에 내용이 있다면 1씩 가산해서 하며, 아니라면 thread_list에 append를 합니다. 이 과정이 끝났다면 lock release를 할 수 있어 multi thread 의 혼선을 방지합니다. 주어진 스레드를 추후에 서술한 프록시 클래스에 넣기 위해 프록시 서버에 thread_id, socket, ip, port등을 전달하였으며, new daemon과 start를 통해 프록시 클래스를 실행시켰습니다.

프록시 클래스의 전체적인 로직은 다음과 같습니다. cli_send_proxy 함수 안에 proxy_send_server함수를 같이 추가시켜 실행하는 구조로 되어있습니다.



```python
class Proxy:
    def __init__(self, server, thread_id, cli_socket, cli_host, cli_port):
        self.server = server #main it is connected to
        self.thread_id = thread_id
        self.cli_socket = cli_socket
        self.srv_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.cli_host = cli_host
        self.cli_port = cli_port
        self.srv_host = str()
        self.srv_port = 80
        self.cli_prx_data = bytes()
        self.prx_cli_data = bytes()
        self.request = str()
        self.user_agent = str()
        self.status_code = str()
        self.content_type = str()
        self.content_length = str()
        self.url_filter = False
        self.image_filter = False
```

다음은 프록시 클래스의 변수입니다. 프록시 클래스는 연결되어있는 서버(기존에 만든 서버 클래스)와 연결하기 위해 server을, 자신의 thread_id, client와 연결하는 cli_socket, 자신의 소켓인 srv_socket, cli 의 adress, 접속할 host, port, cli-prx 간의 데이터 prx-cli간에 보내는 데이터를 byte (encode()위함)으로 설정했으며, request, user agent, status code, content type, length filter, image filter flag을 두 함수를 이용하기 위해 공용 변수로 설정 해놓았습니다.

함수는 두가지로 구성되어있으며, 각각 client_send_proxy(A), proxy_send_server(B)입니다. client과 prxoy, proxy과 server간의 연동을 의미합니다 구성을 보면, (A) - (B) - (A)로 돌아갑니다. 로직 흐름대로 보면 다음과 같습니다.

```python
def client_send_proxy(self):

    data = read_data(self.cli_socket) # get data from client
    self.cli_prx_data = data # in bytes

    imsi_data = byte2str(data) #decoding..
    cd = byte2str(data)
    line = imsi_data.split('\n')
    self.request = line[0]
    for l in line:
        l = l.strip() # not nessary erase

    if "HTTP" not in self.request :
        self.terminate()
        return

    if "HTTP" in line[0]:
        index = line[0].find("HTTP")
        self.request = line[0][:index]

    if "?image_on" in line[0] :
        self.image_filter = False
        self.server.image_tf = False
        imsi_image_filter = 'X'
    elif "?image_off" in line[0] :
        self.image_filter = True
        self.server.image_tf = True
        imsi_image_filter = 'O'
    else:
        self.image_filter = False
        self.server.image_tf = False
        imsi_image_filter = 'X'
```

data를 client socket으로부터 전달을 받으며, 전달 받은 data를 decoding을 한 후, imagefilter에 해당되는지, url filter에 해당되는지에 대해서 OX로 표시하도록 하는 인터페이슬 위한 조치를 취합니다. 만약에 HTTP 파일이 아니라면 terminate를 합니다.

```
print("[ "+imsi_url_filter+" ] URL filter | [ " + imsi_image_filter + " ] Image filter")
for l in line:
    l = l.strip()
    if l.find("Host: ") == 0:
        index = l.find(' ')
        imsi_srv_host = l[index + 1:]
        if ":" in imsi_srv_host:
            imsi_srv_host = imsi_srv_host.split(":")
            self.srv_host = imsi_srv_host[0]
            self.srv_port = int(imsi_srv_host[1])
        else:
            self.srv_host = imsi_srv_host
    elif l.find("User-Agent: ") == 0:
        l.split()
        index = l.find(' ')
        self.user_agent = l[index + 1:]

print("[CLI connected to "+str(self.cli_host)+":"+str(self.cli_port)+"]")
print("[CLI ==> PRX --- SRV]")
print(" > " + self.request)
print(" > " + self.user_agent)
```

