

[Project-1] Multithreaded TCP server

소프트웨어학과 201921085 곽수정

소프트웨어학과 201720707 나용성

1. Describe how to design your multithreaded server

- multiple clients의 concurrent connection을 accept하는 방법

기본으로 제공되는 코드인 echosrv에서는 main함수에서 listening socket을 열어서 client가 listening socket으로 connection을 요청하면 이를 accept하여 client와 1:1로 연결된 socket을 생성한 다음 직접 생성된 socket fd를 파라미터로 하여 serve_connection함수를 실행하여 연결이 끊길 때까지 직접 서비스를 제공한다.

그러나 echosrv에서는 main함수가 연결된 client가 끊기기 전까지 서비스를 직접 제공하기 때문에 다른 client가 listening socket에 연결 요청을 했을 때 이를 accept할 수 없다. 그러므로 multiple clients와 connection될 수 있도록 main함수는 client가 listening socket에 연결을 요청했을 때 이를 accept하여 client와 연결된 socket을 생성하고 생성된 socket의 socket fd를 connection queue에 push하는 역할까지만 수행하도록 한다. 그 다음 acceptor thread가 connection queue에서 socket fd를 pop한 다음 serve_connection함수를 실행하여 client의 입력을 받는 역할을 수행하도록 하여 multiple client와 연결될 수 있도록 한다. 이 때 acceptor thread는 multisrv 오브젝트 파일을 실행할 때 -a 옵션 뒤에 입력한 개수만큼 파일 실행 시 미리 생성해두는 pool-of-threads 기법을 사용했다. acceptor thread는 connection queue에 저장된 sock fd가 없을 땐 conn_cv condition variable을 wait하여 blocking상태가 되어 processor 자원을 낭비하지 않도록 한다. 이 때 main함수는 sock fd를 connection queue에 push한 다음 connection queue에 저장된 sock fd가 한개일 경우 connection을 wait하고 있는 acceptor thread가 존재하는다는 뜻이기 때문에 conn_cv condition variable로 signal을 보내서 blocking상태인 acceptor thread 하나를 깨운다. 이 때 main함수가 acceptor thread의 갯수보다 더 많은 connection을 생성하면 갯수를 초과하여 생성된 connection은 서비스를 제공할 acceptor thread가 없어 연결에 반응하지 못하기 때문에 client와 연결되지 않은 idle상태의 acceptor thread의 갯수를 유지하여 idle상태의 acceptor thread가 없을 땐 connection을 accept하기 전에 idle_acceptor condition variable을 wait하여 connection을 acceptor thread를 초과하여 생성하지 않도록 한다. 이 때 idle_thread를 wait중인 main함수는 acceptor thread가 연결이 끊겼을 때 idle_thread가 0일 땐 main함수가 wait상태일 것이기 때문에 idle_acceptor에 signal을 보내서 main함수를 깨우도록 한다.

```

while (! shutting_down) {
    idle_acceptor_wait();
    errno = 0;
    clilen = sizeof (cliaddr); /* length of address can vary, by protocol */
    if ((connfd = accept (listenfd, (struct sockaddr *) &cliaddr, &clilen)) < 0) {
        if (errno != EINTR) ERR_QUIT ("accept");
        /* otherwise try again, unless we are shutting down */
    } else {
        server_handoff (connfd); /* process the connection */
    }
}

```

그림 1. main 함수에서 listening socket을 통해 client의 연결을 accept하고 server_handoff하는 코드

기존의 echosrv에서 idle_acceptor를 wait하는 함수를 accept전에 실행한다는 점이 추가되었다.

```

void
idle_acceptor_wait(){
    pthread_mutex_lock(&conn_mutex);
    while(idle_thread == 0 && !shutting_down) pthread_cond_wait(&idle_acceptor, &conn_mutex);
    idle_thread--;
    pthread_mutex_unlock(&conn_mutex);
}

```

그림2. idle_acceptor_wait()의 코드

connection관련된 global variable을 확인하고 값을 변경하는 함수이기 때문에 conn_mutex를 lock한 다음 idle acceptor가 있는지 확인하고 없으면 idle_acceptor를 wait하는 api를 호출하고 idle acceptor가 생기고 나면 값을 하나 차감시킨 뒤 conn_mutex를 unlock하고 종료하는 과정으로 idle_acceptor를 할당받는다.

```

void
conn_signal(int sockfd){
    pthread_mutex_lock(&conn_mutex);
    conn_front = (conn_front + 1) % acceptor_num;
    conn_queue[conn_front] = sockfd;
    if((conn_rear + 1) % acceptor_num == conn_front) {
        pthread_mutex_unlock(&conn_mutex);
        pthread_cond_signal(&conn_cv);
    }
    else
        pthread_mutex_unlock(&conn_mutex);
}

void
server_handoff (int sockfd) {
    conn_signal(sockfd);
}

