2019년 2학기 시스템 프로그래밍 2차 Warm-up 과제

< pthread를 이용한 client-side socket programming >

주제 pthread를 이용한 client-side socket programming

제출일 2019. 11. 14. (목)

제출자 2017320168 컴퓨터학과 김민수 (팀장)

2017320192 컴퓨터학과 최원미

환경 Oracle VM VirtualBox, ubuntu 16.04 LTS, kernel: linux-4.4.0

Freeday 0일

목차

1. 목적

client-side에서 server와 통신할 수 있는 socket 통신 프로그램을 작성하여, 5개의 blocking socket으로 동시에 패킷(데이터)을 수신하는 application을 작성하는 것이다.

2. pthread의 정의 및 사용 이유

1) pthread의 정의

pthread는 모든 유닉스 계열 POSIX 시스템에서 병렬적으로 작동하는 소프트웨어 작성을 위해서 제공되는 표준 API이다. 기본적으로 thread를 사용하기 위해 pthread를 사용한다. 따라서 pthread를 사용하는 이유를 알기 위해서는 thread에 대해서 알아야 한다.

Thread는 프로세스 내에서 실행되는 여러 흐름의 단위이다. 여기서 흐름은 실행이라고 할 수 있다. 이런 thread의 장점은 다음과 같다.

- **Thread 간 자원 공유를 통한 성능 향상**

  Thread는 한 프로세스 내에 존재하고, 해당 프로세스의 code, data, heap 영역을 서로 공유한다. 따라서 프로세스를 여러 CPU를 통해 실행하는 것과는 다르게 coherency issue가 발생하지 않으며, IPC와 같은 복잡한 작업 없이 thread 간 communication이 가능하므로 프로세스 실행 시 처리 속도가 빠르다.

- **Context switching의 overhead 감소**

  Thread들은 한 프로세스 내에서 서로 code, data, heap 영역을 공유한다. 따라서 context switching이 발생하는 경우에는 서로 공유하지 않는 영역인 stack만 저장하면 되므로, overhead가 크게 감소한다.

- **병렬처리**

  기본적으로 thread는 한 프로세스 내에서 각자의 흐름을 갖기 때문에, 한 프로세스를 병렬적으로 실행하게 된다. 이런 특성 때문에 thread를 사용해서 실행한 프로세스의 실행 순서는 예측하기 어렵고, 동시에 병렬적으로 처리하므로

더 빠르게 프로세스를 실행할 수 있게 된다.

2) pthread를 사용한 이유

상술한 장점들 중, 이번 과제에서 pthread를 사용한 이유는 **병렬처리**를 위해서이다. 이를 설명하기 위해, 먼저 server-side application의 코드를 보자.

```
//send
while (1) {
    i = rand() % 5;
    //send 횟수
    r = rand() % 10;
    for (int k = 0; k < 1 + (rand() % 50 == 7 ? 10000: 0); k++) {
        //printf("send from port: %d!\n", servPorts[i]);
        for (int j = 0; j < presetMSize[i][r]; j++) message[j] = (rand() % 26) + 'A';
        send(hClntSock[i], message, presetMSize[i][r], 0);

        usleep(rand() % 10);
    }
}
```

Figure 1. 서버에서 데이터를 전송하는 코드

Server에서 데이터를 전송할 때, random한 port를 선택해서 데이터를 전송한다 (위 코드의 변수 i). 이 경우 만약 client-side에서 pthread를 사용하지 않고 프로그램을 실행한다면, 5개의 socket 중 어느 port에 메세지가 들어올 지 알지 못한 채로 기다려야 된다. 반면 pthread를 통해 client-side 프로그램을 실행한다면 각 port가 병렬적으로 서버의 데이터를 기다리게 되므로, random한 port에 데이터가 전송된다 하더라도 서버에서 전송하는 즉시 데이터를 수신할 수 있게 된다. 결과적으로, 단지 프로세스를 병렬적으로 실행시켜서 높은 성능을 기대하는 것뿐만 아니라, 어느 port에서 들어오는지 모르는 상황에서도 유연하게 대처할 수 있게 된다. 이런 이유로 pthread를 사용하는 것이다.

3. 작성한 코드 설명

코드는 C언어로 작성되었으며, include한 header 파일들은 생략하고 설명할 것이다. 먼저 가독성과 코드 실행에 도움을 주는 코드이다.

```c
#define N_OF_PORT    5        // # of port = 5
#define MSG_SIZE     65536    // max size of received message

struct _pthreadArgs {
    int*     clientSocket;
    int      portNumber;
};
typedef struct _pthreadArgs PThreadArgs;
```

Figure 2. #define 및 pthread를 운용하기 위한 구조체

N_OF_PORT는 client-side의 port 개수이고, MSG_SIZE는 server 코드에서 정의된 메시지 길이의 최대 크기와 같게 두었다. _pthreadArgs 구조체는 pthread를 통해서 실행하는 함수에서 사용하는 parameter들을 넣어 주기 위해 정의하였다.

