

# 운영체제 과제 (Assignment #1. Process and thread)

박지환 (184128) | 전남대학교 지역바이오시스템공학과 생물산업기계공학전공

## #1. My first web-server

Step 1) server.c를 컴파일하고 실행해보세요.

- assignment1 폴더에 있는 server.c 파일을 gcc 컴파일러를 이용하여 다음과 같이 컴파일하였습니다.

```
gcc server.c -p server
```

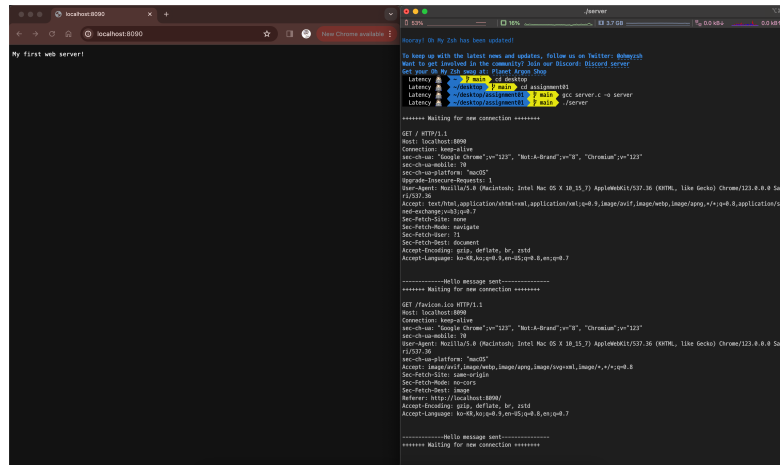
```
Latency ~ ~ main cd desktop
Latency ~ /desktop ~ main cd assignment01
Latency ~ /desktop/assignment01 ~ main gcc server.c -o server
Latency ~ /desktop/assignment01 ~ main ./server

++++++ Waiting for new connection ++++++
_
```

- 생성된 실행 파일인 server를 ./server 명령어를 통해 실행하였습니다.

Step 2) 웹 브라우저를 통해 <http://localhost:8090/> 로 접속을 해보세요. 그리고, server.c의 sleep 주석을 uncomment하시고 여러개의 동시접속을 만들어보세요. accesses.py를 이용하시면 더욱 쉬울겁니다!

- <http://localhost:8090/> 에 접속하여 웹 서버가 정상적으로 작동하는지 확인했습니다.



