WINDOW NETWORK -CHAPTER 3-

SOULSEEK

목차

1. 기존 **TCP**의 문제점과 해결방안

1. 기존 TCP의 문제점과 해결방안

- 기존의 통신방식의 문제점을 이해하고 그에 대한 해결책의 방법으로 접근하자!

기존 TCP 서버 – 클라이언트의 문제점과 해결 방안

1. 클라이언트가 두 개 이상이 서버에 접속할 수 있으나, 서버가 동시에 클라이언트 구대 이상의 서비스를 할 수 없다 .
   - ➔ 서버가 각 클라이언트와 연결해 통신하는 시간을 줄이고, 매번 통신할 때마다 서버에 접속과 해제를 반복 한다.
   - ➔ 서버에 접속한 각 클라이언트를 **Thread**를 이용해 독립적으로 처리한다.
   - ➔ 소켓 입출력 모델을 사용한다.

2. 서버와 클라이언트의 **send(), recv()** 함수의 호출 순서가 맞지 않아 서로 대기하는 상황을 피해야한다.
   - ➔ 데이터 송수신 부분을 잘 설계해 교착 상태가 발생하지 않게 한다.
   - ➔ 소켓에 타임 아웃 옵션을 적용해, 소켓 함수 호출 시 작업이 완료되지 않아도 일정 시간 후에 리턴하게 한다.
   - ➔ 넌블로킹 소켓을 사용한다.
   - ➔ 소켓 입출력 모델을 사용한다.

1. 기존 TCP의 문제점과 해결방안

1번 문제의 해결방안의 장단점.

1. 통신 시간을 짧게 하여 지속적으로 접속해제를 이행하는 방법.
   - 장점 : 특별한 기법을 도입하지 않고도 쉽게 구현할 수 있다. 서버의 시스템 자원을 적게 사용한다
   - 단점 : 파일 전송 프로그램과 같이 대용량 데이터를 전송하는 응용 프로그램을 구현하는 데는 적합하지 않다. 또한 클라이언트 수가 많을 경우 처리 지연 시간이 길어질 확률이 높다.

2. **Thred**를 이용하는 방법
   - 장점 : 소켓 입출력 모델에 비해 비교적 쉽게 구현할 수 있다.
   - 단점 : 접속한 클라이언트 수에 비례해 **Thread**를 생성하므로 서버의 시스템 자원을 많이 사용한다.

3. 소켓 입출력 모델을 사용하는 방법
   - 장점 : 소수의 **Thread**를 이용해 다수의 클라이언트를 처리할 수 있다. 따라서 **Thread**만 이용하는 방법을 사용하는 것보다 자원을 적게 사용한다.
   - 단점 : 구현이 가장 어렵다.

1. 기존 TCP의 문제점과 해결방안

2번 문제의 해결방안의 장단점.

1. 데이터 송수신 부분을 잘 설계해 교착 상태가 발생하지 않게 한다.
   - 장점 : 특별한 기법을 도입하지 않고도 구현할 수 있다.
   - 단점 : 데이터 송수신 패턴에 따라 교착 상태가 발생할 수 있다. 따라서 이 방법을 모든 경우에 적용할 수는 없다.

2. 소켓에 타임아웃 옵션을 적용한다.
   - 장점 : 비교적 간단하게 구현할 수 있다.
   - 단점 : 다른 방법보다 성능이 떨어진다.

3. 넌블로킹 소켓을 사용한다.
   - 장점 : 교착 상태를 막을 수 있다.
   - 단점 : 구현이 복잡하다. 시스템 자원을 불필요하게 낭비할 가능성이 크다.

4. 소켓 입출력 모델을 사용한다.
   - 장점 : 넌블로킹 소켓의 단점을 보완하고 더불어 교착 상태를 막을 수 있다.
   - 단점 : 구현이 가장 어렵지만 일관성 있게 구현할 수 있다.

1. 기존 TCP의 문제점과 해결방안

2번 문제의 해결방안의 장단점.

1. 데이터 송수신 부분을 잘 설계해 교착 상태가 발생하지 않게 한다.
   - 장점 : 특별한 기법을 도입하지 않고도 구현할 수 있다.
   - 단점 : 데이터 송수신 패턴에 따라 교착 상태가 발생할 수 있다. 따라서 이 방법을 모든 경우에 적용할 수는 없다.