```

그림 3. server_handoff함수와 server_handoff가 실행하는 conn_signal함수의 코드

server_handoff는 sockfd를 conn_queue에 push하는 함수인 conn_signal함수를 호출하여 handoff를 수행한다. conn_signal은 sockfd를 conn_queue에 push한 다음 push하기 전에 queue가 empty상태였다면 conn_cv condition variable에게 signal을 보내 acceptor thread를 깨운다. 이 때 idle thread가 있음을 위의 idle_acceptor_wait함수에서 확인했기 때문에 conn_queue가 full인지 확인하는 과정은 필요하지 않다.

```

void*
acceptorThread(void *arg)
{
    int sockfd;
    while(!shutting_down)
    {
        sockfd = conn_wait();
        if(shutting_down)
            break;
        serve_connection(sockfd);
        if(shutting_down)
            break;
        idle_acceptor_signal();
    }
    pthread_exit(NULL);
}

void
idle_acceptor_signal(){
    pthread_mutex_lock(&conn_mutex);
    idle_thread++;
    if(idle_thread == 1) {
        pthread_mutex_unlock(&conn_mutex);
        pthread_cond_signal(&idle_acceptor);
    }
    else
        pthread_mutex_unlock(&conn_mutex);
}

```

그림4. acceptorThread함수와 client와 연결이 끊어진 뒤 idle이 흄을 알리는 함수
idle_acceptor_signal()함수의 코드

acceptorThread는 sockfd를 conn_wait함수로 가져온 다음 serve_connection로 client와 통신한 다음 연결이 끊긴 뒤엔 idle_acceptor_signal로 idle 상태가 됐음을 알린다.
idle_acceptor_signal함수는 idle_thread에 1을 더한 다음 이 thread가 idle상태인 첫 thread일 때 idle_acceptor에 signal을 보내 main함수를 깨운다.

```

int
conn_wait(){
    int sockfd;
    pthread_mutex_lock(&conn_mutex);
    while(conn_front == conn_rear && !shutting_down) pthread_cond_wait(&conn_cv, &conn_mutex);
    if(shutting_down){
        pthread_mutex_unlock(&conn_mutex);
        return 0;
    }
    conn_rear = (conn_rear + 1) % acceptor_num;
    sockfd = conn_queue[conn_rear];
    pthread_mutex_unlock(&conn_mutex);
    return sockfd;
}

```

그림5. conn_wait()함수의 코드

conn_wait함수는 conn_queue가 empty상태가 아닐 때까지 기다린 다음 sockfd를 conn_queue에서 가져온 다음 return하는 과정으로 구현되어 있다.

```

while (! shutting_down) {
    if ((n = readline (&conn, line, MAXLINE)) == 0) goto quit;
    /* connection closed by other end */
    if (shutting_down) goto quit;
    if (n < 0) {
        perror ("readline failed");
        goto quit;
    }
    number = atoi(line);
    work_push(&conn, number);
}

```

그림6. serve_connection함수의 client의 입력을 받은 다음의 변경점

serve_connection함수는 이제 입력을 받은 뒤 바로 work_queue에 push하는 기능 까지만 수행한다.

- client의 요청을 최대한 많이 빠르게 처리하는 방법에 대한 설명

multiple client와의 concurrent connection은 위에서 여러개의 acceptor threads를 통해 구현한 방법에 대해 설명했다. 이 때 client들과 연결된 각각의 acceptor thread에서 client의 입력을 받았을 때 바로 소수판별을 하여 결과를 직접 전송하는 방법도 가능하겠지만 이렇게 구현했을 땐 한 client의 요청을 하나의 thread가 모두 처리하게된다. 그러나 acceptor thread에서는 입력만 받고 입력에 대한 처리는 여러개의 worker thread가 처리하도록 하면 한 client의 요청을 여러 thread가 처리하게 되는 장점이 있다.

위의 방식을 구현하기 위해 우리가 구현한 multisrv에서는 multisrv오브젝트 파일을 실행할 때 acceptor thread와 비슷한 방식으로 -w 옵션 뒤에 입력한 개수만큼 파일 실행 시 미리 worker thread들을 생성해두는 pool-of-threads기법을 사용했다. worker thread가 처리할 work는 acceptor thread가 client의 요청을 구조체 work의 형태로 변환하여 work queue에 push한 것을 pop하여 처리하게 된다. 이 때 구조체 work의 멤버에는 acceptor thread가 입력받은 2이상의 자연수 number와 number를 전송한 client과의 연결의 정보를 담은 connection_t 변수의 주소값을 저장한 connection_t의 포인터 변수 *conn을 갖는다. number는 worker thread가 소수 여부를 판별하기 위해 필요하며 *conn은 판별 결과를 client에게 전송할 때 필요하다.