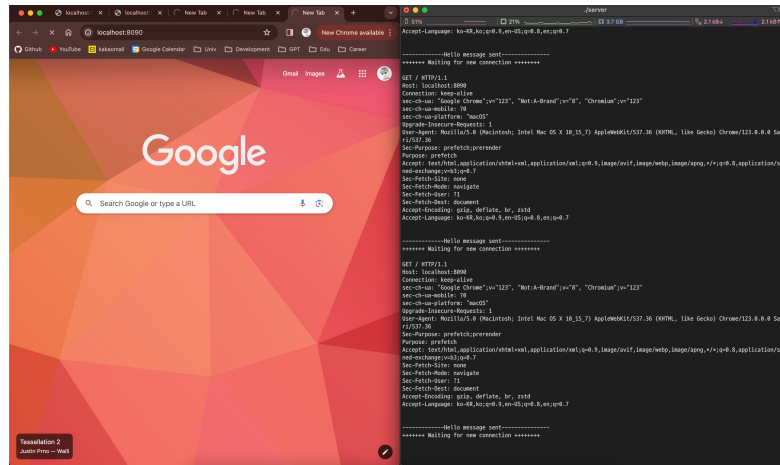
- 이후 server.c 파일에서 sleep 주석을 주석 해제하였습니다.

```

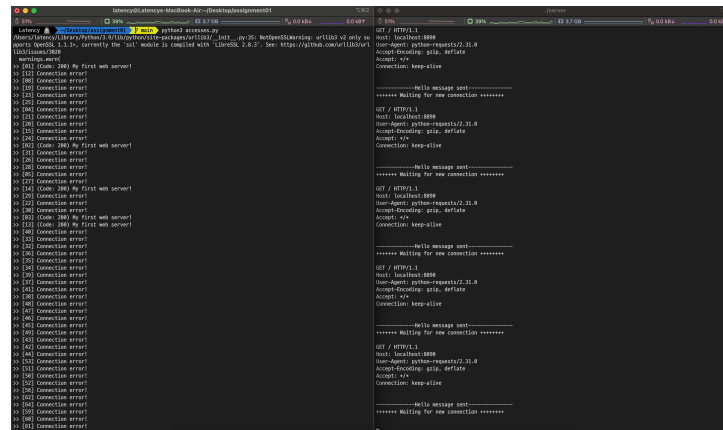
server.c -- Edited
C server | No Selection
14 int main(int argc, char const *argv[])
15 {
16     memset(address.sin_zero, '\0', sizeof address.sin_zero);
17
18     if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0)
19     {
20         perror("In bind");
21         exit(EXIT_FAILURE);
22     }
23     if (listen(server_fd, 10) < 0)
24     {
25         perror("In listen");
26         exit(EXIT_FAILURE);
27     }
28     while(1)
29     {
30         printf("\n+++++ Waiting for new connection +++++\n\n");
31         if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
32                                (socklen_t *)&addrlen)) < 0)
33         {
34             perror("In accept");
35             exit(EXIT_FAILURE);
36         }
37
38         char buffer[30000] = {0};
39         valread = read(new_socket, buffer, 30000);
40         printf("%s\n", buffer);
41         // uncomment following line and connect many clients
42         sleep(5); // 기존에 주석처리 되어있던 부분 주석 해제
43         write(new_socket, hello, strlen(hello));
44         printf("-----Hello message sent-----");
45         close(new_socket);
46     }
47     return 0;
48 }
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65 }
66
67

```

- 이후 accesses.py 스크립트를 실행해보기 전 직접 동시 접속을 시도하여 sleep이 제대로 실행되는 것을 확인해보았습니다.



- 다음으로 `accesses.py` 스크립트를 실행하여 동시에 여러 개의 요청을 서비스에 보내보겠습니다.



## Step 3) fork를 server.c에 추가하여 sleep 5가 있더라도 여러 요청을 잘 처리할 수 있게 만들어 봅시다!

- 클라이언트의 요청을 받으면 `fork()`를 호출하여 자식 프로세스를 생성하고, 자식 프로세스는 클라이언트의 요청을 처리하고 메시지를 보낸 후 종료되며, 부모 프로세스는 새로운 클라이언트의 연결을 계속해서 받게 하여 서버가 동시에 여러 요청을 처리할 수 있게 작성하였습니다.

```

while(1)
{
    printf("\n+++++++ Waiting for new connection ++++++\n\n");
    if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
        (socklen_t*)&addrlen))<0)
    {
        perror("In accept");
        exit(EXIT_FAILURE);
    }

    int pid = fork();
    if(pid == 0)
    {
        char buffer[30000] = {0};
        valread = read( new_socket , buffer, 30000);
        printf("%s\n",buffer );
        sleep(5);
        write(new_socket , hello , strlen(hello));
        printf("-----Hello message sent-----");
        close(new_socket);
        exit(0);
    }

    else if(pid > 0)
    {
        close(new_socket);
    }

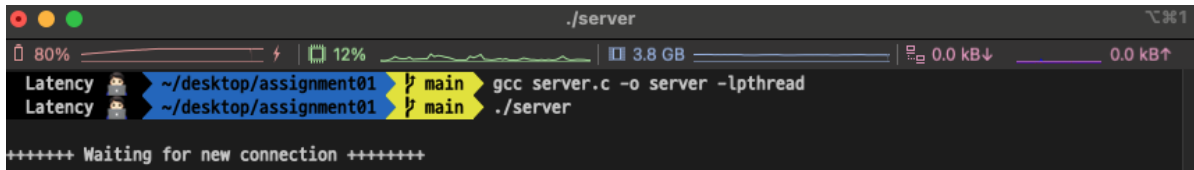
    else
    {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }
}
return 0;
}

