Socket Programming Project

수학과 김호진 (2016314786)

[Serialized tokens that received from the server]

구현한 IPv6 Server의 Thread 안에 토큰 값을 출력하는 코드를 추가하여, IPv6 Clients로부터 전달받은 토큰 값들을 확인해보았습니다.

```
84  void *thread_action(void *arg) {
85      int receive_sockfd = *((int*)arg);
86      int transmit_sockfd;
87      struct sockaddr_in client_v4;
88      struct hostent* hp;
89      char buffer[BUFFER_LEN];
90
91      while (1) {
92          /* Read data(token) sent from IPv6 Clients */
93          /* Close the connection as soon as receive the data so that the number of connections does not exceed 3 */
94          if (read(receive_sockfd, buffer, sizeof(buffer)) <= 0) {
95              close(receive_sockfd);
96              client_number--;
97              break;
98          }
99
100         /* Output the received token */
101         printf("%s", buffer);
```

프로그램을 실행한 결과, 다음의 토큰 값들을 받았음을 확인했습니다.

```
borussen@DESKTOP-L834KLC: /mnt/c/Programming/CN_03
borussen@DESKTOP-L834KLC:/mnt/c/Programming/CN_03$ ./server 14786
RANDOM1:47530296566750645675

RANDOM2:56446850633532340473

RANDOM3:53517947953298835975

RANDOM4:97538365916797553788

RANDOM5:34976484361022912533
```

- RANDOM1:47530296566750645675
- RANDOM2:56446850633532340473
- RANDOM3:53517947953298835975
- RANDOM4:97538365916797553788
- RANDOM5:34976484361022912533

[Setup to connect to the IPv4 Client and IPv6 Server]

- **IPv4 Client**

  IPv4 Client 구현에 앞서, 다음과 같이 변수들을 선언합니다.

```
11      #define SERVER_TCP_PORT 50000
12      #define BUFFER_LEN 256
13
14      int main(int argc, char **argv) {
15          int n, sockfd, new_sockfd, port_number;
16          int client_len, new_sd;
17          struct hostent *server, *client;
18          struct sockaddr_in server_v4, client_v4;
19          char buffer_1[BUFFER_LEN], buffer_2[BUFFER_LEN];
20
21          switch (argc) {
22          case 2:
23              port_number = SERVER_TCP_PORT;
24              break;
25          case 3:
26              port_number = atoi(argv[2]);
27              break;
28          default:
29              fprintf(stderr, "Usage: %s host [port]₩n", argv[0]);
30              exit(1);
31          }
```

  그 다음으로 IPv4 Server와의 연결에 사용할 소켓을 생성합니다.

```
33          /* Create a stream socket to connect with IPv4 Server */
34          if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
35              fprintf(stderr, "ERROR: Cannot create a socket₩n");
36              exit(1);
37          }
```

  IPv4 Server에 연결 요청을 하기 위해서 IPv4 Server의 주소와 포트번호 등을 구해줍니다.

```
39          /* Get IPv4 Server's address */
40          if ((server = gethostbyname(argv[1])) == NULL) {
41              fprintf(stderr, "ERROR: Cannot get server's address₩n");
42              exit(1);
43          }
44          bzero((char*)&server_v4, sizeof(struct sockaddr_in));
45          server_v4.sin_family = AF_INET;
46          server_v4.sin_port = htons(port_number);
47          bcopy(server->h_addr, (char*)&server_v4.sin_addr, server->h_length);
```

이렇게 구한 값들을 이용하여 IPv4 Server에 연결 요청을 보냅니다.

```
49      /* Request connection to IPv4 Server */
50      if (connect(sockfd, (struct sockaddr*)&server_v4, sizeof(server_v4)) == -1) {
51          fprintf(stderr, "ERROR: Cannot connect₩n");
52          exit(1);
53      }
```

이와 동시에 IPv6 Server로부터의 연결 요청을 수신하기 위한 새로운 소켓을 만듭니다. IPv6 Server와 IPv4 Client 간 통신에서는 IPv6 Server가 Client 역할을 하고, IPv4 Client가 Server 역할을 합니다.

```
55      /* Create a new stream socket to receive a connection request from IPv6 Server */
56      if ((new_sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
57          fprintf(stderr, "ERROR: Cannot create a socket₩n");
58          exit(1);
59      }
```

