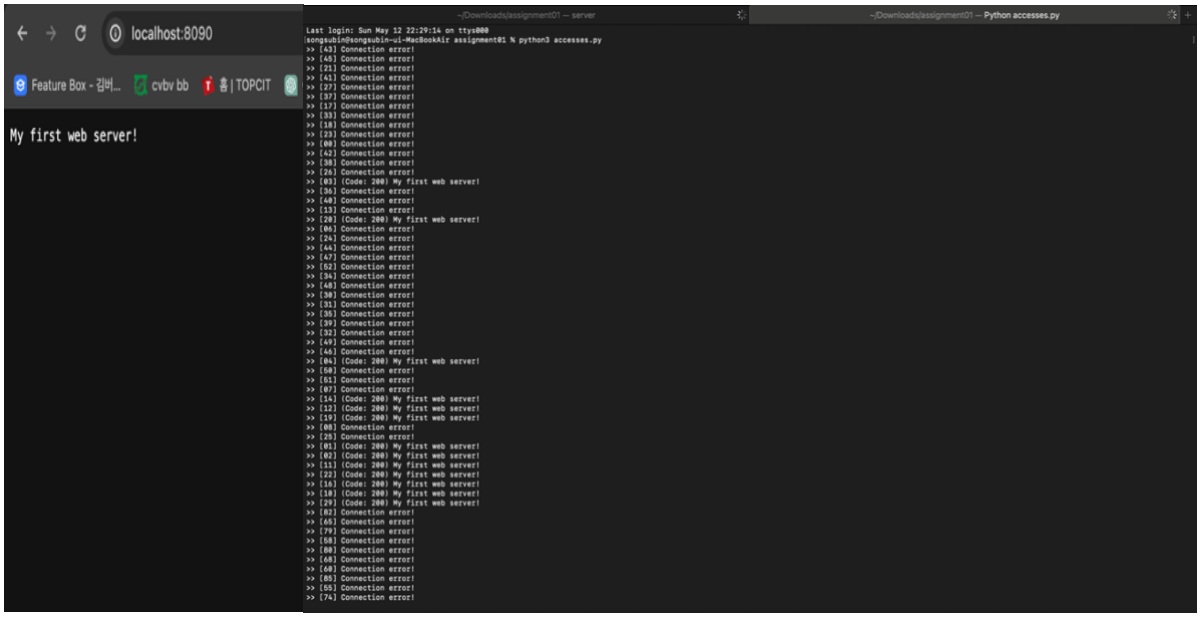
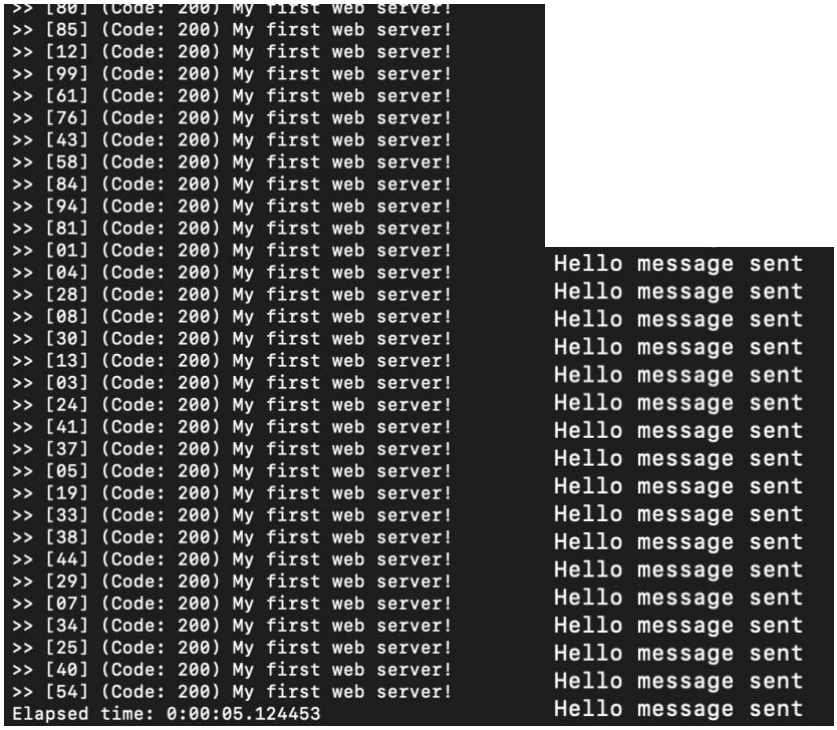
#1. My first web-server