인터페이스 구성을 위한 것으로 filter별 해당사항을 출력합니다. 그다음은, 접속하려는 srv_host, port를 추출하기 위해서 이용을 하며, request, agent 또한 추출하여 출력하는 interface입니다.

```
self.srv_port = int(self.srv_port)

self.proxy_send_server()
```

받은 정보를 data 변수에 self 저장한 후에 이를 이용하기 위해서 proxy_send_server함수를 실행합니다.

```
def proxy_send_server(self):
    if(self.url_filter == True): # redirection
        self.srv_socket.connect((self.server.redirect, self.srv_port))
        data_decode = byte2str(self.cli_prx_data) # decoding
        data_line = [line.strip() for line in data_decode.split('\n')]
        for i, line in enumerate(data_line): #change http and host into reidirect
            if "HTTP" in line:
                data_line[i] = data_line[i].split(' ')
                data_line[i][1] = data_line[i][1].replace(self.srv_host, self.server.redirect)
                data_line[i] = ' '.join(data_line[i])
                request_line = data_line[i]
            elif line.find("Host: ") == 0:
                data_line[i] = data_line[i].split(' ')
                data_line[i][1] = self.server.redirect
                data_line[i] = ' '.join(data_line[i])
        data_decode = "\r\n".join(data_line) # http header ends with \r\n
        self.cli_prx_data = data_decode.encode() # encode in order to save into bytes
        self.srv_host = self.server.redirect
        self.request = "GET " + self.server.redirect # request alter
```

proxy_send_server 함수입니다. 함수를 실행 시켰을 때에는 바로 url filter이 적용되어있는지에 대한 여부를 먼저 check 합니다. 만약에 그렇다고 한다면, http 안의 주소와, srv_host를 변경시켜, socket에 연결시키도로고 합니다. 변경한 정보를 전달하기 위하여 다시 byte형태로 젖아해 cli_prx_data에저장시켜 놓습니다.

```
else: # if not redirect
    self.srv_socket.connect((self.srv_host, self.srv_port))
    data_decode = byte2str(self.cli_prx_data)
    data_line = [line.strip() for line in data_decode.split('\n')]
    data_decode = "\r\n".join(data_line)
    self.cli_prx_data = data_decode.encode()
print("[SRV connected to "+str(self.srv_host)+":"+str(self.srv_port)+"]")
print("[CLI --- PRX ==> SRV]")
print(" > " + self.request + "\n" + " > " + self.user_agent)

self.srv_socket.sendall(self.cli_prx_data) # send to target socket whether it is redirected or not

data = read_data(self.srv_socket) # receive data from target
```

만약에 redirect할 필요가 없는 경우는 다음과 같이 그냥 원래대로 socket에 connect를 합니다. 그 후에 어떤 위치(socket address)에 connect했는지에 대한 출력하는 interface를 조정을 하여 request, user_agent에 대해서 조정을 해줍니다. 정제한 정보를 srv_socket 에 sendall을 통해 정보를 보내게 됩니다. 정보를 보낸 이후에는, read_data를 통해 target으로부터 request에 대

한 정보를 받게 됩니다.

```
if "HTTP" in line[0]: # get request
    index = line[0].find("HTTP")
    self.request = line[0][:index]
for l in line:
    l = l.strip()
index = line[0].find(' ')
self.status_code = line[0][index + 1:] #fetch whcih status code
self.content_type = str()
self.content_length = str()
for l in line:
    if l.find("Content-Type: ") == 0: # can show if it is image, text etc...
        imsi_index = l.find(' ')
        self.content_type = l[imsi_index + 1:].strip()
    if l.find("Content-Length: ") == 0:
        imsi_index = l.find(' ')
        self.content_length= l[imsi_index + 1:].strip() + "bytes" # what contents, and bytes
```

전달받은 data를 line으로 split하여 정보의 status code와, content type length로 분할하여 정보를 추출합니다. content type을 추출하여 차후에 image filtering에 이용하게 됩니다.

```
print("[CLI -- PRX <== SRV]") # for interface issues
print(" > " + self.status_code)
print(" > " + self.content_type +" "+ self.content_length)

if("image" in self.content_type and self.server.image_tf == True): # if there is image and filtering on
    self.prx_cli_data = bytes()
else:
    self.prx_cli_data = data
```

srv에서 prx로 정보를 전달 받았음을, 어떤 타입의 종류의 정보인지, 그리고 얼마나의 양을 가지고 있는지에 대해서 interface를 통해 표기한 후, content type과, image_filtering이 일어나고 있는지에 대해서 정보를 얻은 후, image를 표기하지 않는 off라면, 정보를 유실하는 bytes()를 하였으며, 그렇지 않다면 평소대로 출력을 하도록 하였습니다.

```
self.proxy_send_server()  done

print("[CLI <== PRX --- SRV]")
print(" > " + self.status_code)
print(" > " + self.content_type+" "+self.content_length)
self.cli_socket.sendall(self.prx_cli_data)
self.terminate()

print("[CLI TERMINATED]")
print("[SRV TERMINATED]")
```