새로운 소켓은 서버의 역할을 하기 때문에 bind()를 통해 소켓에 IP 주소와 포트번호 등을 지정해줍니다. 저의 경우, 학번 앞 다섯자리(20163)을 포트번호로 지정했습니다.

```
61      /* Bind an address to the new stream socket */
62      bzero((char*)&server, sizeof(struct sockaddr_in));
63      client_v4.sin_family = AF_INET;
64      client_v4.sin_port = htons(20163);
65      client_v4.sin_addr.s_addr = htonl(INADDR_ANY);
66      if (bind(new_sockfd, (struct sockaddr*)&client_v4, sizeof(client_v4)) == -1) {
67          fprintf(stderr, "ERROR: Cannot bind name to socket₩n");
68          exit(1);
69      }
```

listen()을 이용해, IPv6 Server로부터 보내지는 연결 요청을 5개까지 기다리게 합니다.

```
71      /* Queue up to 5 connect requests */
72      listen(new_sockfd, 5);
```

다시 IPv4 Client와 IPv4 Server 간 통신으로 돌아와서, 서로 데이터를 주고받습니다.

```
74      /* Communicate between IPv4 Server and IPv4 Client */
75      while (n = read(sockfd, buffer_1, sizeof(buffer_1)) > 0) {
76          printf("%s", buffer_1);
77          memset(buffer_1, 0x00, sizeof(buffer_1));
78
79          read(sockfd, buffer_1, sizeof(buffer_1));
80          printf("%s", buffer_1);
81          memset(buffer_1, 0x00, sizeof(buffer_1));
82
83          read(0, buffer_1, sizeof(buffer_1));
84          write(sockfd, buffer_1, strlen(buffer_1));
85          if (strncmp(buffer_1, "OK", 2) == 0) {
86              memset(buffer_1, 0x00, sizeof(buffer_1));
87              break;
88          }
89          memset(buffer_1, 0x00, sizeof(buffer_1));
90      }
```

IPv4 Client와 IPv4 Server 간 통신이 정상적으로 이뤄져서 정보를 성공적으로 전달하면, IPv4 server는 IPv6 clients를 생성합니다. 이렇게 생성된 IPv6 clients는 전달받은 주소를 통해 IPv6 Server에 접근하여 랜덤 토큰 값을 전달합니다. IPv6 Server는 IPv4 Client에게 전달받은 토큰 값을 보내기 위해서 IPv4 연결을 요청합니다. IPv4 client는 accept()를 통해 연결 요청을 받아들이고 IPv6 Server와의 통신에 사용할 새로운 소켓을 만듭니다.

```
93      memset(buffer_2, 0x00, sizeof(buffer_2));
94      while (1) {
95          /* Accept the connection request sent by IPv6 Server */
96          /* Create a new stream socket to connect with IPv6 Server */
97          if ((new_sd = accept(new_sockfd, (struct sockaddr*)&client, &client_len)) == -1) {
98              fprintf(stderr, "ERROR: Cannot accept client\n");
99              exit(1);
100         }
```

read()를 이용해, IPv6 Server로부터 전달되는 토큰 값들을 받아옵니다.

```
102         /* Read data(token) sent from IPv6 Server */
103         if (read(new_sd, buffer_2, sizeof(buffer_2)) <= 0) {
104             close(new_sd);
105             break;
106         }
```

이렇게 전달받은 토큰 값들은 write()를 이용하여 IPv4 Server로 보냅니다. 마지막에 '0x0a'를 추가하여 IPv4 Server가 입력에 반응할 수 있도록 합니다.

```
108         /* Write recieved data(token) to IPv4 Server */
109         if (strncmp(buffer_2, "RANDOM5", 7) == 0) {
110             write(sockfd, buffer_2, 28);
111             write(sockfd, "\n", 1);
112             break;
113         }
114         write(sockfd, buffer_2, 28);
115         write(sockfd, ",", 1);
116         close(new_sd);
```