⇨ **sleep(5) uncomment하고 실행 했을 때 중간중간 connection error가 발생했다.**



-> python3 accesses.py

[fork 추가 코드]

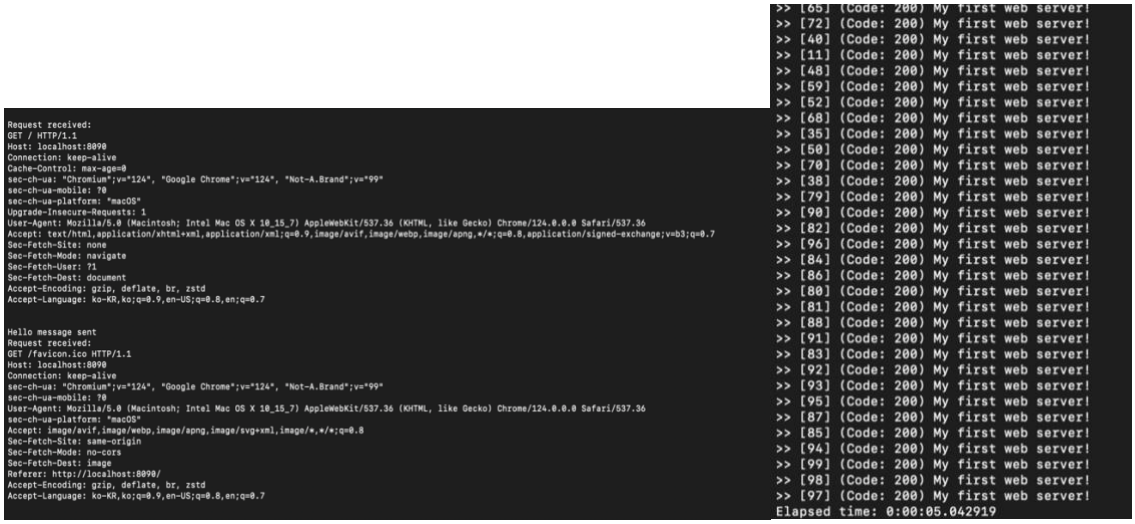
```
if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    } /*서버 소켓 생성*/
    printf("Socket created successfully\n");
```

```
address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);
    /*소켓 주소 설정해줬습니다*/
```

```
if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0)
    {
        perror("Bind failed");
        exit(EXIT_FAILURE);
    }
    printf("Bind successful\n");/*소켓 바인딩하기*/
```

```
if (pid == 0) {   //여기서부터 실행이 된다.
                close(server_fd);   /*자식 프로세스 ->  서버 소켓을 닫기*/
                handle_client(new_socket);   /*클라이언트 요청 처리*/
            } else {   // 부모 프로세스의 경우
                close(new_socket);   /* 부모 프로세스는 클라이언트 소켓을 닫기*/
                waitpid(-1, NULL, WNOHANG);   /* 비동기적으로 자식 프로세스의 종료를 기다림*/
            }
    }
```

#2. My second web server



[thread 코드 추가]

```c
#include <pthread.h>// <pthread.h> : 멀티스레드 프로그래밍을 위한 함수 제공하는 라이브러리

#define PORT 8090

void* handle_client(void* arg) {
    int new_socket = *(int*)arg;
    free(arg); // 소켓 디스크립터를 저장한 메모리 해제
    char *hello = "HTTP/1.1 200 OK\nContent-Type: text/plain" \
                  "Content-Length: 20\n\nMy first web server!";
    char buffer[30000] = {0};
    long valread = read(new_socket, buffer, 30000);
    printf("Request received:\n%s\n", buffer);
    sleep(5);
    write(new_socket, hello, strlen(hello));
    printf("Hello message sent\n");
    close(new_socket);
    return NULL;
}
```