이를 통해 proxy_send_server()함수가 완료가 됐다면, 호출한 함수 위치로 돌아간다면, prx에서 cli로 정보를 전달하였음을, status code, content type, content_length를 출력을 하면서 cli_socket에 전달을 하면서 현 스레드는 종료하게 됩니다. 해당 스레드는 종료하면서 terminate 함수를 호출하며, terminate함수는 socket들을 thread 내에 있는 연결을 모두 close()하게 됩니다.

```
def terminate(self):
    self.cli_socket.close()
    self.srv_socket.close()

    self.server.lock.acquire()
    self.server.thread_list.remove(self.thread_id)
    self.server.lock.release()
```

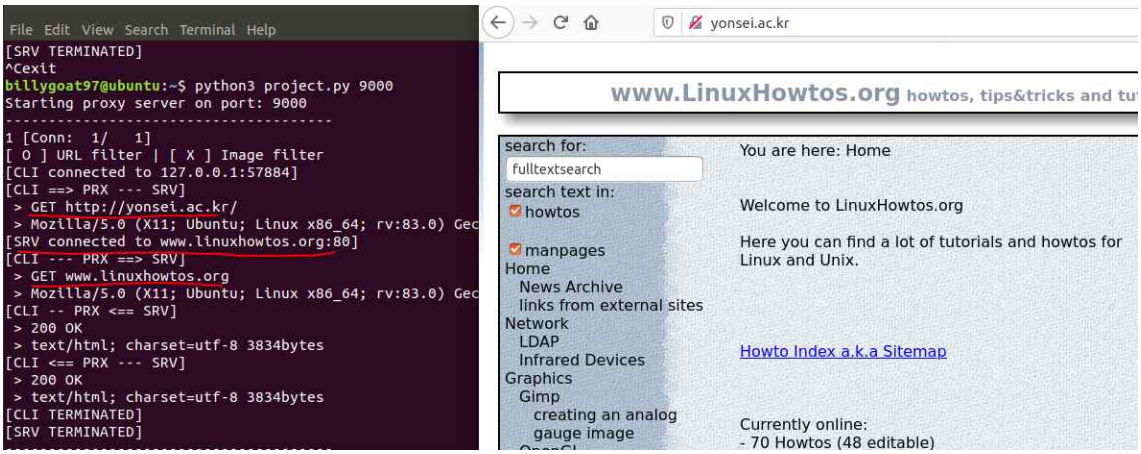
terminate함수는 연결을 끊을 때마다 호출하는 함수로, thread를 닫기 위해 cli_socket, srv_socket을 지우는 역할을 하고 있습니다.

```
def terminate(self):
    self.cli_socket.close()
    self.srv_socket.close()

    self.server.lock.acquire()
    self.server.thread_list.remove(self.thread_id)
    self.server.lock.release()
    global flag
    flag = 1
    global count
    count += 1
    print("--------------------------------------")
    print(str(count)+ " [Conn:  "+str(self.thread_id)+"/   "+str(len(self.server.thread_list))+"]")
    print("[ "+self.imsi_url_filter+" ] URL filter | [ " + self.imsi_image_filter + " ] Image filter")
    print()
    print("[CLI connected to "+str(self.cli_host)+":"+str(self.cli_port)+"]")
    print("[CLI ==> PRX --- SRV]")
    print(" > " + self.request)
    print(" > " + self.user_agent)
    print("[SRV connected to "+str(self.srv_host)+":"+str(self.srv_port)+"]")
    print("[CLI --- PRX ==> SRV]")
    print(" > " + self.request + "\n" + " > " + self.user_agent)
    print("[CLI -- PRX <== SRV]") # for interface issues
    print(" > " + self.status_code)
    print(" > " + self.content_type +" "+ self.content_length)
    print("[CLI <== PRX --- SRV]")
    print(" > " + self.status_code)
```

위의 그림에서 출력을 그때그때 하는 것처럼 보였지만, 실제로 disconnect할 때 모두 출력을 해야 하기 때문에 나머지는 주석 처리 후, terminate 부분에서 모두 출력하는 것으로 하였습니다. 이를 위해서 global count로 끝난 순서대로 출력을 하도록 하였습니다.