```

## #2. My second web server

**Step 1) server.c를 컴파일 하고 실행해 보세요.**

- 컴파일러에 -lpthread 옵션을 추가하여 pthread 라이브러리를 링크하였습니다.



A terminal window titled `./server` with a dark background. The top status bar shows system metrics: 80% battery, 12% CPU usage, 3.8 GB memory, and network activity (0.0 kB down, 0.0 kB up). The terminal content shows two commands being executed in a shell: `gcc server.c -o server -lpthread` and `./server`. The output of the second command is `+++++++ Waiting for new connection ++++++`. The terminal also displays a visual representation of the process tree, showing the current process as `main` with a parent `main`.

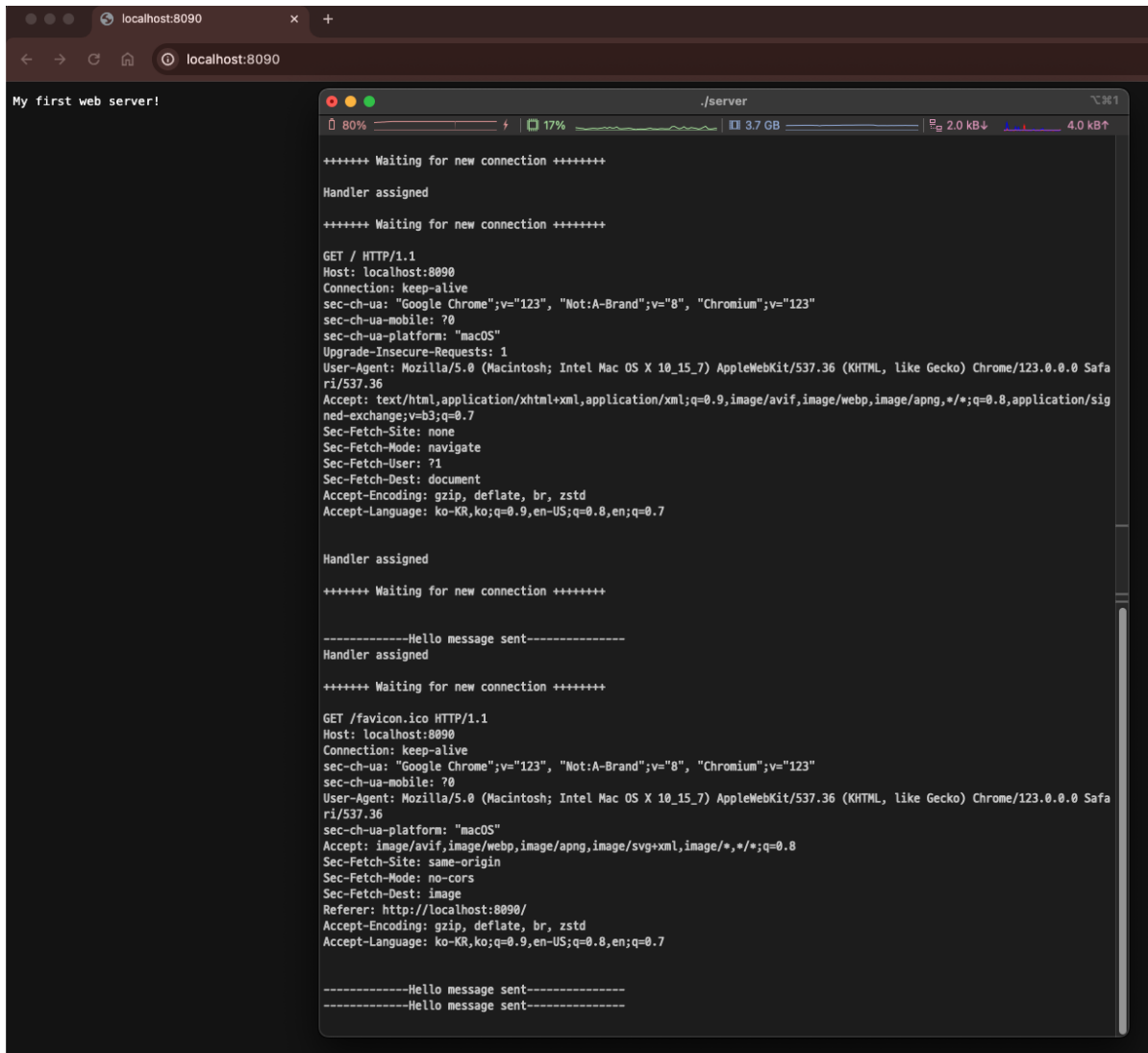
**Step 2) 웹 브라우저를 통해 `http://localhost:8090/`로 접속을 해보세요. 그리고, `server.c`의 `sleep` 주석을 `uncomment`하시고 여러개의 동시접속을 만들어보세요. `accesses.py`를 이용하시면 더욱 쉬울겁니다!**