2. 소켓에 타임아웃 옵션을 적용한다.
   - 장점 : 비교적 간단하게 구현할 수 있다.
   - 단점 : 다른 방법보다 성능이 떨어진다.

3. 넌블로킹 소켓을 사용한다.
   - 장점 : 교착 상태를 막을 수 있다.
   - 단점 : 구현이 복잡하다. 시스템 자원을 불필요하게 낭비할 가능성이 크다.

4. 소켓 입출력 모델을 사용한다.
   - 장점 : 넌블로킹 소켓의 단점을 보완하고 더불어 교착 상태를 막을 수 있다.
   - 단점 : 구현이 가장 어렵지만 일관성 있게 구현할 수 있다.

2. THREAD

2. THREAD

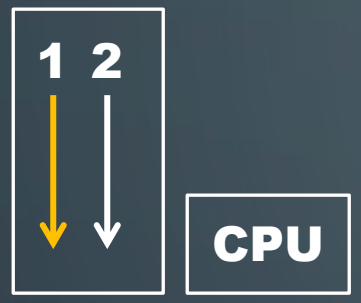
Process와 Thread

- **Process**
  - 코드, 데이터, 리소스를 파일에서 읽어 들여 운영체제가 할당해놓은 메모리 영역에 담고 있는 일종의 컨테이너이며 정적이다.
  - **Process**에 **Program**이 올라가기 전까지 활성화 되지 않는다.

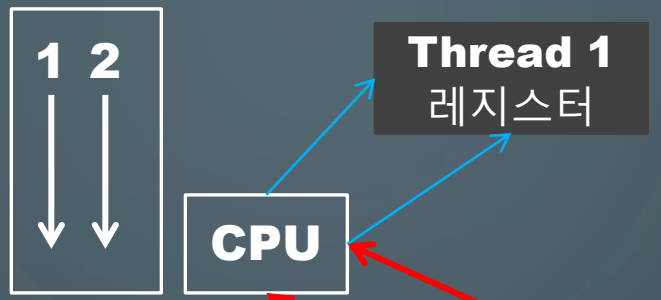
- **Thread**
  - **CPU**시간을 할당 받아 **Process** 메모리 영역에 있는 코드를 수행하고 데이터를 사용하며 동적이다.
  - **Program**이 **Process**에 올라가 실행이 되기 위해서는 하나이상의 **Thread**가 필요하다.
  - 최소에 생성되는 **Thread**를 주 **Thread** 또는 **Main Thread**라고 한다.
  - 대부분의 **OS** 응용 프로그램들은 **MultyThread**로 운용 된다.
  - 두 개의 **Thread**가 동시에 운용될 순 없지만 빠른 타이밍으로 교차 실행을 하면서 전환하면 동시에 실행되는 것처럼 느끼게 된다. ➔ 컨텍트 전환이라는 것을 통해 **Thread**는 자신이 사용하고 있는 재원을 유지한 채 전환을 할 수 있다.