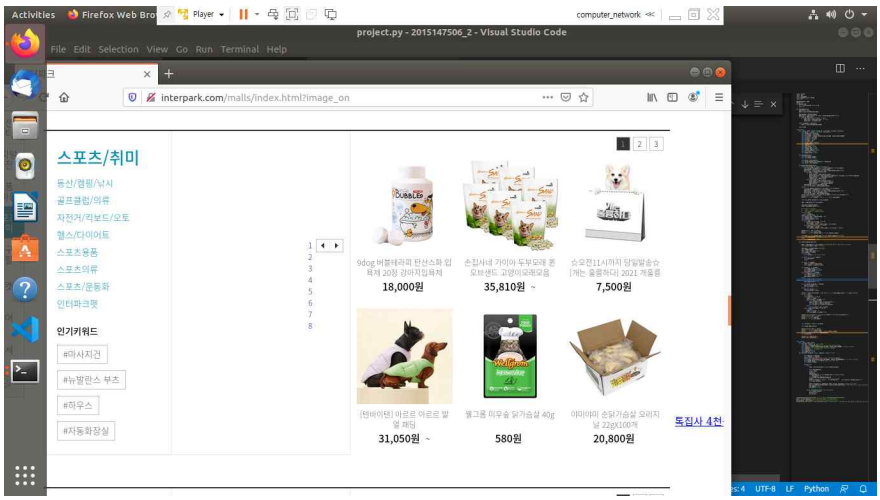
# 3 Results

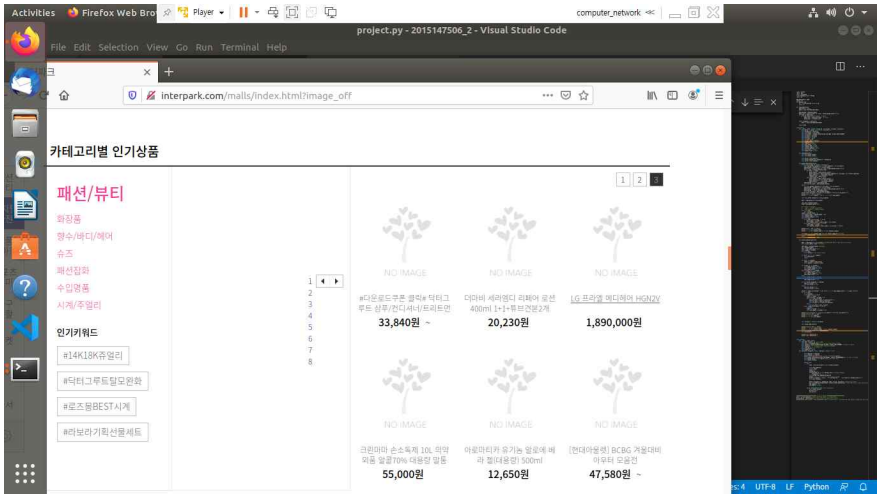
(1) Redirection




redirection을 통해 yonsei가 들어간 경우, linuxhowtos.org로 넘어간 결과입니다. image filtering을 이용하지 않았기 때문에 X가 되어있으며, url filtering이 되어있기 때문에 O가 쳐져있음을 알 수 있습니다. 클라이언트에서 프록시 서버로 전달할 때에는 yonsei에 접속하라고 했지만,프록시에서 메시지를 가로채서 linuxhowto org redirect 하는 것을 볼 수 있습니다. 이로 인해서 서버에서는 프록시에게 linux 관련 정보를 주어지며, 이를 client에게 전달하여 우측 화면과 같은 결과를 얻을 수 있습니다.