- 현재까지는 #1 에서와 동일하게 작동합니다.

**Step 3) pthread 라이브러리를 활용하여 `server.c`를 업그레이드 해보세요! 아마 프로그램 구조가 조금 바뀌게 될겁니다!**

- 기존 `server_01.c` 파일에서 `fork()` 호출을 제거하고 `pthread_create()` 함수를 사용해 각 클라이언트 연결을 처리하는 별도의 스레드를 생성하였습니다.
- 새로운 소켓 핸들러를 스레드 함수에 전달하기 위해 동적 메모리 할당을 사용하였고, 각 클라이언트 요청을 처리한 후 스레드를 종료하게 설정하였습니다.

```
server_02.c
C server_02 | main(argc, argv)
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <netinet/in.h>
6 #include <pthread.h>
7
8 #define PORT 8090
9
10 void *handle_connection(void *socket_desc) {
11     int sock = *(int*)socket_desc;
12     long valread;
13     char *hello = "HTTP/1.1 200 OK\nContent-Type: text/plain" \
14                 "Content-Length: 20\n\nMy first web server!";
15     char buffer[30000] = {0};
16     valread = read(sock, buffer, 30000);
17     printf("%s\n", buffer);
18     sleep(5);
19     write(sock, hello, strlen(hello));
20     printf("-----Hello message sent-----\n");
21     close(sock);
22     free(socket_desc);
23
24     return 0;
25 }
26
27 int main(int argc, char const *argv[]) {
28     int server_fd, new_socket;
29     struct sockaddr_in address;
30     int addrlen = sizeof(address);
31
32     if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
33         perror("In socket");
34         exit(EXIT_FAILURE);
35     }
36
37     address.sin_family = AF_INET;
38     address.sin_addr.s_addr = INADDR_ANY;
39     address.sin_port = htons(PORT);
40
41     memset(address.sin_zero, '\0', sizeof address.sin_zero);
42
43     if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
44         perror("In bind");
45         exit(EXIT_FAILURE);
46     }
47     if (listen(server_fd, 10) < 0) {
48         perror("In listen");
49         exit(EXIT_FAILURE);
50     }
51
52     while(1) {
53         printf("\n+++++ Waiting for new connection +++++\n\n");
54         if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t*)&addrlen)) < 0) {
55             perror("In accept");
56             exit(EXIT_FAILURE);
57         }
58
59         pthread_t thread_id;
60         int *new_sock = malloc(1);
61         *new_sock = new_socket;
62
63         if (pthread_create(&thread_id, NULL, handle_connection, (void*) new_sock) < 0) {
64             perror("could not create thread");
65             return 1;
66         }
67
68         printf("Handler assigned\n");
69     }
70
71     return 0;
72 }
73
74
```



### #3. Orchestration and Load-balancing

Step 1) server.c에 10칸정도의 크기를 가지는 메세지 큐를 추가하세요.