2. THREAD

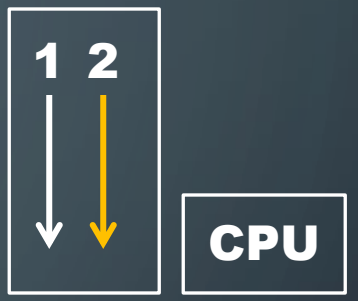
Thread 전환 과정



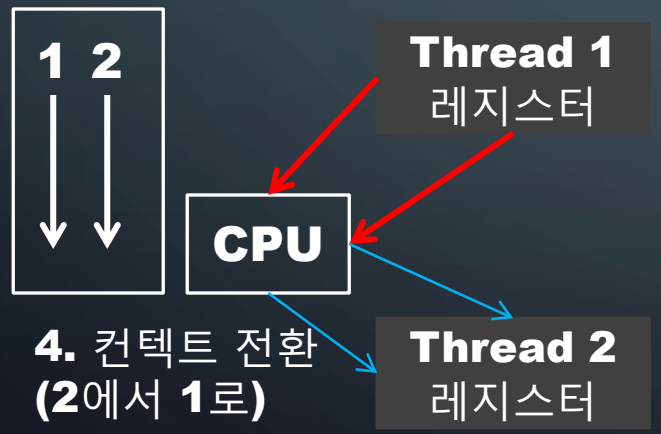
1. Thread 1 실행



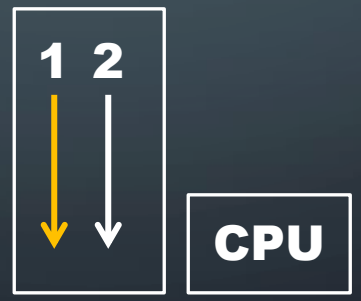
2. 컨텍트 전환
(1에서 2로)



3. Thread 1 실행



4. 컨텍트 전환
(2에서 1로)