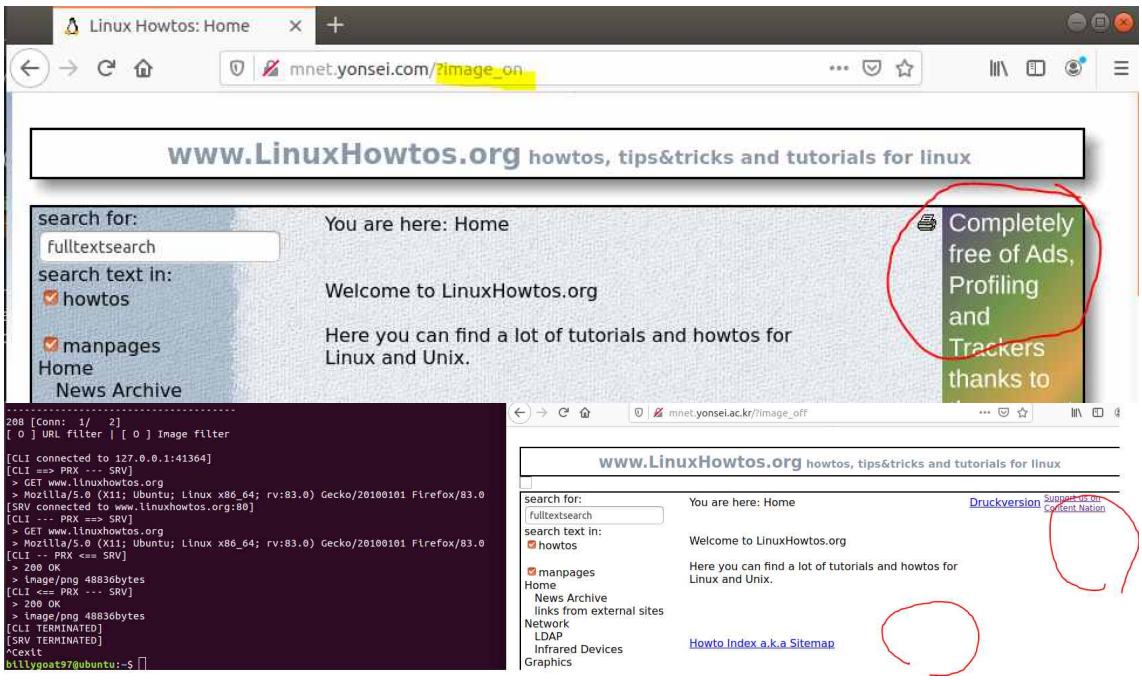
2) image on/off






image_off/on를 했을 경우 인터파크 홈페이지에서 제품 이미지가 나오지 않는 모습입니다. 그저 이미지가 사라지는 것이 아닌, 이미지 파일 위치가 정해져 있기 때문에 다음과 같은 결과로 이어지게 됩니다. 하단의 그림에서 인터파크에 접속했을 때 볼 수 있으며, thread가 multi로 되어있는 것을 볼 수 있습니다.

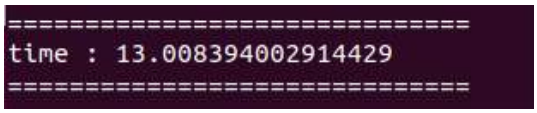
```
--------------------------------------
1688 [Conn:  1/   2]
[ X ] URL filter | [ O ] Image filter

[CLI connected to 127.0.0.1:56086]
[CLI ==> PRX --- SRV]
 > GET http://bimage.interpark.com/goods_image/0/2/0/8/343770208s.jpg
 > Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
[SRV connected to bimage.interpark.com:80]
[CLI --- PRX ==> SRV]
 > GET http://bimage.interpark.com/goods_image/0/2/0/8/343770208s.jpg
 > Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
[CLI -- PRX <== SRV]
 > 200 OK
 > image/jpeg 27969bytes
[CLI <== PRX --- SRV]
 > 200 OK
 > image/jpeg 27969bytes
[CLI TERMINATED]
[SRV TERMINATED]
```




다음의 예시는 image filtering과, redirection 모두 있는 예시입니다. image_on을 했을 때에는 ad의 그림이 보이지만, image_off 했을때는 ad의 그림이 보이지 않으며 모두 linuxhowtos.org로 redirection이 된 것을 볼 수 있습니다.

# 4 single threading/ multi threading

싱글스레딩과 멀티 스레딩을 통해 나온 time usage는 다음과 같습니다. 같은 페이지를 여러 번의 횟수를 거쳐서 결과를 얻기 위해서 ctrl + shift + r을 이용하여 매번 캐쉬를 제거하고 측정을 하였습니다. 싱글스레딩을 위해서는 server 클래스와 프록시 클래스에 간단한 조율을 거쳐 이용하였습니다.

