네트워크 3

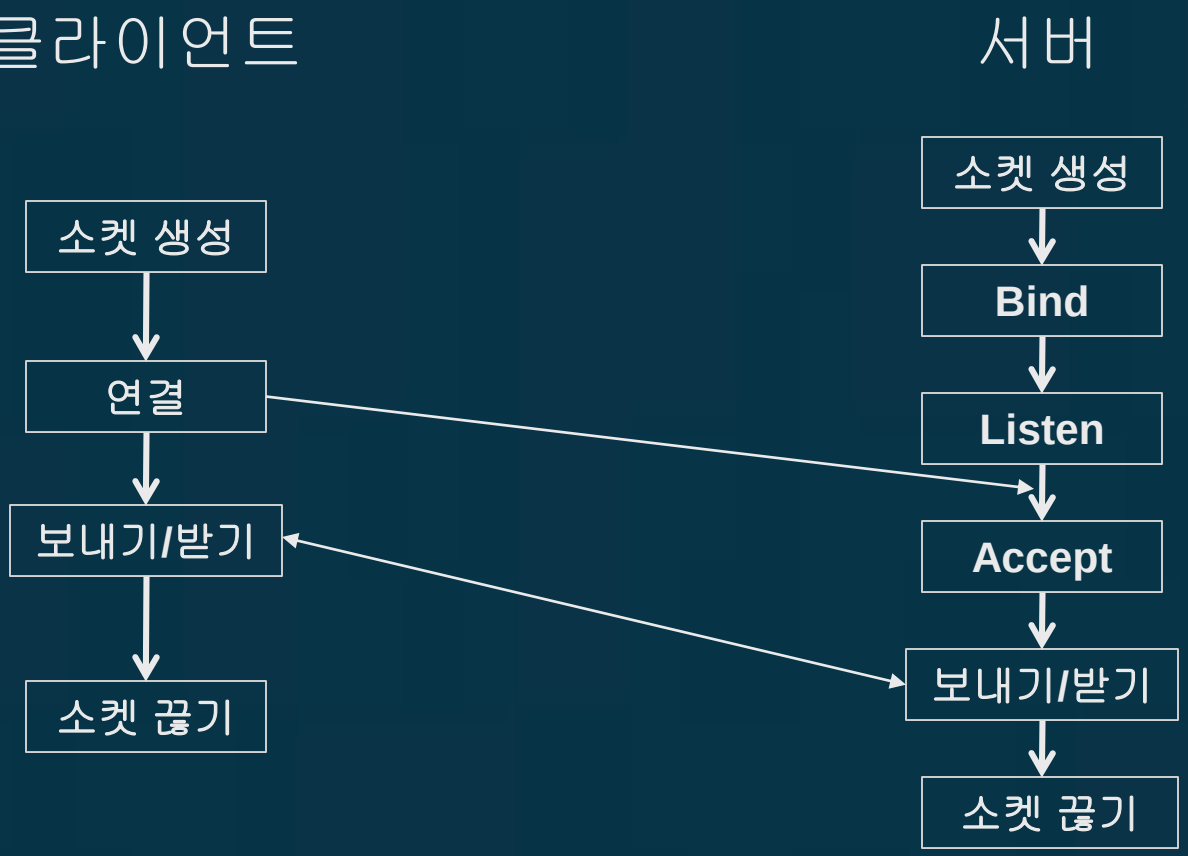
MM4220 게임서버 프로그래밍

정내훈

내용

- 네트워크 복습
- Windows I/O 모델
- 숙제

기본 프로그래밍


클라이언트
서버
소켓 생성
연결
보내기/받기
소켓 끊기
소켓 생성
Bind
Listen
Accept
보내기/받기
소켓 끊기

Windows I/O 모델

- I/O 모델이 필요한 이유
  - 비 규칙적인 입출력 관리
  - 다중 접속 관리
- 게임 서버의 접속
  - 정해지지 않은 동작 순서
    - 그래도 멈추지 않아야 하는 게임
  - 수 천 개의 접속
  - 상대적으로 낮은 접속당 bandwidth
    - 효율적 자원관리 필요