아래는 main함수의 상단부이다.

```c
int main(int argc, char* argv[]) {

    int      clientSocket[N_OF_PORT];     // client socket
    int      portNumbers[N_OF_PORT];      // port number for each sockets
    char     serverIp[20];                // server ip address
    struct sockaddr_in clientAddress[N_OF_PORT];    // address for each client
    struct sockaddr_in serverAddress;               // server address

    pthread_t   pThread[N_OF_PORT]; // thread
    char        message[MSG_SIZE];  // received message from server
    int         status;

    int i;  // index

    printf("Enter server IPv4 address: ");
    scanf("%s", serverIp);
```

Figure 3. main함수의 변수를 정의한 부분

clientSocket 변수는 client-side에서 socket을 구현하기 위해 생성한 변수이다. socket 함수를 통해 생성된 file descriptor 형태의 socket을 저장한다. portNumber 변수는 직접 정한 port number를 저장하는 변수이고, 총 5개의 포트 번호를 저장한다. serverIp 변수는 사용자의 편의를 위해 생성한 변수이고, Figure 3의 가장 아래에 scanf로 받는 서버의 ip를

입력 받는다. clientAddress와 serverAddress는 각 socket의 주소를 저장할 때 필요한 변수이다. pThread 변수는 pthread를 위한 변수이고, 총 5개의 pthread를 생성한다. message는 서버로부터 넘어온 메시지를 저장하는 변수이다. status는 pthread_join 함수에서 사용한 변수이다. 또한 scanf를 통해 server의 ip를 직접 입력하도록 하였다.

아래는 main 함수의 하단부이다.

```
// using port number 1111, 2222, 3333, 4444, 5555
init5Ports(portNumbers, 1111);

// connect each sockets to the server
for (i=0; i<N_OF_PORT; ++i) {
    printf("Processing port %d...\n", portNumbers[i]);

    // 1. configure a socket
    configureSocket(&clientSocket[i], portNumbers[i]);

    // 2. assigning a name to a socket
    socketNaming(&clientAddress[i], clientSocket[i], portNumbers[i]);

    // 3. configure server
    configureServer(&serverAddress, serverIp, portNumbers[i]);

    // 4. connect each socket to the server
    connect2Server(&serverAddress, clientSocket[i], portNumbers[i]);

    printf("Port %d process done!\n\n", portNumbers[i]);
}

// threads
for (i=0; i<N_OF_PORT; ++i) {
    // create thread
    createThread(&pThread[i], &clientSocket[i], portNumbers[i]);
}
for (i=0; i<N_OF_PORT; ++i) {
    pthread_join(pThread[i], (void*)&status);
}

closeSockets(clientSocket);

return 0;
```

Figure 4. main함수의 실질적인 동작을 정의한 부분

먼저 5개의 port들의 번호를 seed를 통해 지정해준 후에, 각 port들에 대해 반복문을 실행하면서 configuration 및 server와 연결을 수행한다. 이후에는 각 port에 대해 pthread를 생성하고 실행시킨 후에 종료하게 된다.

아래부터는 main함수에서 사용된 함수들을 순서대로 설명할 것이다.

먼저 init5Ports 함수이다.

```
// set five port number with seed
void init5Ports(int* portNumbers, int seed) {

    for (int i=0; i<N_OF_PORT; ++i) {
        portNumbers[i] = (i+1) * seed;
    }

    return;
}
```

Figure 5. init5Ports 함수

init5Ports 함수는 port 번호를 저장할 변수와 seed 값을 parameter로 받는다. 함수 내에서 반복문을 통해, seed로 일정한 간격의 port 번호를 생성한다. 사용된 seed는 1111이다.

다음은 configureSocket 함수이다.

```
// set client socket
void configureSocket(int* clientSocket, int portNumber) {

    printf("configuring socket for port %d...", portNumber);

    *clientSocket = socket(PF_INET, SOCK_STREAM, 0);  // socket uses TCP
    if (*clientSocket == -1) {  // error
        printf("\nport %d socket error\n", portNumber);
        exit(1);
    }

    printf("done.\n");

    return;
}
```

Figure 6. configureSocket 함수

configureSocket 함수는 client socket 변수와 port 번호를 받아서 client socket 변수에 socket을 할당하게 된다. 코드에서는 socket 함수에 PF_INET 인자로 ipv4를 사용한다는 것을 알리고, SOCK_STREAM을 인자로 주어서 TCP를 통해 통신을 하게끔 했다.