5. Thread 1 실행

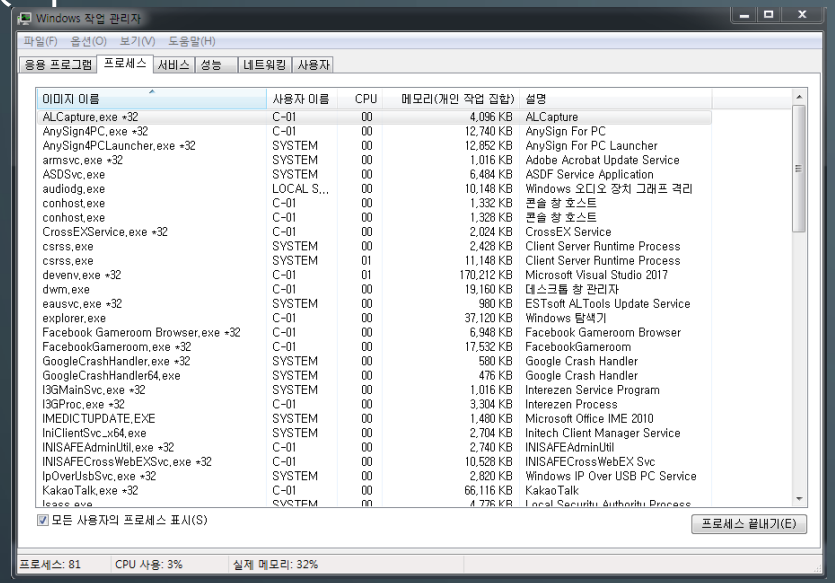
2. THREAD

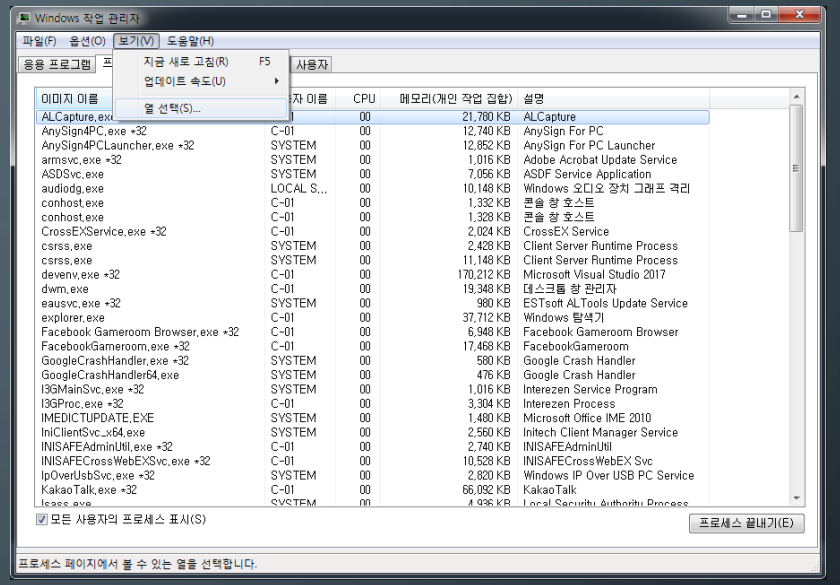
Thread 전환 과정

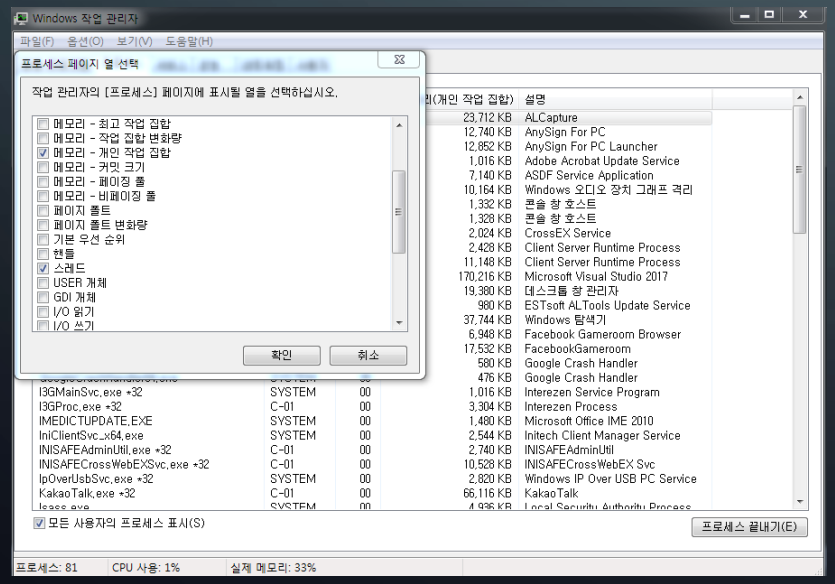
1. **Thread 1**번 실행 중**,** 명령을 하나씩 수행할 때마다 **CPU** 레지스터 값과 메모리의 스택 내용이 변경된다**.**
2. **Thread 1**번의 실행을 중지하고 실행 상태를 저장한다**.(**스택은 메모리에 계속 유지되므로 그림에서는 **CPU** 레지스터만 저장하는 것으로 표시했다**).** 이전에 저장해둔 **Thread 2**의 상태를 복원한다**.**
3. **Thread 2**번 실행 중**,** 명령을 하나씩 수행할 때마다 **CPU** 레지스터 값과 메모리의 스택 내용이 변경된다**.**
4. **Thread 2**번의 실행을 중지하고 실행 상태를 저장한다**.** 이전에 저장해둔 **Thread 1**의 상태를 복원한다**.**
5. **Thread 1**을 다시 실행한다**.** 이전 실행 상태를 복원했으므로 **Thread 1**은 마지막으로 수행한 명령 다음 위치부터 진행한다**.**

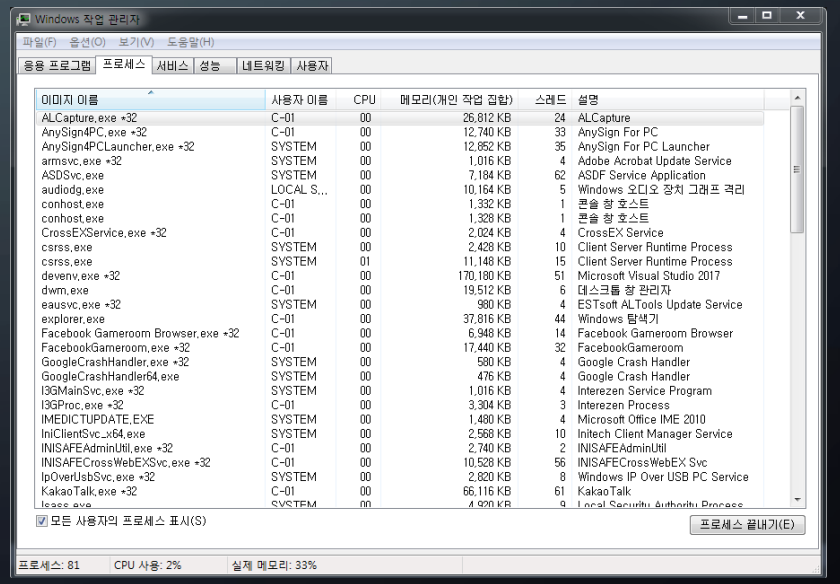
2. THREAD

- **Window** 작업 관리자를 통해 어떠한 응용 프로그램들이 **Thread**를 어떻게 쓰고 있는지 볼 수 있다.