- 각 연결에 대해 메세지 큐를 만들고, 클라이언트로부터 받은 메세지를 해당 큐에 저장하게 하였습니다.
- 메세지 큐는 최대 10개의 메세지를 저장하게 하였고 각 연결이 종료될 때 메세지 큐도 닫힙니다.

```

C server_03
C server_03 ) handle_connection(socket_desc)

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <netinet/in.h>
6 #include <pthread.h>
7 #include <fcntl.h>
8 #include <sys/stat.h>
9 #include <mqueue.h>
10
11 #define PORT 8090
12 #define MAX_MESSAGES 10
13 #define MAX_MSG_SIZE 256
14 #define MSG_BUFFER_SIZE (MAX_MSG_SIZE + 10)
15 #define QUEUE_NAME "/my_queue"
16
17 void *handle_connection(void *socket_desc) {
18     int sock = *(int*)socket_desc;
19     long valread;
20     char *hello = "HTTP/1.1 200 OK\nContent-Type: text/plain" \
21         "Content-Length: 20\n\nMy first web server!";
22     char buffer[30000] = {0};
23     valread = read(sock, buffer, 30000);
24
25     mqd_t mq;
26     struct mq_attr attr;
27
28     attr.mq_flags = 0;
29     attr.mq_maxmsg = MAX_MESSAGES;
30     attr.mq_msgsize = MAX_MSG_SIZE;
31     attr.mq_curmsgs = 0;
32
33     mq = mq_open(QUEUE_NAME, O_CREAT | O_RDWR, 0644, &attr);
34     if(mq == -1) {
35         perror("In mq_open");
36         exit(EXIT_FAILURE);
37     }
38
39     if(mq_send(mq, buffer, strlen(buffer), 0) == -1) {
40         perror("In mq_send");
41         exit(EXIT_FAILURE);
42     }
43
44     printf("%s\n", buffer);
45     sleep(5);
46     write(sock, hello, strlen(hello));
47     printf("-----Hello message sent-----\n");
48     close(sock);
49     free(socket_desc);
50     if(mq_close(mq) == -1) {
51         perror("In mq_close");
52         exit(EXIT_FAILURE);
53     }
54
55     return 0;
56 }
57
58 int main(int argc, char const *argv[]) {
59     int server_fd, new_socket;
60     struct sockaddr_in address;
61     int addrlen = sizeof(address);
62
63     if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
64         perror("In socket");
65         exit(EXIT_FAILURE);
66     }
67
68     address.sin_family = AF_INET;
69     address.sin_addr.s_addr = INADDR_ANY;
70     address.sin_port = htons(PORT);
71
72     memset(address.sin_zero, '\0', sizeof address.sin_zero);
73
74     if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
75         perror("In bind");
76         exit(EXIT_FAILURE);
77     }
78     if (listen(server_fd, 10) < 0) {
79         perror("In listen");
80         exit(EXIT_FAILURE);
81     }
82
83     while(1) {
84         printf("\n+++++ Waiting for new connection +++++\n\n");
85         if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t*)&addrlen)) < 0) {
86             perror("In accept");
87             exit(EXIT_FAILURE);
88         }
89
90         pthread_t thread_id;
91         int *new_sock = malloc(1);
92         *new_sock = new_socket;
93
94         if (pthread_create(&thread_id, NULL, handle_connection, (void*) new_sock) < 0) {
95             perror("could not create thread");
96             return 1;
97         }
98
99         printf("Handler assigned\n");
100     }
101
102     return 0;
103 }
104