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

```
            perror("Accept failed");
            exit(EXIT_FAILURE);
        }
        printf("Connection accepted\n");

        // 스레드를 사용하여 클라이언트 요청을 처리
        pthread_t thread_id;
        int* pclient = malloc(sizeof(int)); // 소켓 디스크립터를 저장할 메모리 할당
        *pclient = new_socket;
        if (pthread_create(&thread_id, NULL, handle_client, pclient) != 0) {
            perror("Failed to create thread");
            free(pclient); // 스레드 생성 실패 시 메모리 해제
        }
        pthread_detach(thread_id); // 스레드 분리
    }
```

#3. Orchestration and Load-balancing

```
>> [57] (Code: 200) My first web server!
>> [71] (Code: 200) My first web server!
>> [70] (Code: 200) My first web server!
>> [68] (Code: 200) My first web server!
>> [69] (Code: 200) My first web server!
>> [80] (Code: 200) My first web server!
>> [76] (Code: 200) My first web server!
>> [78] (Code: 200) My first web server!
>> [67] (Code: 200) My first web server!
>> [81] (Code: 200) My first web server!
>> [66] (Code: 200) My first web server!
>> [79] (Code: 200) My first web server!
>> [74] (Code: 200) My first web server!
>> [72] (Code: 200) My first web server!
>> [73] (Code: 200) My first web server!
>> [75] (Code: 200) My first web server!
>> [90] (Code: 200) My first web server!
>> [94] (Code: 200) My first web server!
>> [88] (Code: 200) My first web server!
>> [86] (Code: 200) My first web server!
>> [92] (Code: 200) My first web server!
>> [99] (Code: 200) My first web server!
>> [98] (Code: 200) My first web server!
Elapsed time: 0:00:20.023576
```

```
songsubin@songsubin-ui-MacBookAir assignment01 % gcc -o server server.c -lpthread
songsubin@songsubin-ui-MacBookAir assignment01 % ./server

+++++++ Waiting for new connection ++++++++

Queue size: 0
Thread count: 1
```

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

Queue size: 7
Thread count: 1
Thread added. Current thread count: 2

+++++++ Waiting for new connection ++++++++

Queue size: 8
Thread count: 2
Thread added. Current thread count: 3
```

80% 이상일 때, thread 증가되는 것을 확인할 수 있다.

```c
// Queue 구조체 정의
typedef struct {
    int *data; // 메시지 저장용 배열 포인터
    int rear, front; // 큐의 앞 뒤 인덱스
} Queue;
// 함수 원형 선언
void init(Queue *queue);
int queue_size(Queue *queue);
int enqueue(Queue *queue, int item);
int dequeue(Queue *queue);
```

queue

```c
// 큐 초기화
void init(Queue *queue) {
    // 큐의 메시지 배열을 동적으로 할당하고 -1 로 초기화하기
    queue->data = (int *)malloc(QUEUE_SIZE * sizeof(int));
    for (int i = 0; i < QUEUE_SIZE; i++) { // 배열의 모든 요소를 -1 로 초기화하기 위한 반복문
        queue->data[i] = -1;
    }
    queue->rear = queue->front = -1; // rear 와 front 인덱스를 -1 로 설정해서 큐가 비어 있음을 표시
}

// 큐의 현재 크기를 반환하는 함수
int queue_size(Queue *queue) {
    if (queue->rear >= queue->front) {
        // rear 가 front 보다 크거나 같은 경우 큐의 크기를 계산
```

```
        return queue->rear - queue->front;
    } else {
        // rear 가 front 보다 작은 경우 계산
        return QUEUE_SIZE - (queue->front - queue->rear);
    }
}

// 큐에 새로운 요소를 추가하는 함수
int enqueue(Queue *queue, int item) {
    // 큐의 숫자가 최대치면 큐가 가득 찬 상태
    if (queue_size(queue) == QUEUE_SIZE - 1) {
        printf("overflow\n");
        return -1;
    } else {
        // rear 인덱스를 한 칸 이동하고 새로운 아이템을 추가함
        queue->rear = (queue->rear + 1) % QUEUE_SIZE;
        queue->data[queue->rear] = item;
        return 1;
    }
}

// 큐에서 요소를 제거하는 함수
int dequeue(Queue *queue) {
    if (queue_size(queue) == 0) { // 큐가 비어 있는지 확인하기
        return -1; // 실패면 -1 반환
    } else {
        // front 인덱스를 한 칸 이동
        queue->front = (queue->front + 1) % QUEUE_SIZE;
        int removed_value = queue->data[queue->front];
        // 제거된 위치를 -1 로 설정하여 비어 있음을 표시
        queue->data[queue->front] = -1;
        return removed_value; // 제거된 요소 반환
    }
}
```

thread