2. THREAD

Thread 생성과 종료

- **Main()**함수가 주 **Thread** 함수에 해당하고 별개로 **Thread**에서 돌아갈 **Thread**함수가 따로 필요하다**.**

Thread 생성함수

- **CreateThread() – C/C++**라이브러리를 쓴다면 **_beginThreadex()**를 사용하면 된다**.**

HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes, SIZE_T dwStackSize, LPTHREAD_START_ROUTINE lpStartAddress, LPVOID lpParameter, DWORD dwCreationFlags, LPDWORD lpParameter);

- **성공시 Thread 핸들, 실패시 NULL**
- **첫 번째 인자 : SEBURITY_ATTRIBUTES** 구조**CP**를 통해 핸들 상속과 보안 디스크립터 정보를 전달**. NULL**을 사용
- **두 번째 인자 : Thread**에 할당되는 스택 크기다**. 0**을 사용하면 기본크기를 사용한다**. C++**에서 **1MB**
- **세 번째 인자 : Thread** 함수의 시작 주소다**. Thread** 함수는 반드시 **DWORD WINAPI ThreadProc(LPVOID lpParameter){ }** 형식이 되어야 한다**.**
- **네 번째 인자 : Thread** 함수에 전달할 인자**, void**형 포인터므로 포인터 크기보다 작거나 같은 데이터는 값 또는 주소 형태로 전달된다**.** 포인터 보다 큰 데이터는 값을 구조체나 배열에 넣고 주소 형태로 전달한다**.**
- **다섯 번째 인자 : Thread** 생성을 제어하는 값으로 **0**또는 **CREATE_SUSPENDED**를 사용**, 0**은 곧바로 실행**, CREATE_SUSPENDED**는 **ResumeThread()**함수 호출 전까지 대기
- **여섯 번째 인자 : DWORD**형 변수를 전단하면 여기에 **Thread ID**가 저장된다**,** 필요 없다면 **NULL**값을 사용해도 된다**.**

2. THREAD

Thread 종료 방법

- **Thread** 함수를 리턴한다**.**
- **Thread** 함수 안에서 **ExitThread()** 함수를 호출한다**.**
- 다른 **Thread**가 **TerminateThread()**함수를 호출해 **Thread**를 강제 종료 시킨다**.**
- **Main Thread**가 종료하면 모든 **Thread**가 종료된다**.**

Thread 종료 함수

- **void ExitThread(DWORD dwExitCode);**
  - **C/C++**라이브러리하면 **_endthreadex()**를 사용하면 된다**.**
  - 첫 번째 인자 **:** 종료코드

- **BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode);**
  - 첫 번째 인자 **:** 종료할 **Thread**를 가리키는 핸들
  - 두 번째 인자 **:** 종료 코드

2. THREAD

Thread 구현 형태 – **Thread Project**를 확인하자.

```
DWORD WINAPI f(LPVOID arg)
{
    ...
    return 0; // Thread 종료
}

int Main()
{
    ...
    // 첫 번째 Thread 생성
    HANDLE hThread1 = CreateThread(NULL, 0, f, NULL, 0, NULL);
    if(hThread1 == NULL) 오류처리;

    ...
    //두 번째 Thread 생성
    HANDLE hThread2 = CreateThread(NULL, 0, f, NULL, 0, NULL);
    if(hThread == NULL) 오류처리;
}
```

2. THREAD

Thread 제어

Thread Priority 변경

- **Thread**들은 **CPU**시간을 사용하려고 서로 경쟁한다**.** 따라서 각 **Thread**에 **CPU** 시간을 적절히 분배하기 위한 정책을 사용하는데**,** 이를 **Thread scheduling** 또는 **CPU scheduling**이라고 한다**.**
- **windows OS**에서는 **Priority**를 기준으로 하는 기법으로 **scheduling**한다**.**
- **Priority**를 결정하는 요소
  - **Process Priority :** 우선순위 클래스라 부른다**.**
  - **Thread Prioriry :** 우선순위 레벨이라 부른다**.**
  - 보통 응용 프로그램에서는 우선순위레벨을 변경하고 **SetThreadPriority()**와 **GetThreadPriority()**가 이를 지원한다**.**