```



**Step 2) 메시지큐의 용량이 80%가량 차오르면 서버로직을 추가하도록 만들어보세요. 그리고 이후 메시지큐 메시지들은 추가된 로직들끼리 돌아가며 처리(Round-robin)가 되도록 합시다.**

- 메시지 큐가 80% 이상 차면 각 서버 로직이 독립적인 스레드에서 실행되고, 각 스레드가 메시지 큐에서 메시지를 순차적으로 읽어 처리하기 때문에 새로운 서버 로직을 생성하고 라운드 로빈 방식으로 메시지를 처리하게 하였습니다.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <netinet/in.h>
6 #include <pthread.h>
7 #include <fcntl.h>
8 #include <sys/stat.h>
9 #include <mq.h>
10
11 #define PORT 8090
12 #define MAX_MESSAGES 10
13 #define MAX_MSG_SIZE 256
14 #define MSG_BUFFER_SIZE (MAX_MSG_SIZE + 10)
15 #define QUEUE_NAME "/message_queue"
16 #define MAX_SERVERS 5
17
18 typedef struct {
19     int id;
20     mqd_t mq;
21 } Server;
22
23 Server servers[MAX_SERVERS];
24 int serverIdx = 0;
25
26 void *server_logic(void *arg) {
27     Server server = *(Server *)arg;
28     char buffer[MSG_BUFFER_SIZE];
29     while (1) {
30         ssize_t bytes_read = mq_receive(server.mq, buffer, MSG_BUFFER_SIZE, NULL);
31         buffer[bytes_read] = '\0';
32         printf("Server %d received: %s\n", server.id, buffer);
33     }
34 }
35
36 void *handle_connection(void *socket_desc) {
37     int sock = *(int*)socket_desc;
38     long valread;
39     char *hello = "HTTP/1.1 200 OK\nContent-Type: text/plain \n\n"
40                 "Content-Length: 20\n\nMy first web server!";
41     char buffer[30000] = {0};
42     valread = read(sock, buffer, 30000);
43     printf("%s\n", buffer);
44
45     struct mq_attr attr;
46     attr.mq_flags = 0;
47     attr.mq_maxmsg = MAX_MESSAGES;
48     attr.mq_msgsize = MAX_MSG_SIZE;
49     attr.mq_curmsgs = 0;
50
51     mqd_t mq = mq_open(QUEUE_NAME, O_CREAT | O_RDONLY, 0644, &attr);
52     if (mq == -1) {
53         perror("In mq_open");
54         exit(EXIT_FAILURE);
55     }
56
57     if (attr.mq_curmsgs > MAX_MESSAGES * 0.8) {
58         if (serverIdx < MAX_SERVERS) {
59             servers[serverIdx].id = serverIdx;
60             servers[serverIdx].mq = mq;
61             pthread_t server_thread;
62             if (pthread_create(&server_thread, NULL, server_logic, &servers[serverIdx]) < 0) {
63                 perror("could not create server thread");
64                 return NULL;
65             }
66             serverIdx++;
67         }
68     }
69
70     sleep(5);
71     write(sock, hello, strlen(hello));
72     printf("-----Hello message sent-----\n");
73     close(sock);
74     mq_close(mq);
75     free(socket_desc);
