COSE322-00

SYSTEM PROGRAMMING

pthread를 이용한 client-side socket programming

Group	**18**
Members	**김학영 2010100086** **김인호 2012130888**
Professor	**유혁**
Free day	**X**
Submission Date	**2016.11.24**

Korea University

TABLE OF CONTENTS

1. pthread를 사용하는 이유

이번 과제에서 기본적으로 제공되는 server application은 send 이전 while문에서 볼 수 있듯, 지정된 5개의 포트가 전부 accept가 되어야 send를 시작한다. 그리고 send 과정에서도 5개의 포트 중 무작위의 포트를 통해 메시지를 보낸다.

```
printf("accepting...\n");
while (1) {
    acceptAll = 0;
    for (int i = 0; i < N_OF_PORT; i++) {
        if (acceptState[i] == 1) {
            acceptAll++; continue;
        }

        szClntAddr = sizeof(clntAddr[i]);
        hClntSock[i] = accept(hServSock[i], (struct sockaddr*)&clntAddr[i], &szClntAddr);
        if (hClntSock[i] <0)                continue;//ErrorHandling("accept() error");
        printf("accept port: %d!\n", servPorts[i]);
        acceptState[i] = 1; //connected;
    }
    if (acceptAll == N_OF_PORT) break;
}

//send
while (1) {
    i = rand() % 5;
    //send ■ ■
    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);
    }
}
```

따라서 client application이 server application으로부터 정상적으로 패킷을 수신하기 위해서는 5개의 포트에 동시에 connection을 맺은 후에도 지속적으로 read를 해야 한다. 매시간 어떤 포트를 통해 data가 전송될 지 예측하기 어렵기 때문이다. 게다가 blocking I/O 방식을 사용하는 경우 하나의 client application에서 Thread의 도움 없이 여러 소켓을 생성하여 read를 한다면, 최악의 경우 기아 현상이 발생할 수 있다. 예를 들어, client application에서 반복문 안에서 소켓 5개가 순차적으로 read 함수를 실행한다고 하면 첫 번째 소켓이 read 함수를 실행하는 순간 block이 되어서 프로세스가 잠시 중단되는데, 만약 server application에서 첫 번째 소켓에 연결된 포트 쪽으로 data를 보내지 않는다면, client application은 아무런 data도 받지 못하게 되는 것이다.

물론 이러한 문제점은 non-blocking I/O 방식을 사용하거나(read with O_NONBLOCK 또는 aio_read) select 시스템콜을 사용하여 해결할 수 있다. 또는 client application 자체를 여러 개 수행하는 방법도 있다. 그런데, non-blocking 방식의 경우 과제의 요구사항 중 하나인 패킷의 도착시간을 기록할 때 sk_buff 구조체를 참조하여야 하므로 예외처리 등을 고려했을 때 코드

가 복잡해지며, 커널 모듈을 사용해야 하기 때문에 이번 과제의 의도에 부합하지 않는다. 또한, client application이 read를 수행하는 시간에 제한을 두었다고 가정할 경우 서버에서 보낸 모든 패킷을 read하지 못할 가능성도 있다. 또한, polling을 하는 과정이 패킷이 도착하는 속도보다 현저히 느릴 경우에는, sk_buff 구조체가 지속적으로 쌓이면서 커널 영역의 메모리가 낭비되는 문제점이 발생할 가능성이 높아진다. 그리고 select 시스템콜의 경우에는 기존의 read 시스템콜 뿐만 아니라 select라는 시스템콜이 추가적으로 사용되기 때문에 context switching 비용이 증가하게 되는 문제점이 있다. 이와 같은 맥락으로, 여러 개의 blocking I/O 방식의 client app을 실행하는 경우에도 프로세스 5개에 대한 context switching 비용은 프로세스 1개보다 높아져 overhead가 커진다.

그런데, Thread를 사용한다면 Thread가 프로세스에 비해 메모리 활용면에서 효율적이기 때문에 서버 측의 5개의 포트에 연결을 하기 위하여 굳이 여러 개의 application을 실행하지 않아도 된다. 프로세스 5개를 실행할 때에 비해 context switching의 비용이 줄어들 뿐만 아니라, 하나의 client application이 여러 개의 Thread를 생성하도록 하여 각각의 Thread가 소켓을 하나씩 생성하고 서버 측의 포트에 각각 연결되도록 하면, 과제에서 주어진 server application의 조건을 쉽게 만족시킬 수 있다. 따라서 pthread를 이용해서 총 5개의 쓰레드를 생성하여 각 쓰레드가 각각 다른 포트로 connection을 맺은 후 일정 시간동안 read 하도록 client application을 작성하였다.

2. 소스 코드 설명

client.c 파일은 main 함수와 connect_server 함수로 구성되어 있다. main 함수에서는 5개의 포트 번호(5555, 6666, 7777, 8888, 9999)를 지정한 뒤, 각각의 포트 번호를 인자로 받는 connect_server 함수를 실행하는 5개의 Thread를 생성한다. 함수 실행이 완료되면 pthread_join를 실행하여 Thread가 정상적으로 종료될 수 있도록 한다. 또한 thread가 서버로부터 받은 패킷에 대한 정보를 저장할 output file들을 저장하기 위한 directory를 생성한다.

```c
int main() {
  pthread_t p_thread[5];
  int thr_id, status, port_number[5] = {5555,6666,7777,8888,9999}, i;

  //make a directory to store output files
  mkdir(OUTPUT_DIR, 0755);

  //create 5 threads which call connect_server function with respective port number
  for(i=0;i<5;i++)
    thr_id = pthread_create(&p_thread[i], NULL, connect_server, (void *)&port_number[i]);


  //quit 5 threads
  for(i=0;i<5;i++) {
    pthread_join(p_thread[i], (void **) &status);
    if(status != 0)
      printf("An error while joining a thread\n");
  }

  return 0;
}
```