다음은 socketNaming 함수이다.

```
// naming socket
void socketNaming(struct sockaddr_in* clientAddress, int clientSocket, int portNumber) {

    printf("assigning a name to the socket for port %d...", portNumber);

    memset(clientAddress, 0x00, sizeof(*clientAddress));   // clear address as zero

    clientAddress->sin_family = AF_INET;  // IPv4
    clientAddress->sin_addr.s_addr = htonl(INADDR_ANY);   // TCP
    clientAddress->sin_port = htons(portNumber); // port num

    if (bind(clientSocket, (struct sockaddr*)clientAddress, sizeof(*clientAddress)) < 0) {  // error check
        printf("\nport %d binding error\n", portNumber);
        exit(1);
    }

    printf("done.\n");

    return;
}
```

Figure 7. socketNaming 함수

socketNaming 함수는 각 socket에 주소를 할당하기 위해 사용하는 함수이다. 인자로는 빈 변수인 clientAddress, client의 socket 번호, port 번호를 받는다. memset 함수를 통해 변수를 초기화 해주고, 필요한 설정 값들(IPv4여부, TCP여부, port 번호)을 저장해준 후 각 socket에 binding을 한다.

다음은 configureServer 함수이다.

```
// configure server
void configureServer(struct sockaddr_in* serverAddress, char* serverIp, int portNumber) {

    memset(serverAddress, 0, sizeof(*serverAddress));

    serverAddress->sin_family = AF_INET; // IPv4
    serverAddress->sin_port = htons(portNumber);     // port number
    serverAddress->sin_addr.s_addr = inet_addr(serverIp);     // server addr

    return;
}
```

Figure 8. configureServer 함수

ConfigureServer 함수는 server의 주소 변수와 server의 ip, port 번호를 parameter로 받는다. 위의 socketNaming 함수와 비슷한 작업을 server에 대해 수행한다.

```
// connect socket to server
void connect2Server(struct sockaddr_in* serverAddress, int clientSocket, int portNumber) {

    printf("port %d connecting to server...", portNumber);

    if (connect(clientSocket, (struct sockaddr*)serverAddress, sizeof(*serverAddress)) < 0) {
        printf("\nport %d connecting fail\n", portNumber);
        exit(1);
    }

    printf("done.\n");

    return;
}
```

Figure 9. connect2Server 함수

Connect2Server 함수는 server와 client를 연결해주는 역할을 한다. connect 함수를 사용하였다.

```
void* receiveServerMsg(void* pThreadArgs) {

    int     msgLen;         // length of received message
    char    msg[MSG_SIZE];  // received message
    int     socketFd = *(((PThreadArgs*)pThreadArgs)->clientSocket);
    int     portNumber = ((PThreadArgs*)pThreadArgs)->portNumber;

    FILE*   fp;
    char    logPath[20];

    sprintf(logPath, "./log/%d.txt", portNumber);   // save path

    fp = fopen(logPath, "w");   // open file
    if (!fp) {  // error
        printf("file open failed\n");
        exit(1);
    }

//    while ( (msgLen = recv(socketFd, msg, 10, 0) ) != -1) {
    for (int i=0; i<1000; ++i) {
        char    logMsg[80];
        msgLen = recv(socketFd, msg, MSG_SIZE, 0);
        sprintf(logMsg, "%s  %d  %s", getCurrentTime(), msgLen, msg);
        fputs(logMsg, fp); // write message to log
        fputs("\n", fp);
    }

    fclose(fp); // close file

    return pThreadArgs;
}
```

Figure 10. receiveServerMsg 함수

receiveServerMsg 함수는 server로부터 데이터를 받는, pthread로 실행하는 함수이다. 함수의 인자는 가장 처음에 정의했던 _pthreadArgs 구조체를 통해, socket 번호와 port 번호를 넘겨주었다. 함수 내에서는 fopen 함수를 통해 각 port 번호를 이름으로 갖는 로그 파일을 쓰도록 하였다. 가장 중요한 부분은 반복문인데, 반복문을 통해서 총 1000줄의 메시지를 받아온다. recv 함수의 인자로 socket 번호, 데이터를 저장할 변수, 읽어올 데이터의 최대 길이를 넣었고, 반환 값인 수신한 데이터의 길이를 msgLen 변수에 저장하였다. getCurrentTime 함수는 현재 시간을 얻어오게 만든 함수이다. 마지막으로 fputs 함수를 통해 과제 명세의 형식에 맞게 로그 파일에 저장하였다.