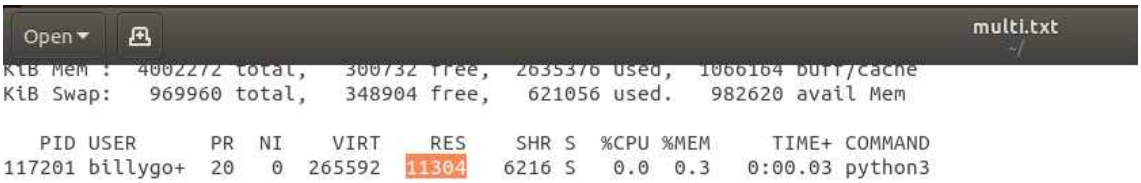

다음과 같이 20번마다 시간을 측정하였습니다. (재시작 버튼을 누르는 시간의 오차가 1초가량 존재할 수 있음.) time.time()을 이용하여 측정하였습니다

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| sin | 13.00 | 15.02 | 16.85 | 14.22 | 17.32 | 15.66 | 16.55 | 14.57 | 16.13 | 14.89 |
| mul | 2.42 | 2.53 | 2.62 | 2.44 | 2.46 | 2.77 | 2.41 | 2.66 | 2.90 | 2.57 |

싱글 스레딩을 이용할 경우에는 병렬적이지 않고 직렬적이기 때문에 다른 요청들이 끝날때까지 기다리는 시간이 가산되어 시간이 매우 비효율적으로 늘어남을 알 수 있습니다. 재시작을 누르기 위해서 시간이 조금 소모되는 것을 감안한다면 멀티 스레딩이 훨씬 효율적임을 알 수 있습니다. 시간과 메모리 두가지 토끼를 모두 잡는 것이 좋지만, 시간의 경우, 이렇게 비효율적인 결과물이 날 때는 메모리부족이 있는 한이 있더라도 멀티 스레드가 훨씬 효율적임을 예측할 수 있습니다.

```
billygoat97@ubuntu:~$ ps aux | grep "a 9000"xt
billygo+ 116239  0.0  0.0  14432  1024 pts/1    S+   18:52   0:00 grep --color=auto a 9000xt
billygoat97@ubuntu:~$ top -p 116239 -b -d 1 > multi.txt
^Cbillygoat97@ubuntu:~
billygoat97@ubuntu:~$ ps aux | grep "a 9000"xt
billygo+ 116342  0.0  0.0  14432  1016 pts/1    S+   18:55   0:00 grep --color=auto a 9000xt
billygoat97@ubuntu:~$ top -p 116342 -b -d 1 > single.txt
^Cbillygoat97@ubuntu:~$
```

다음은 메모리 관련입니다. ps aux | grep “project.py”xt 로 pid를 얻었으며
top -p [pid] -b -d 1 > txt로 1초마다 결과물을 얻었습니다.




다음과 같은 형식으로 구성되어있으며, 초당 메모리 usage의 차를 구하여 계산하였습니다.
ex) 10508 - 10308 = 200

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| sin | 200 | 235 | 199 | 242 | 243 | 219 | 274 | 255 | 189 | 249 |
| mul | 608 | 723 | 698 | 711 | 598 | 674 | 851 | 605 | 613 | 666 |

대략적으로 초당 memory usage는 400 가량의 차가 났으나, 동시에 얼마나 많이 처리하는지에 대한 내용을 고찰해 본다면, multi thread가 훨씬 더 자원의 효율적 이용이라는 측면에서 더욱 이득이라는 측면을 볼 수 있습니다. (memory usage/time)

## 애로사항 및 소감

프록시 서버를 이용하여 redirection을 할 때, 조금만 형식이 틀어져도 400 error이 떠서 이 형식을 맞추기 위해 시행착오가 상당수 있었으며, image_off/on을 할 때, data를 없애는 것이 쉽지 않았습니다. 그리고 정부 기관에 대해서 접속을 하는 예시를 하려고 했었는데 정부 기관 웹사이트에서는 접속이 잘 되지 않아 다른 웹사이트를 찾아야 한다는 것을 깨닫기 전까지 시간이 조금 걸렸습니다.