Acceptor thread는 client에게서 숫자를 입력받으면 바로 conn의 주소와 입력받은 숫자로 work구조체를 생성해 work_queue에 push한다. push명령을 내리면 work_queue에 빈 공간이 있는지 확인하고 빈 공간이 없다면 work_empty condition variable을 wait한다. push한 다음 work_queue에 들어있는 work가 1개라면 비어있던 work_queue에 push한 것이기 때문에 worker thread들이 wait상태일 것이므로 work_full condition variable에 signal을 보내 worker thread 하나를 깨워 work를 처리하도록 하게 한다. worker thread는 먼저 work를 get하고자 시도하는데 work를 get하고자 하면 work_queue에 work가 존재하는지 확인하고 존재하지 않는다면 work_full condition variable을 wait한다. 이미 work가 존재하거나 wait상태에서 signal을 받아 깨 다음 work를 pop하고 나서 만약 pop하기 전의 work queue가 full상태였다면 acceptor thread중에 work queue가 full상태여서 work를 더 push하지 못하여 work_empty condition variable을 wait중인 acceptor thread가 존재할 수 있으므로 work_empty로 signal을 보내 acceptor thread를 깨워준다. worker thread는 work를 work_queue에서 pop한 다음 is_Prime(int number)함수를 이용해 number의 소수여부를 판별하고 *conn을 이용해 client에게 판별결과 문장을 전송하는 것을 반복하면서 work를 처리한다. 위와 같은 과정을 여러 worker thread가 수행함으로써 동시에 여러개의 client의 요청을 처리할 수 있다.

2. 성능 측정

- acceptor thread의 개수에 따른 concurrent하게 연결될 수 있는 client의 개수 측정

서버가 동시에 연결되어 서비스를 제공할 수 있는 client의 개수는 multisrv를 실행할 때 옵션으로 지정한 acceptor thread의 개수와 동일했다. 그 이상의 연결을 시도했을 경우엔 listening socket이 listen api에서 입력된 큐의 길이 파라미터에 + 1 만큼의 client까지는 서버에 연결은 됐지만 서비스는 제공받지 못하는 상태에 있었다. 이 때 큐의 길이 + 1인 이유는 acceptor thread들이 모두 client와 연결된 다음 첫 client가 바로 listening socket의 큐에서 대기하는 것이 아닌 이 client까지는 listening socket 내부에서 accept를 대기하고 다음 client부터 큐에서 대기하기 때문에 큐에서 대기하는 client들 + listening socket 내부에서 accept를 대기하는 client까진 서버에 연결은 된 상태인 것으로 보인다. 그 뒤로 연결을 시도한 client는 서버에 연결된 다른 client가 연결을 끊기까지 응답을 받지 못했다.

```
st201720707@sce394-1:~/proj1$ SERVERHOST=localhost ./echocli 56823
connection wait time = 555 microseconds
```

```
st201720707@sce394-1:~/proj1$ SERVERHOST=localhost ./echocli 56823
connection wait time = 64877958 microseconds
```

그림7. 서버에 연결될 공간이 남아있을 때 연결을 시도한 client(위)와 listening socket의 queue까지 full상태가 되어 연결되지 못해 대기중이다가 연결된 client(아래)의 대기시간 비교

- worker thread의 개수에 따른 client의 request의 처리 속도 측정

먼저 측정을 하기 전에 client는 terminal을 통해 입력한 숫자 n개 만큼 random하게 2 이상의 숫자를 생성하여 서버에 n번 보낸 다음 n번 서버에게서 결과를 읽어오고 보내기 시작한 순간부터 결과를 모두 받은 순간까지의 시간을 측정하여 처리 속도를 측정했다.

worker thread	1	5	10	20	50	100
시간경과 (ms)	234795	227971	232636	232175	224863	223967

위와 같은 결과를 얻었는데 처리속도가 개선되지 않은 것처럼 보인다. 그래서 우리는 위와 같은 측정이 된 원인을 생각해보았는데 먼저 소수 판별에 필요한 overhead가 그리 크지 않은데 mutex를 lock하고 unlock하는 과정에서의 serial한 부분의 비율이 과도하게 클 것으로 예상되며 이를 개선할 방법으로 work하나에 대해 mutex를 lock unlock하지 않고 work를 모두 push한 다음 signal을 보내는 방법이 있을 것으로 보이며 또 client는 결과를 serial하게 수신하기 때문에 시간이 차이나지 않을 수 있다.