WINDOW NETWORK -CHAPTER 4-

SOULSEEK

목차

1. 소켓 입출력 모델 – 1

1. 소켓 입출력 모델 – 1

소켓 입출력 모델(socket I/O model)

- 다수의 소켓을 관리하고 소켓에 대한 입출력을 처리하는 일관된 방식.
- 프로그래밍 복잡도는 높아지지만 시스템 자원을 적게 사용하면서도 다수의 클라이언트를 효율적으로 처리하는 서버를 만들 수 있다.

소켓 모드의 종류

- 소켓 함수 호출 시 동작하는 방식에 따라 블로킹과 넌블로킹 소켓으로 구분하고 이를 소켓모드라 부른다.

블로킹 소켓(blocking socket)

- 소켓 함수 호출 시 조건이 만족되지 않으면 함수가 리턴하지 않고 **Thread**실행이 정지한다.
- 조건이 만족되면 소켓 함수가 리턴하면서 정지된 **Thread**가 깨어나 실행을 재개한다.

넌블로킹 소켓(nonblocking socket)

- 소켓 함수 호출 시 조건이 만족되지 않더라도 함수가 리턴하므로 **Thread**가 중단 없이 다음 코드를 수행한다.
- 접속한 클라이언트가 없어도 **accept()**함수가 리턴하고, 소켓 수신 버퍼에 도착한 데이터가 없어도 **recv()**함수가 리턴한다.
- 조건이 만족되지 않았을 시 오류를 리턴하고 **WSAEWOULDBLOCK**라는 코드가 나온다. 이 코드가 감지되면 그 시점의
- 작업을 재시작 하면 된다.

1. 소켓 입출력 모델 – 1

소켓 함수의 리턴 조건

소켓 함수	리턴 조건
accept()	접속한 클라이언트가 있을 때
connect()	서버에 접속을 성공했을 때
send()	응용 프로그램이 전송을 요청한 데이터를 소켓 송신 버퍼에 모두 복사했을 때
recv()	소켓 수신 버퍼에 도착한 데이터가 **1**바이트 이상 있고 이를 응용 프로그램이 제공한 버퍼에 복사 했을 때

nonblocking 소켓 생성 코드

socket() 함수는 기본적으로 블로킹 소켓을 생성한다**, iocltsocket()** 함수를 호출해서 **nonblocking**을 만든다**.**

```
u_long on = 1;
reval = ioctlsocket(sock, FIONBIO, &on); //socket을 넌블로킹으로 전환

if(retval = SOCKET_ERROR)
    err_quit("ioctlsocket()");
```

Nonblocking project를 확인해보자**.** 클라이언트는 **TCP** 클라이언트를 사용해도 된다**.**

1. 소켓 입출력 모델 – 1

넌블로킹 소켓의 특징

장점

- 소켓 함수 호출 시 항상 리턴하므로 조건이 만족되지 않아 **Thread**가 오랜 시간 정지하는 상황**,** 즉 교착 상태**(deadlock)**가 생기지 않는다**.**
- **Multy Thread**를 사용하지 않고도 여러 소켓에 대해 돌아가면서 입출력을 처리할 수 있다**.** 필요하다면 중간에 소켓과 직접 관계가 없는 다른 작업을 할 수 도 있다**.**

단점

- 소켓 함수를 호출할 때마다 **WSAEWOULDBLOCK**과 같은 오류 코드를 확인하고 처리해야 하므로 프로그램구조가 복잡해 진다**.**
- 블로킹 소켓을 사용한 경우보다 **CPU** 사용률이 높다**.**

서버 작성 모델의 종류

반복 서버

- 여러 클라이언트를 한 번에 하나씩 처리**,** 기본적인 형태의 **TCP, UDP** 서버가 여기에 속한다**.**
- 장점 **: Thread** 한 개 만으로 구현하므로 시스템 자원 소모가 적다**.**
- 단점 **:** 한 클라이언트의 처리 시간이 길어지면 다른 클라이언트의 대기 시간이 길어진다**.**

병렬 서버

- 여러 클라이언트를 동시에 처리한다**. MultyThreadTCP** 서버가 여기에 속한다**.**
- 장점 **:** 한 클라이언트의 처리 시간이 길어지더라도 다른 클라이언트에 영향을 주지 않는다**.**
- 단점 **: Thread**를 여러 개 생성하여 구현하므로 시스템 자원 소모가 없다**.**

1. 소켓 입출력 모델 – 1

소켓 입출력 모델의 특징
- 반복서버와 병행 서버의 장점을 모두 갖추면서 각자의 단점을 해결한 형태로 만들어 갈 수 있다.

이상적인 소켓 입출력 모델
- 가능한 많은 클라이언트가 접속 할 수 있다.
- 서버는 각 클라이언트의 서비스 요청에 빠르게 반응하며, 고속으로 데이터를 전송한다.
- 시스템 자원 사용량을 최소화 한다.

소켓 입출력 모델의 종류
Windows OS에서 지원하는 소켓 입출력 하는 모델 중에 **WSAAsyncSelect, Overlapped, Completion Port**를 알아볼 것이다.