IPv4 Server로부터 모든 토큰 값들이 정상적으로 전달되었는지 확인을 받은 다음, 통신을 종료합니다.

```
119     /* Receive a success message from IPv4 Server */
120     while (n = read(sockfd, buffer_1, sizeof(buffer_1)) > 0) {
121         printf("%s", buffer_1);
122         memset(buffer_1, 0x00, sizeof(buffer_1));
123
124         read(sockfd, buffer_1, sizeof(buffer_1));
125         printf("%s", buffer_1);
126     }
127
128     close(sockfd);
129     return(0);
130 }
```

■ **IPv6 Server**

IPv6 Server 구현에 앞서, 다음과 같이 변수들을 선언해줍니다. IPv6 Server 포트 번호의 경우, 학번 뒤 다섯자리(14786)로 설정했습니다.

```
10  #define SERVER_TCP_PORT 14786
11  #define BUFFER_LEN      256
12
13  void *thread_action(void *data);
14  int client_number = 0;
15
16  int main(int argc, char **argv) {
17      int port_number, sockfd, new_sockfd, client_len;
18      struct sockaddr_in6 server_v6, client_v6;
19      pthread_t thread[BUFFER_LEN];
20
21      switch (argc) {
22      case 1:
23          port_number = SERVER_TCP_PORT;
24          break;
25      case 2:
26          port_number = atoi(argv[1]);
27          break;
28      default:
29          fprintf(stderr, "Usage: %s [port]₩n", argv[0]);
30          exit(1);
31      }
```

제일 먼저 IPv6 Clients의 연결 요청을 수신하는데 사용할 소켓을 생성합니다.

```
33      /* Create a stream socket to connect to receive a connection request from IPv6 Clients */
34      if ((sockfd = socket(AF_INET6, SOCK_STREAM, 0)) == -1) {
35          fprintf(stderr, "ERROR: Cannot create a socket₩n");
36          exit(1);
```

bind()를 이용하여 소켓에 IP 주소와 포트번호 등을 지정해줍니다. IPv6 주소체계를 따라야 하므로 sockaddr_in6, AF_INET6 등을 사용합니다.

```
39      /* Bind an address to the socket */
40      bzero((char*)&server_v6, sizeof(struct sockaddr_in6));
41      server_v6.sin6_family = AF_INET6;
42      server_v6.sin6_flowinfo = 0;
43      server_v6.sin6_port = htons(port_number);
44      server_v6.sin6_addr = in6addr_any;
45      if (bind(sockfd, (struct sockaddr*)&server_v6, sizeof(server_v6)) == -1) {
46          fprintf(stderr, "ERROR: Cannot bind name to socket₩n");
47          exit(1);
48      }
```

listen()을 이용해, IPv6 Clients로부터 보내지는 연결 요청을 5개까지 기다리게 합니다.

```
50      /* Queue up to 5 connect requests */
51      if (listen(sockfd, 5) == -1) {
52          fprintf(stderr, "ERROR: Cannot listen₩n");
53          exit(1);
54      }
```

IPv6 Server는 accept()를 통해 IPv6 Clients에서 보낸 연결 요청을 받아들이고, IPv6 Clients와의 통신에 사용할 새로운 소켓을 만듭니다.

```
56    while (1) {
57        /* Accept the connection request sent by IPv6 Clients */
58        /* Create a new stream socket to connect with IPv6 Clients */
59        client_len = sizeof(client_v6);
60        if ((new_sockfd = accept(sockfd, (struct sockaddr*)&client_v6, &client_len)) == -1) {
61            fprintf(stderr, "ERROR: Cannot accept client₩n");
62            exit(1);
63        }
```