BOOL SetThreadPriority(HANDLE hThread, int nPriority);
우선순위 레벨을 변경한다**,** 성공 **: 0**이 아닌 값**,** 실패 **: 0**
첫 번째 인자 **: Thread** 핸들 값
두 번째 인자 **:** 우선 순위 레벨

Int GetThreadPriority(HANDLE hThread);
우선순위 레벨을 알려준다**,** 성공 **:** 우선순위 레벨**,** 실패 **: THREAD_PRIORITY_ERROR_RETURN**
첫 번째 인자 **: Thread** 핸들 값

2. THREAD

Thread종료 기다리기, 실행 중지, 재 시작

Thread 종료 기다리기

- 다른 **Thread**의 종료 여부를 체크 할 때 사용한다.

종료 기다리기 함수

DWORD WaitSingleObject(HANDLE hHandle, DWORD dwMilliseconds);

- 성공 **: WAIT_OBJECT_0 or WAIT_TIMEOUT,** 실패 **: WAIT_FAILED**
- 첫 번째 인자 **:** 종료를 기다릴 대상 **Thread**를 나타낸다.
- 두 번째 인자 **:** 대기 시간으로**,** 밀리초 단위를 사용한다**.** 이 시간안에 **Thread**가 종료하지 않으면 **WaitForSingleObject()** 함수는 리턴하고**,** 이때 리턴 값은 **WAIT_TIMEOUT**이 된다**. Thread**가 종료한 경우에는 **WAIT_OBJECT_0**을 리턴**,** 대기 시간으로 **INFINITE** 값을 사용하면 **Thread**가 종료할 때까지 무한히 기다린다**.**

Ex)

```
HANDLE hThread = CreateThread(...);
DWORD retval = WaitForSingleObject(hThread, 1000);
if(retval == WAIT_OBJECT_0){ ... } // Thread 종료
else if(retval == WAIT_TIMEOUT) { ... } // 타임아웃(Thread는 아직 종료 안 함)
else { ... } //Error 발생
```

2. THREAD

DWORD WaitForMultipleObjects(DWORD nCount, const HANDLE *lpHandles, BOOL bWaitAll, DWORD dwMilliseconds);

- **WaitForSingleObject()**함수를 사용하면 **Thread** 개수 만큼 호출해야 하는데 한번만 호출해서 이를 해결 할 수 있다.
- 성공 **: WAIT_OBJECT_0 ~ WAIT_OBJECT_0 + nCount -1 or WAIT_TIMEOUT** 실패 **: WAIT_FAILED**
- 첫 번째 인자 **:** 종료 대기할 **Thread** 개수**, MAXMUM_WAIT_OBJECT**를 최대 값으로 설정
- 두 번째 인자 **:** 종료 대기할 **Thread**들의 배열
- 세 번째 인자 **: TRUE**면 모든 **Thread**가 종료할 때까지 기다린다**. FALSE**면 하나의 **Thread**가 종료하는 즉시 리턴한다**.**
- 네 번째 인자 **:** 대기 시간으로**,** 밀리초 단위를 사용한다**.** 이 시간안에 **Thread**가 종료하지 않으면 **WaitForSingleObject()** 함수는 리턴하고**,** 이때 리턴 값은 **WAIT_TIMEOUT**이 된다**. Thread**가 종료한 경우에는 **WAIT_OBJECT_0**을 리턴**,** 대기 시간으로 **INFINITE** 값을 사용하면 **Thread**가 종료할 때까지 무한히 기다린다**.**

ex)

```
//모든Thread 종료를 기다린다.
HANDLE hThread[2];
hThread[0] = CreateThread(...);
hThread[1] = CreateThread(...);
WaitForMultipleObjects(2, hThread, TRUE, INFINITE);
```

2. THREAD

Thread 실행 중지와 재 시작

void Sleep(DWORD dwMilliseconds);

- 일정 시간 대기 후 자동 시작
- 첫 번째 인자 **:** 대기 시간

SuspendThread(HANDLE hThread);

- 일시 정지 함수
- 성공 **:** 중지 횟수**,** 실패 **: -1**
- 첫 번째 인자 **:** 일시 정지 **Thread** 핸들**.**