2. WSAASYNCSELECT

2. WSAASYNCSELECT

Select 형식의 원리

- 소켓 셋이라는 것이 필요하다, 소켓 디스크립터의 집합을 말하며, 호출할 함수의 종류에 따라 소켓을 적당한 셋에 넣어두어야 한다.
- 각 시점마다 소켓 셋을 정해두고 필요한 동작을 수행하게 하면된다.
- 크게 읽기 셋, 쓰기 셋, 예외 셋을 쓴다.

읽기 셋

함수 호출 시점

- 접속한 클라이언트가 있으므로 **accept()**함수를 호출할 수 있다.
- 소켓 수신 버퍼에 도착한 데이터가 있으므로 **recv(), recvfrom()**등의 함수를 호출해 데이터를 읽을 수 있다
- **TCP** 연결이 종료되었으므로 **recv(), recvform()** 등의 함수를 호출해 연결 종료를 감지할 수 있다.

쓰기 셋

함수 호출 시점

- 소켓 송신 버퍼의 여유 공간이 충분하므로 **send(), sendto()** 등의 함수를 호출하여 데이터를 보낼 수 있다.

함수 호출 결과

- 넌블로킹 소켓을 사용한 **connect()**함수 호출이 성공했다.

예외셋

함수 호출 시점

- 데이터가 도착했으므로 **recv(), recvform()**등의 함수를 호출하여 데이터를 받을 수 있다.

함수 호출 결과

- 넌블로킹 소켓을 사용한 **connect()**함수 호출이 실패했다.

2. WSAASYNCSELECT

WSAAsyncSelect 모델의 동작 원리

- 소켓 함수 호출이 성공할 수 있는 시점을 윈도우 메시지 수신으로 알 수 있다.
- 소켓 함수 호출 시 조건이 만족되지 않아 생기는 문제를 해결 할 수 있다.

소켓 입출력 절차

1. **WSAAsyncSelect()** 함수를 호출하여, 소켓 이벤트를 알려줄 윈도우 메시지와 관심있는 네트워크 이벤트를 등록한다. ➔ 소켓으로 데이터를 받을 수 있는 상황이 되면 정의된 윈도우 메시지로 알려달라고 등록한다. **#define WM_SOCKET (WM_USER+1);**
2. 등록한 네트워크 이벤트가 발생하면 윈도우 메시지가 발생하며 윈도우 프로시저가 호출된다.
3. 윈도우 프로시저에서는 받은 메시지의 종류에 따라 적절한 소켓 함수를 호출하여 처리한다.

int WSAAsyncSelect(SOCKET s, HWND hWnd, unsigned int wMsg, long lEvent);

- 성공 **: 0,** 실패 **: SOCKET_ERROR**
- 첫 번째 인자 **:** 네트워크 이벤트를 처리하고자 하는 소켓
- 두 번째 인자 **:** 네트워크 이벤트가 발생하면 메시지를 받을 윈도우의 핸들이다.
- 세 번째 인자 **:** 네트워크 이벤트가 발생하면 윈도우가 받을 메시지다. 소켓을 위한 윈도우 메시지는 따로 정의되어 있지 않으므로 **WM_USER+x** 형태의 사용자 정의 메시지를 이용하면 된다.
- 네 번째 인자 **:** 관심있는 네트워크 이벤트를 비트 마스크조합으로 나타낸다.

WSAAsyncSelect Project를 참조하자.

2. WSAASYNCSELECT

네트워크 이벤트

- **FD_ACCEPT :** 접속한 클라이언트가 있다.
- **FD_READ :** 데이터 수신이 가능하다.
- **FD_CLOSE :** 상대가 접속을 종료했다.
- **FD_CONNECT :** 통신을 위한 연결 절차가 끝났다.
- **FD_OOB : OOB** 데이터가 도착했다.

WSAAsyncSelect()함수 사용시 주의 사항

- 함수를 호출하는데 넘겨준 소켓은 자동으로 넌블로킹 모드로 전환된다.
- **Accept()**함수가 리턴하는 소켓은 연결 대기 소켓과 동일한 속성을 갖게 되기 때문에 직접적으로 송수신 처리를 하지 않는다, **FD_READ, FD_WRITE** 이벤트를 처리하지 않기 때문에 **FD_READ, FD_WRITE**를 이벤트로 등록해야한다.
- **WSAEWOULDBLOCK**오류 코드가 발생하는 경우가 드물게 있으므로 이를 처리하는 부분이 필요하다.
- 윈도우 메시지를 받았을 때 적절한 소켓 함수를 호출하지 않으면, 다음 번에는 같은 윈도우 메시지가 발생하지 않는다, **FD_READ**에 대응하는 **recv()**함수를 호출하지 않으면 동일 소켓에 대한 **FD_READ**이벤트는 다시 발생하지 않는다. 따라서 윈도우 메시지가 발생하면 대응 함수를 호출해야 하며, 그렇지 않을 경우 응용프로그램이 나중에 직접 메시지를 발생 시켜야 한다.

대응 함수

FD_ACCEPT : accept()
FD_READ : recv(), recvfrom()
FD_WRITE : send(), sendto()
FD_CLOSE : 없음
FD_CONNECT : 없음
FD_OOB : recv(), recvfrom()