5개의 IPv4 Clients가 여러 프로세스에서 동시에 연결 요청을 보내기 때문에, 이를 처리하기 위해서 Thread-based concurrent server를 구축합니다. 서버가 동시에 처리할 수 있는 클라이언트 수가 최대 3개를 넘어가지 않도록 예외처리를 해줍니다.

```
65        /* Keep the number of clients that the IPv6 Server can handle at the same time at 3 or less */
66        if (client_number > 3) {
67            continue;
68        }
69
70        /* Implement thread-based concurrent server */
71        if (pthread_create(&thread[client_number], NULL, thread_action, (void*)&new_sockfd) == -1) {
72            fprintf(stderr, "ERROR: Cannot create Thread₩n");
73            close(new_sockfd);
74            continue;
75        }
76        client_number++;
77    }
```

Thread의 동작을 구현하기 위해서, 변수들을 다음과 같이 정의했습니다.

```
84  void *thread_action(void *arg) {
85      int receive_sockfd = *((int*)arg);
86      int transmit_sockfd;
87      struct sockaddr_in client_v4;;
88      char buffer[BUFFER_LEN];
```

Thread는 IPv6 Clients로부터 토큰을 전달받습니다. 서버가 동시에 처리할 수 있는 클라이언트 수가 최대 3개를 넘어가지 않도록 하기 위해 토큰을 받는 즉시 연결을 종료합니다

```
91    while (1) {
92        /* Read data(token) sent from IPv6 Clients */
93        /* Close the connection as soon as receive the data so that the number of connections does not exceed 3 */
94        if (read(receive_sockfd, buffer, sizeof(buffer)) <= 0) {
95            close(receive_sockfd);
96            client_number--;
97            break;
98        }
```

앞서 언급했듯이, 출력을 통해 받은 토큰 값들을 확인합니다.

```
100       /* Output the received token */
101       printf("%s", buffer);
```

IPv4 Client에 토큰 값들을 전달하기 위해서 연결을 요청할 새로운 소켓을 생성합니다.

```
103         /* Create a new stream socket to connect with IPv4 Client */
104         if ((transmit_sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
105             fprintf(stderr, "ERROR: Cannot create a socket₩n");
106             exit(1);
```

IPv4 Client에 연결요청을 하기 위해서 IPv4 Client의 주소와 포트번호 등을 구합니다. IPv6 Server와 IPv4 Client는 동일한 컴퓨터에서 작동하기 때문에 IPv4의 special address 중 하나인 Loop back address (127.0.0.1)를 활용합니다.

```
109         /* Get IPv4 Client's address */
110         bzero((char*)&client_v4, sizeof(struct sockaddr_in));
111         client_v4.sin_family = AF_INET;
112         client_v4.sin_port = htons(20163);
113         inet_pton(AF_INET, "127.0.0.1", &client_v4.sin_addr);
```

이렇게 구한 값들을 이용하여 IPv4 Client에 연결요청을 보냅니다.

```
115         /* Request connection to IPv4 Client */
116         if (connect(transmit_sockfd, (struct sockaddr*)&client_v4, sizeof(client_v4)) == -1) {
117             fprintf(stderr, "ERROR: Cannot connect₩n");
118             exit(1);
119         }
```

연결요청이 수락되면, IPv4 Client로 전달받은 토큰 값들을 모두 전송한 뒤 통신을 종료합니다.

```
121         /* Write recieved data(token) to IPv4 Client */
122         write(transmit_sockfd, buffer, BUFFER_LEN);
123
124         if (strncmp(buffer, "RANDOM5", 7) == 0) {
125             break;
126         }
127     }
128     close(transmit_sockfd);
129     close(receive_sockfd);
130 }
```

[Unique experience]

- 프로젝트를 진행하는 과정에서 bind error가 종종 발생하는 것을 확인했습니다. 이를 해결하고자 문제가 발생하는 원인에 대해 알아봤습니다. bind() 함수를 사용하면 로컬 프로토콜, 로컬 IP 주소, 로컬 포트번호 등이 생성됩니다. 그런데 이러한 자료구조는 프로그램이 종료되더라도 메모리에서 곧바로 해지되지 않고, 스케줄링 정책에 의해서 일정 기간이 지나야만 해제가 됩니다. 만약 연결을 해제하는데 소요되는 시간 내에 똑같은 bind 요청이 들어오게 되면, bind error를 통해 이전 작업이 완료되지 않았다는 것을 알리게 됩니다. 그러므로 bind error가 발생하는 경우, 시간 간격을 둔 뒤 연결을 다시 요청하면 대부분 해결됩니다.