DWORD ResumeThread(HANDLE hThread);

재 시작 함수
성공 **:** 중지 횟수**,** 실패 **: -1**
첫 번째 인자 **:** 재 시작 **Thread** 핸들**.**

ThreadPriority_WaitEnd 프로젝트를 참고 하자**.**

3. MULTY THREAD 구현

3. MULTY THREAD 구현

```
int main()
{
    ...
    while(1)
    {
        //1. 클라이언트 접속 수용
        client_sock = accept(listen_sock, ...);
        ...
        //2. Thread 생성
        CreateThread(NULL, 0, ProcessClient, (LPVOID)Client_sock, 0, NULL);
    }
    ...
}

DWORD WINAPI ProcessClient(LPVOID arg)
{
    //3. 전달된 소켓 저장.
    SOCKET client_sock = (SOCKET)arg;
    //4. 클라이언트 정보 얻기
    addrlen = sizeof(clientaddr);
    getpeername(client_sock, (SOCKADDR*)&clientaddr, &addrlen);
    //클라이언트와 데이터 통신
    while(1){ ... }
}
```

3. MULTY THREAD 구현

1. 클라이언트가 접속하면 **accept()** 함수는 클라이언트와 통신할 수 있는 소켓을 리턴 한다.

2. 클라이언트와 통신을 담당할 **Thread**를 생성한다. 이때 **Thread**함수에 소켓을 넘긴다.

3. **Thread**함수는 인자로 전달된 소켓을 **SOCKET** 타입으로 **Casting**하여 저장해 둔다.

4. **getpeername()**함수를 호출해 클라이언트의 **IP**주소와 **PORT**번호를 얻는다, 이 부분은 클라이언트의 정보 출력을 위한 부분이다.

5. 클라이언트와 데이터를 주고 받는다.

클라이언트 소켓 정보를 알아보는 함수

int getpeername(SOCKET s, struct sockaddr* name, int* namelen);
- 연결 대상(클라이언트라면 서버, 서버라면 클라이언트)의 **IP, PORT**

int getsockname(SOCKET s, struct sockaddr* name, int* namelent);
- 본인의 **IP, PORT**

- 첫 번째 인자 : 소켓.
- 두 번째 인자 : 소켓 주소 구조체
- 세 번째 인자 : 소켓 주소 구조체의 크기

Multy Thread 프로젝트를 참고하자!!

4. THREAD 동기화

4. THREAD 동기화

Thread 동기화의 필요성

공유 변수

int money = 1000

Thread 1

1.read money into ECX
2.ECX = ECX + 2000
3.Write ECX into money

Thread 2

1.read money into ECX
2.ECX = ECX + 4000
3.Write ECX into money

Thread 1이 **1**번과정을 수행한 상태에서 정지되고 **Thread2**가 **1~3**번 과정을 수행하면**, money** 값은 **5000**이 된다**.** 다시 **Thread1**이 **CPU** 시간을 할당 받아 **2~3** 과정을 수행하면**, ECX**에 저장되어 있던 값에 **2000**이 더해지고 이 값이 메모리에 저장되어 **money**값은 **3000**이 된다**.** 결과적으로 선처리한 **Thread2**의 과정은 삭제되는 것이다**,** 이 같은 문제를 해결 하기 위해 **Thread**동기화 를 한다**.**

종류	기능
임계 영역(criticla section)	공유 자원에 대해 오직 한 **Thread**의 접근만 허용 **-** 한 프로세스에 속한 **Thread** 간에만 사용 가능
뮤텍스(mutex)	공유 자원에 대해 오직 한 **Thread**의 접근만 허용 **-** 서로 다른 프로세스에 속한 **Thread** 간에도 사용 가능
이벤트**(event)**	사건 발생을 알려 대기 중인 **Thread**를 깨운다**.**
세마포어**(semaphore)**	한정된 개수의 자원에 여러 **Thread**가 접근할 때**,** 자원을 사용할 수 있는 **Thread** 개수를 제한한다**.**
대기 기능 타이머**(waitable timer)**	정해진 시간이 되면 대기중인 **Thread**를 깨운다**.**

4. THREAD 동기화

Thread 동기화 기본 개념

- **동기화가 필요한 상황**
- 둘 이상의 **Thread**가 공유자원에 접근.
- 한 **Thread**가 작업을 완료한 후, 기다리고 있는 다른 **Thread**에 알려준다.