directory의 이름과 서버의 IP 주소는 #define으로 정의된 매크로 상수를 수정하여 변경할 수 있다.

```c
#define SERVER_ADDR "192.168.56.103"
#define OUTPUT_DIR "output"
```

connect_server 함수는 main 함수에서 생성한 5개의 Thread가 실행하게 될 함수이다. 인자로 포트 번호를 받는다. 함수가 실행이 되면 우선 socket 함수를 호출하여 TCP/IP 방식의 소켓을 생성한다.

```c
//create a socket
client_socket=socket(PF_INET, SOCK_STREAM, 0);

if(client_socket==-1) {
  printf("Port #%d : Socket creation failed.\n", port);
  exit(1);
}
```

그 후에 sockaddr_in 구조체 변수인 server_addr를 통해 서버의 IP 주소 정보, 포트 정보 등을 설정하고 connect 함수를 호출하여 서버에 연결을 시도한다.

```c
//connect to the server
memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family=AF_INET;
server_addr.sin_port=htons(port);
server_addr.sin_addr.s_addr=inet_addr(SERVER_ADDR);

if(connect(client_socket, (struct sockaddr*)&server_addr, sizeof(server_addr))<0) {
  printf("Port #%d : Connection failed.\n", port);
  exit(1);
}
```

연결이 완료되었다면 서버로부터 받은 패킷의 정보들을 저장하기 위한 파일을 연다. 이때 파일명은 "포트번호.txt"이며, main 함수에서 미리 생성해둔 directory에 포함된다. 만약 이미 같은 이름을 가진 파일이 존재한다면 덮어쓰고, 그렇지 않다면 새로 생성한다.

```
//open a file to write data
sprintf(outputfiledir, "/%d.txt", port);
strcat(outputfilepath, outputfiledir);
outputfile=fopen(outputfilepath, "w");
```

그 다음 gettimeofday 함수를 통해 현재 시간과 메시지의 도착 시간을 현재 시간으로 초기화해 두고, 서버로부터 메시지를 받을 때마다 도착 시간을 갱신한다. 그리고 해당 메시지가 도착한 시간과 메시지의 길이, 그리고 내용을 output file에 write한다. 이 과정을 3초 동안 반복한다.

```
//record the current time & initialize packet arrival time
gettimeofday(&start_time, NULL);
gettimeofday(&received_time, NULL);

//write contents and arrival times of packets to the file for 3 seconds
while(received_time.tv_sec-start_time.tv_sec < 3) {
  read(client_socket, buffer, BUF_LEN);
  gettimeofday(&received_time, NULL);
  tm=localtime(&received_time.tv_sec);
  fprintf(outputfile, "%02d:%02d:%02d.%03ld %ld %s\n",
      tm->tm_hour, tm->tm_min, tm->tm_sec, received_time.tv_usec/1000, strlen(buffer), buffer);
}
```

3초가 지나면, output file을 close하고 함수를 종료한다.

3. Troubleshooting

지정된 시간 형식에 맞추는 것에 어려움을 겪었다. 시간을 측정할 때 사용된 timeval 구조체는 멤버로 초 단위를 구하는 tv_sec과 마이크로 초 단위를 구하는 tv_usec밖에 없기 때문이다. tm이라는 구조체를 발견하여 localtime이라는 함수를 통해 시간, 분 단위까지 구할 수 있었다.

```
gettimeofday(&received_time, NULL);
tm=localtime(&received_time.tv_sec);
```

```
tm->tm_hour, tm->tm_min, tm->tm_sec
```

그 뒤 밀리 초 단위를 출력하는 것은 단순히 tv_usec에 1000을 나누어서 해결하였지만, 문제가 발생하였다. "s.ms" 형식으로 출력을 해야 하는데, 만약 12345 마이크로 초를 1000으로 나누면 12 밀리 초가 결과로 나오기 때문에 현재 시간이 2초라고 하면 "2.012"로 표기를 해야 하는데, "2.12"가 되는 현상이 나타난 것이다. 이는 fprintf 함수에서 마이크로 초 부분의 출력 형식을 %03ld로 설정하여 해결하였다.

```
"%02d:%02d:%02d.%03ld %ld %s\n",
tm->tm_hour, tm->tm_min, tm->tm_sec, received_time.tv_usec/1000
```

서버로부터 받은 메시지의 길이를 출력할 때도 문제가 있었다. 원래는 sizeof 함수를 통해 buffer의 크기를 출력하도록 하였으나, 이는 실제 메시지의 길이가 아닌 buffer의 크기를 반환하기 때문에 항상 같은 값이 나오는 문제가 발생했다. 따라서 실제 메시지의 길이를 출력하기 위해 strlen 함수로 대체하였다.

그리고 초기에 buffer의 크기를 128 Byte로 설정하고 Thread가 실행되는 기간을 60초로 설정하였었는데, 한 번에 받아오는 data의 크기가 너무 적어 output file이 너무 길어진다고 판단하여 buffer size를 1024 Byte로 늘렸다. 그런데도 여전히 output file이 심각하게 길어지는 현상을 발견하여 Thread가 실행되는 시간을 3초로 단축하였다. 그랬더니 대략적으로 한 파일 당 10000줄 정도가 나오게 되었다.

```
//write contents and arrival times of packets to the file for 3 seconds
while(received_time.tv_sec-start_time.tv_sec < 3) {
```