- IPv4와 IPv6의 기본적인 소켓 프로그래밍의 경우, BSD 소켓을 기반으로 동일하게 진행됩니다. 그런데 프로젝트 과정 중 IPv4 Client와 IPv6 Server를 구현해보면서 IPv4와 IPv6의 socket programming에 사용되는 API에 약간의 차이가 존재한다는 것을 알게 되었습니다. 이에 해당하는 대표적인 예시들을 찾아보니 AF_INET6과 sockaddr_in6 구조체가 있었습니다. 32-bit 주소체계를 갖는 IPv4와 다르게 IPv6는 128-bit 주소체계를 가지기 때문에 기존의 IPv4 주소 구조체인 sockaddr_in을 그대로 사용할 수 없게 되었습니다. 특히 sockaddr_in은 가지고 있는 여유 저장공간이 매우 부족했기 때문에 기존의 구조체를 확장하여 사용할 수 없었고, 이로 인해 새로운 구조체를 만들게 되었습니다. 이렇게 만들어진 sockaddr_in6의 멤버 변수 중에는 인터페이스를 가리키는 인덱스 값을 의미하는 sin6_scope_id가 존재합니다. 오늘날에는 IPv4와 IPv6가 함께 사용되기 때문에 하나의 컴퓨터가 다양한 인터넷 주소 환경을 가질 수 있게 되었으며, 각각의 인터페이스마다 다양한 주소 체계를 가지게 되었습니다. 그렇기 때문에 소켓을 bind 할 때 어떠한 인터페이스를 사용할 지 정확하게 명시하기 위해서 sin6_scope_id가 사용됩니다.

- Concurrent Server를 구현하는 대표적인 방법에는 프로세스와 스레드를 활용하는 방안이 있습니다. Concurrent Server를 구현하기에 앞서 각각이 가지는 장단점에 대해 알아보았습니다. 프로세스는 운영체제 위에서 실행하는 프로그램으로, 운영체제로부터 다른 프로세스의 접근이 불가능한 독립된 메모리 영역을 할당 받습니다. 그에 반해 스레드는 하나의 프로세스 내에서 동작하는 여러 실행들의 흐름으로, 프로세스의 자원을 공유하면서 일련의 과정을 여러 개로 동시에 실행시킬 수 있습니다. 멀티 프로세스의 경우, 다른 프로세스에 서로 영향을 끼치지 않기 때문에 자식 프로세스 중 하나가 죽더라도 정상적인 수행을 할 수 있다는 장점이 있지만, 멀티 스레드에 비해 많은 메모리 공간과 CPU 시간을 차지하고 context switching의 오버헤드가 발생한다는 문제를 가지고 있습니다. 멀티 스레드의 경우에는 상대적으로 작은 메모리 공간 차지하고 context switch가 빠르다는 장점이 있습니다. 또한 스레드 간 통신 시 전역변수 공간이나 Heap 영역을 통해 데이터 주고받을 수 있어 통신부담이 적습니다. 하지만, 스레드 간에 자원을 공유하기 때문에 하나의 스레드 오류로 인해 전체가 종료될 위험이 있고 동기화 문제 역시 존재합니다. 저는 저장 공간을 공유하는 스레드가 토큰을 전달하는데 있어 더 용이할 것이라고 판단했고 프로그래밍이 상대적으로 복잡하지 않기 때문에 스레드에 문제가 발생하거나 동기화 문제가 나타나는 경우가 거의 없을 것이라고 생각해 Thread-based Concurrent Server를 구현했습니다.

[Screenshot of the successful conversation]

IPv4 Client가 IPv6 Server로부터 전달받은 다섯 개의 토큰 값을 IPv4 Server에게 올바르게 전송하였고, IPv4 Server로부터 "Result: Concurrent" 문자열을 수신 받으면서 모든 과정이 성공적으로 진행되었음을 확인했습니다.