Windows I/O 모델

- Socket Thread
- Non-blocking I/O
- Select
- WSAAsyncSelect
- WSAEventSelect
- Overlapped I/O (Event)
- Overlapped I/O (Callback)
- I/O Completion Port

Windows I/O 모델

- Socket Thread
  - Thread를 통한 처리

Windows I/O 모델

- Thread를 통한 처리

```
while (!shutdown) {
    new_sock = accept(sock, &addr, &len);
    thread t = thread { do_io, new_sock };
}
```

```
do_io(mysock) {
while (true) {
  recv(mysock)
  process_packet(); }}
```

  - 다중 소켓 처리 가능
  - 과도한 thread 개수로 인한 운영체제 Overhead
    - thread당 오버헤드 : Thread Control Block, **Stack**

Windows I/O 모델

- Non-blocking I/O

```
unsigned long noblock = 1;
int nRet = ioctlsocket(sock, FIONBIO, &noblock);
```

  - Socket의 모드를 blocking에서 non-blocking으로 변환
  - Socket 함수 호출이 즉시 끝나지 않을 때
    - WSAEWOULDBLOCK 에러를 내고 끝난다.
    - 기다리지 않는다
  - Busy Waiting
    - recv를 돌아가면서 반복 check해야 함.
    - CPU낭비 -> 성능 저하

Windows I/O 모델

- Non-blocking I/O

```
While (true) {
    while (true) {
        i++;
        i = i % 동접;
        recv(socket[i],…);
        if (no_error) break;
    }
    패킷 처리
}
```

Windows I/O 모델

- Select

```
int select(
        __in int nfds,
        __inout fd_set* readfds,
        __inout fd_set* writefds,
        __inout fd_set* exceptfds,
        __in const struct timeval* timeout );
```

  - nfds : 무시
  - readfds : 읽기 가능 검사용 소켓 집합 포인터
  - writefds : 쓰기 가능 검사용 소켓 집합 포인터
  - exceptfds : 에러 검사용 소켓 집합 포인터
  - timeout : select가 기다리는 최대 시간
  - return value : 사용 가능한 소켓의 개수

Windows I/O 모델

- Select
  - Unix시절부터 내려온 고전적인 I/O 모델
  - unix나 linux에서는 socket 개수의 한계 존재
    - unix: 64, linux: 1024
  - socket의 개수가 많아질수록 성능 저하 증가
    - linear search

```
FD_SET(sock1, &rfds);
FD_SET(sock2, &rfds);

select(0. &rfds, NULL, NULL, &time);

if (FD_ISSET(sock1, &rfds)) recv(sock1, buf, len, 0)
if (FD_ISSET(sock2, &rfds)) recv(sock2, buf, len, 0)
```

Windows I/O 모델

- WSAAsyncSelect
  - 소켓 이벤트를 특정 윈도우의 메시지로 받는다

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

  - s : 소켓
  - hWnd : 메시지를 받을 윈도우
  - wMsg : 메시지 번호
  - lEvent : 반응 event 선택

Windows I/O 모델

- WSAAsyncSelect
  - 클라이언트에 많이 쓰임
    - 윈도우 필요
    - 윈도우 메시지 큐를 사용 -> 성능 느림

lEvent

비트 값	의미
FD_READ	Recv 할 데이터가 있음
FD_WRITE	Send할 수 있는 버퍼 공간 있음
FD_OOB	Out-of-band 데이터 있음
FD_ACCEPT	Accept 준비가 됨
FD_CONNECT	접속이 완료됨
FD_CLOSE	소켓연결이 종료됨

Windows I/O 모델

- WSAAsyncSelect
  - 자동으로 non-blocking mode로 소켓 전환
  - 아래와 같은 함수로 이벤트 처리

```
LRESULT CAsyncSelectDlg::OnSocketMsg(WPARAM wParam, LPARAM lParam)
{
        SOCKET sock=(SOCKET)wParam;
        int nEvent = WSAGETSELECTEVENT(lParam);
        switch(nEvent){
                case FD_READ :
                case FD_ACCEPT :
                case FD_CLOSE :
```