```
// 스레드에 전달할 인자 구조체 정의
```

```c
typedef struct {
    Queue *queue;
    int *thread_count;
} ThreadArgs;

void *worker_thread(void *args) {
    pthread_detach(pthread_self()); // 스레드를 분리하여 독립적으로 실행함
    long valread;
    int client_socket;
    Queue *queue = ((ThreadArgs *)args)->queue;
    int *active_thread_count = ((ThreadArgs *)args)->thread_count;

    while (1) {
        pthread_mutex_lock(&queueMutex); // 큐 접근을 위한 뮤텍스 잠금
        if (queue_size(queue) <= 0.2 * QUEUE_SIZE && *active_thread_count > 2) { // 큐 크기가 작고 스레드 수가 2 개 이상인 경우 스레드 수 감소됨
            printf("Thread num is %d\n", *active_thread_count);
            (*active_thread_count)--;
            pthread_mutex_unlock(&queueMutex);
            break;
        }
        client_socket = dequeue(queue);
        pthread_mutex_unlock(&queueMutex);

        if (client_socket < 0) {
            usleep(10000); // 큐가 비어 있으면 10ms 대기
            continue;
        }

        char buffer[30000] = {0};
        valread = read(client_socket, buffer, sizeof(buffer));
        sleep(5); // 처리 지연을 시뮬레이션
        write(client_socket, hello, strlen(hello));
        printf("Hello message sent. Queue size: %d\n", get_queue_size(queue));
        close(client_socket);
    }
    return NULL;
}
```

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

server.c의 대략적인 동작

```
while(1) //while(1) 루프는 서버가 계속해서 클라이언트 연결을 수락하는 역할을 함
    {
        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);
        }

        char buffer[30000] = {0};
        valread = read( new_socket , buffer, 30000);
        printf("%s\n",buffer );
        // uncomment following line and connect many clients
        sleep(5);
        write(new_socket , hello , strlen(hello));
        printf("------------Hello message sent--------------");
        close(new_socket);
    }
```

HTTP 요청을 처리하는 간단한 TCP 서버의 기본 구조를 보여주는 코드이며, 클라이언트-서버 통신을 구현한 코드이다.

1번 : fork()를 사용

fork()는 부모 프로세스의 복사본인 자식 프로세스를 복제한다. If(pid==0) 코드부터 실행된다. 코드와 데이터는 동일하지만, 내용은 달라진다. 또한 각 클라이언트 요청은 별도의 프로세스로 처리한다.

- ➔ **fork() 장점** : 프로세스의생성 속도가 빠르고 추가 작업 없이 자원을 상속 가능하다. 이는 모든걸 복제하기 때문에 사용하기에 편리하다. 추가적으로 프로세스가 갑자기 오류가 나더라도 다른 프로세스에 영향을 미치지 않는다.
- ➔ **fork() 단점** : 비효율적이다. 프로세스를 생성하는데 시간이 오류 걸리고 메모리와 CPU 자원을 더 많이 사용한다. 또한 복잡하고 비용이 많이 들면서 멀티태스킹의 경우 시분할의 사이시간이 길어지는 단점이 있다.

2번 : multi-thread()를 사용

각 thread는 TCB가 생성되고, PCB에 등록된다. 각 thread는 동일한 주소 공간(프로세스의 코드, 데이터, 힙)을 공유하지만 독립적인 공간(stack)을 사용한다. 하나의 thread에서 변경된 데이터는 다른 thread에서 바로

확인이 가능하다.

- ➔ **thread() 장점** : 통신 비용이 낮으며, fork()와 비교해서 더 적은 자원을 소모한다.
- ➔ **thread() 단점** : 모든 자원을 공유하기 때문에, 하나의 thread가 잘못되면 프로세스 전체가 다 위험하다. 만약 여러 thread가 동시에 같은 데이터에 접근하거나 변경되는 경우에, 데이터 일관성에 문제가 생길 수 있다. 또한 너무 많은 thread가 있는 경우에는 교착상태가 발생할 수 있다. 즉 무한대기에 빠질 수 있다.