76
77     return 0;
78 }
79
80 int main(int argc, char const *argv[]) {
81     while(1) {
82         if (pthread_create(&thread_id, NULL, handle_connection, (void*) new_sock) < 0) {
83             perror("could not create thread");
84             return 1;
85         }
86         printf("Handler assigned\n");
87     }
88     return 0;
89 }
90
91
92
93
94
95

```

**Step 3) 반대로 메시지큐의 용량이 20% 이하로 유지되면 서버 로직을 제거 하도록 만들어보세요. 당연히 첫번째 로직은 사라지면 안됩니다! 메시지 수신 속도에 따라 서버 로직의 갯수의 변화를 한번 확인해보는 것도 재미있는 결과 일겁니다. 뭐 적당히 초당 1개에서 100개 정도로 메시지를 보내보면 될거같네 요. 함께 제공해드린 accesses.py에 sleep과 쓰레드 생성 갯수 등 을 조절 하면 하실 수 있으실겁니다!**

- `accesses.py` 파일을 메시지 수신 속도에 따라 서버 로직의 갯수 변화를 관찰하기 위해 메 세지 전송 속도를 조절할 수 있도록 `access_page` 함수에 `time.sleep`을 추가하고, `CONNECTIONS` 변수를 통해 동시 연결 수를 조절하게 하였습니다.

The image shows two terminal windows. The left window is a terminal running a script that installs and updates Homebrew packages. The right window shows the code of the `accesses.py` script, which is a Python program that uses the `requests` library to make HTTP requests and uses `concurrent.futures` to manage a pool of threads. The script includes a `sleep` function and a `CONNECTIONS` variable to control the number of concurrent connections.

## #4. What is the difference between #1 and #2?

**Step 1) server.c의 대략적인 동작, #1의 접근방법과 #2의 접근방법의 장단 점, 동작형태등을 적당히 적어봅시다.**

- `server.c`는 웹 요청을 받아 처리하는 기능을 하며, 웹 브라우저를 통해 서버에 접속하면 간 단한 메시지를 반환합니다.
- #1
  - 클라이언트의 요청마다 새로운 프로세스를 생성하여 요청을 처리하는 멀티 프로세스 기반 형태이며, `fork()` 시스템 콜을 사용하여 부모 프로세스(서버)가 자식 프로세스를 생성하고 각각의 요청을 별도의 프로세스에서 처리합니다.
  - 장점: 각 요청이 독립적인 메모리 공간에서 처리되므로 요청 간의 데이터 충돌을 방지 할 수 있습니다.

- 단점: 프로세스 생성과 관리에 높은 오버헤드가 발생하여 시스템 자원의 한계로 인해 동시에 처리할 수 있는 요청의 수가 제한됩니다.
  - 변수 및 메모리: 멀티 프로세스 모델에서는 각 프로세스가 독립적인 메모리 공간을 가지므로 변수 공유에 있어 IPC 기법을 사용해야 합니다.
  - 프로그램 작동 형태: 멀티 프로세스 모델은 독립된 프로세스에서 처리하기 때문에 하나의 요청 처리에 실패하더라도 다른 요청의 처리에 영향을 미치지 않습니다.
- #2
    - 클라이언트의 요청마다 새로운 스레드를 생성하여 처리하는 멀티 스레드 기반 형태이며, pthread 라이브러리를 사용하여 멀티 스레딩을 구현했습니다. 이는 프로세스 내에서 병렬 실행이 가능한 스레드를 생성하여 각 요청을 처리할 수 있습니다.
    - 장점: 스레드는 프로세스보다 훨씬 적은 자원을 사용하여 생성 및 관리할 수 있으므로, 더 많은 요청을 효율적으로 처리할 수 있으며 스레드 간 메모리를 공유하기 때문에 통신 비용도 낮다고 합니다.
    - 단점: 메모리 공유로 인해 데이터 충돌이 발생할 수 있으며 이를 관리하기 위한 동기화 작업이 필요합니다. 물론 동기화 문제 관리를 하게 되면 복잡성을 증가시킬 수 있습니다.
    - 변수 및 메모리: 멀티 스레드 모델은 모든 스레드가 부모 프로세스의 메모리 공간을 공유하므로 전역 변수를 통한 데이터 공유가 가능합니다.
    - 프로그램 작동 형태: 멀티 스레드 모델은 스레드가 프로세스 내 메모리를 공유하기 때문에 하나의 스레드에서 발생한 문제가 전체 프로세스에 영향을 미칠 수 있습니다.