Windows I/O 모델

- WSAEventSelect

```
int WSAEventSelect(
        __in SOCKET s,
        __in WSAEVENT hEventObject,
        __in long lNetworkEvents );
```

  - s : 소켓
  - hEventObject :
  - lNetworkEvents : WSAAsyncSelect와 같음
  - 메시지를 처리할 윈도우가 필요 없음.

Windows I/O 모델

- WSAEventSelect
  - socket과 event의 array를 만들어서 `WSAWaitForMultipleEvents()` 의 리턴값으로 부터 socket 추출
  - 소켓의 개수 64개 제한!
    - 멀티 쓰레드를 사용해서 제한 극복가능

Windows I/O 모델

- WSAEventSelect
  - 다음의 API로 socket대기 상태 검출

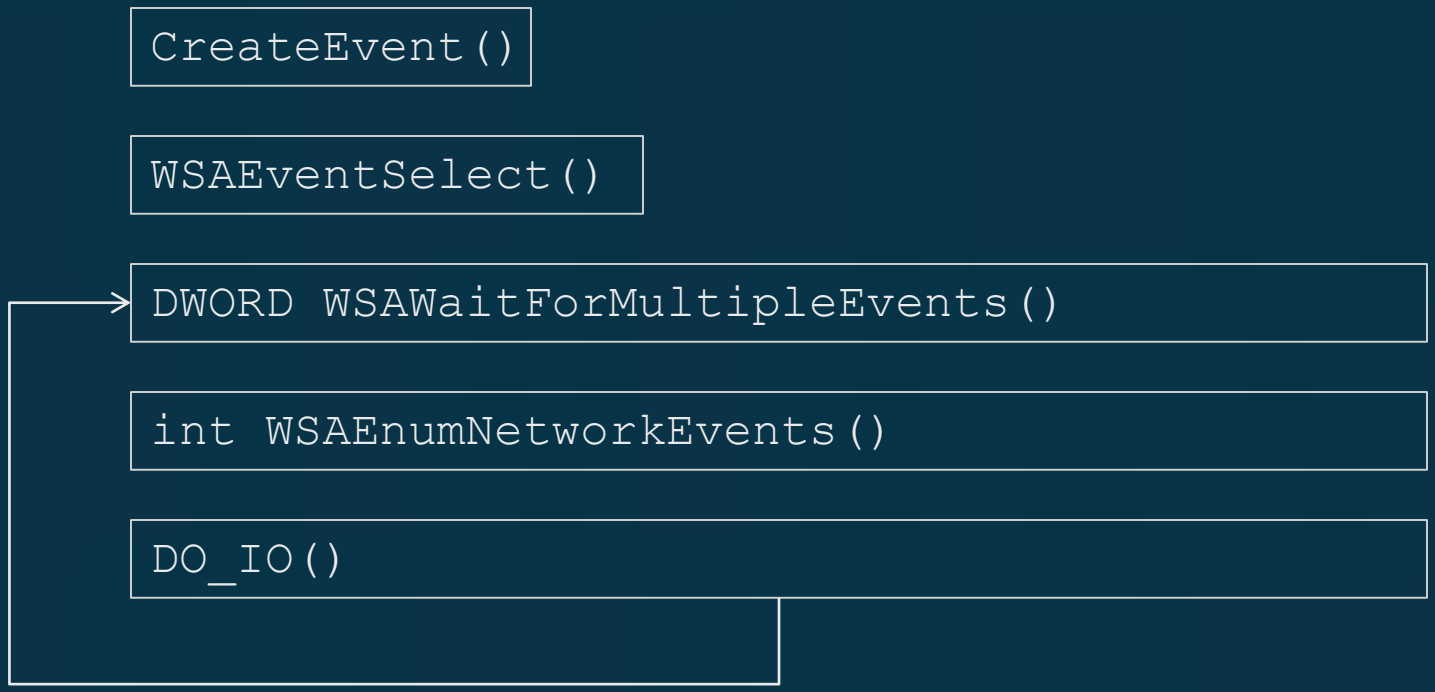
```
DWORD WSAWaitForMultipleEvents(
        DWORD nCount,
        const WSAEVENT *lphEvents,
        BOOL bWaitAll,
        DWORD dwMiliseconds);
```

```
int WSAEnumNetworkEvents(SOCKET s,
        WSAEVENT hEventObject;,
        LPWSANETWORKEVENTS lpNetworkEvents);
```

```
lpNetworkEvents->lNetworkEvents;
```