- **동기화 객체(synchronization object)**
  - **Windows OS**에서의 매개체
  - 각 **Thread**가 독립적으로 실행하지 않고 다른 **Thread**와 상호 작용을 토대로 자신의 작업을 진행하게 되는데 **Thread**를 동기화 하기 위해서는 각 **Thread**가 상호작용해야 하므로 이것을 조율할 중간 매체가 필요하다.

- **동기화 객체의 특징**
  - **Create*()**함수를 호출하면 커널**(kernel :** 운영체제의 핵심 부분을 뜻함**)** 메모리 영역에 동기화 객체가 생성되고, 이에 접근할 수 있는 핸들**(HANDLE** 타입**)**이 리턴된다.
  - 평소에는 비신호 상태**(non-singaled state)**로 있다가 특정 조건이 만족되면 신호 상태**(signaled state)**가 된다. 비신호 상태에서 신호 상태로 변화 여부는 **Wait*()**함수를 사용해 감지할 수 있다.
  - 사용이 끝나면 **CloseHandle()**함수를 호출한다.

4. THREAD 동기화

임계영역(critical section) – ThreadSync Project를 확인해보자

- 둘 이상의 **Thread**가 공유 자원에 접근할 때, 오직 한 **Thread**만 접근을 허용해야 하는 경우에 사용한다.
- 일반 동기화 객체와 달리 개별 프로세스의 유저 메모리 영역에 존재하는 단순한 구조체다, 따라서 다른 프로세스가 접근할 수 없으므로 한 프로세스에 속한 **Thread** 간 동기화에만 사용된다.

```
CRITICAL_SECTION cs; // 1

DWORD WINAPI MyThread1(LPVOID arg)
{
    ...
    EnterCriticalSection(&cs); // 3
    //공유 자원 접근
    LeaveCriticalSection(&cs); // 4
    ...
}

DWORD WINAPI MyThread2(LPVOID arg)
{
    ...
     EnterCriticalSection(&cs); // 3
    //공유 자원 접근
    LeaveCriticalSection(&cs); // 4
    ...
}
```

```
Int main(int argc, char* argv[])
{
    ...
    InitializeCriticalSection(&cs); // 2
    // Thread를 두 개 이상 생성해 작업을 진행한다.
    // 생성한 모든 Thread가 종료할 때까지 기다린다.
    DeleteCriticalSection(&cs); // 5
    ...
}
```

1. **CRITICAL_SECTION** 구조체 변수를 전역 변수로 선언한다. 일반 동기화 객체는 **Create*()** 함수를 호출해 커널 메모리 영역에 생성하지만, 임계 영역은 유저 메모리 영역에(대개는 전역변수 형태로)생성한다.
2. 임계 영역을 사용하기 전에 **InitializeCriticalSection()**함수를 호출해 초기화한다.
3. 공유 자원에 접근하기 전에 **EnterCriticalSection()**함수를 호출한다. 공유 자원을 사용하고 있는 **Thread**가 없다면 **EnterCriticalSection()** 함수는 곧바로 리턴한다. 하지만 공유 자원을 사용하고 있는 **Thread**가 있다면 **EnterCriticalSection()**함수는 리턴하지 못하고 **Thread**는 대기 상태가 된다.
4. 공유 자원 사용을 마치면 **LeaveCriticalSection()** 함수를 호출한다. 이때 **EnterCriticalSection()**함수에서 대기중인 **Thread**가 있다면 하나만 선택되어 깨어난다.
5. 임계 영역을 사용하는 모든 **Thread**가 종료하면 **DeleteCriticalSection()** 함수를 호출해 삭제한다.

Chating Project와 **MutiThreadServerPacket Project**를 확인하고

- 콘솔 버전 **오목을 온라인**으로 만들어보자
- 콘솔 채팅을 패킷으로 변경하고 **API를 이용한 채팅 프로그램**을 만들어보자
  - 대기방이나 닉네임 같은 설정은 필요 없고 채팅을 쓰고 갱신하고 하는 과정만 보여주자