아래는 getCurrentTime 함수이다.

```
char* getCurrentTime() {

    struct timeb itb;
    struct tm   *lt;
    static char s[20];

    ftime(&itb);

    lt = localtime(&itb.time);

    sprintf(s, "%02d:%02d:%02d.%03d"
            , lt->tm_hour, lt->tm_min, lt->tm_sec
            , itb.millitm);

    return s;
}
```

Figure 11. getCurrentTime 함수

getCurrentTime 함수에서, time.h header 파일에서 정의된 형식에 맞춰 인자를 넣어 현재 시간을 출력하도록 하였다.

아래는 createThread 함수이다. createThread 함수에서는 pthread 함수에서 사용하는 함수인 receiveServerMsg와 그 인자를 이용해 pthread_create 함수를 호출한다. pthread에서 사용하는 함수의 parameter로는 client socket 번호와 port 번호를 주었다.

```c
void createThread(pthread_t* pThread, int* clientSocket, int portNumber) {

    printf("creating thread... ");

    int             threadId;   // thread id
    PThreadArgs*    pThreadArgs = (PThreadArgs*)malloc(sizeof(pThreadArgs));    // pthread function arguments
    pThreadArgs->clientSocket = clientSocket;
    pThreadArgs->portNumber = portNumber;

    threadId = pthread_create(pThread, NULL, receiveServerMsg, (void *)pThreadArgs);  // create thread

    if (threadId < 0) {     // error
        perror("thread creation error");
        exit(0);
    }

    printf("done.\n");

    return;
}
```

Figure 12. createThread 함수

마지막으로 통신을 마친 후에 socket을 정리하는 함수인 closeSockets 함수이다.

```c
void closeSockets(int* clientSocket) {

    for (int i=0; i<N_OF_PORT; ++i) {
        close(clientSocket[i]);
    }

    return;
}
```

Figure 13. closeSockets 함수

closeSockets 함수는 client-socket의 번호를 받아서 각 socket들을 close하는 작업을 한다.

4. Application 실행 과정

아래 실행 과정은 server와 client가 같은 subnet에 있다고 가정하고 실행한 것이다.

1) Server-side

Blackboard에 제공된 VM 이미지 파일을 virtualbox를 통해 구동시킨 후, 로그인하여 다음 명령어를 수행하면 server-side application이 실행된다.

```
$ ./server
```

위 명령어를 실행하면 server는 5개의 client-side port를 모두 accept할 때까지 기다리게 된다.

2) Client-side

먼저, 위의 c언어로 작성한 코드를 다음 명령어를 통해 컴파일한다.

```
$ gcc client.c -o client -lpthread
```

작성한 코드 파일 이름은 client.c이다. -o 옵션은 컴파일 결과인 executable의 이름을 지정하는 것이고, -lpthread 옵션은 pthread 관련 라이브러리를 linking하기 위한 것이다.

다음으로, log 파일을 저장하기 위한 디렉토리를 생성한다.

```
$ mkdir log
```

마지막으로, 컴파일 된 프로그램을 실행한다.

```
$ ./client
```

프로그램을 실행하고 server의 ip 주소를 입력하면, 연결이 정상적으로 되었을 경우에 ./log/ 경로에 각 port 번호를 이름으로 하는 로그들이 저장된다. 각 로그는 1000줄이고, 과제의 명세 형식에 맞춰서 저장하였다.

5. 과제 수행 시의 trouble 및 troubleshooting 과정

1) pthread_create 함수

pthread_create 함수를 다룰 때, 인자로 넣어주는 함수 포인터와 함수에 넣어주는 파라미터의 타입을 정할 때 어려움을 겪었다. 그래도 함수 포인터의 경우는 void * 타입을 반환하는 함수로 해야 하는 것은 금방 알았는데, 함수의 파라미터의 타입을 정할 때에는 이해하는데 좀 오래 걸렸다. 특히 처음 구현에서는 파라미터가 한 개만 필

요해서 구조체를 사용하지 않고 하나를 넣어줬었는데, 나중에는 두 개가 필요한 상황이 와서 구조체가 필요하다는 사실을 검색을 통해 알아내서 해결하였다.

2) 로그 파일 이름 지정 문제

python이나 javascript에서는 string 간의 + 연산이 가능해서 간단하게 파일 이름을 지정할 수가 있었는데, c언어에서는 직접 더해줄 수는 없어서 다른 방법을 찾아야 했다. 익숙하지 않았던 것이 가장 큰 이유였던 것 같고, 이전 과제에서 사용했던 sprint 함수를 사용해서 버퍼에 formatting된 string을 넣어주는 것으로 해결하였다.