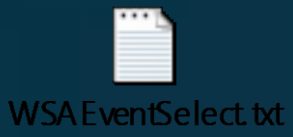
Windows I/O 모델

- WSAEventSelect : 동작



Windows I/O 모델

- WSAEventSelect : 예제
  - http://perfectchoi.blogspot.com/2009/09/wsaeventselect.html



Windows I/O 모델

- Overlapped I/O 모델
  - Windows에서 추가된 고성능 I/O 모델
  - 다른 이름으로는 Asynchronous I/O 또는 비동기 I/O
    - 리눅스의 경우 boost/asio 라이브러리로 사용 가능
  - 대용량 고성능 네트워크 서버를 위해서는 필수 기능
  - IOCP도 Overlapped I/O를 사용
  - 사용 방법이 select style의 I/O 모델과 다르다.

Windows I/O 모델

- Overlapped I/O 모델

```
While (true) {
    select(&recv_ready_sockets);
    for (i : ready_sockets) {
        recv(socket[i],…);
        패킷 처리 for i;
    }
}
```

non-overlapped I/O

overlapped I/O

```
for (i : all_socket)
    recv(socket[i], buf[i]);
While (true) {
    i = wait_for_recv_complete_socket();
    패킷 처리 for i;
    recv(socket[i], …);
}
```

Windows I/O 모델

- Overapped I/O 모델
  - 비 동기 데이터 송수신을 할 수 있다
  - 소켓 내부 버퍼를 사용하지 않고 직접 사용자 버퍼에서 데이터를 보내고 받을 수 있다. (옵션)
    - 버퍼 크기를 0으로 했을 때

```
int result;
int buffsize = 0;
result = setsockopt(s, SOL_SOCKET, SO_SNDBUF, &buffsize, sizeof(buffsize));
```

- 비 동기? Non-blocking과의 차이는?
  - 앞의 다중 I/O 모델들은 recv와 send의 **가능 여부**만 비동기
  - Overapped는 아예 여러 소켓의 send, recv **실행 자체를 동시**에 수행

Windows I/O 모델

- Overapped I/O 모델
  - Send와 Recv를 호출했을 때 패킷 송수신의 완료를 기다리지 않고 Send, Recv함수 종료
  - 이때 Send와 Recv는 단순한 송수신의 시작을 지시하는 함수
  - 여러 번 Recv, Send 를 실행함으로써 여러 소켓에 대한 동시 다발적 Recv, Send도 가능
    - 하나의 socket은 하나의 recv만 가능!!!

Windows I/O 모델

- Overlapped I/O
  - SOCKET WSASocket(int af, int type, int protocol, LPWSAPROTOCOL_INFO lpProtocolInfo, GROUP g, DWORD dwFlags)
    - af : address family
      - AF_INET만 사용 (AF_NETBIOS, AF_IRDA, AF_INET6)
    - type : 소켓의 타입
      - tcp를 위해 SOCK_STREAM사용 (SOCK_DGRAM)
    - protocol : 사용할 프로토콜 종류
      - IPPROTO_TCP (IPPROTO_UDP)
    - lpProtocolInfo : 프로토콜 정보
      - 보통 NULL
    - g : 예약
    - dwFlags : 소켓의 속성
      - 보통 0 (또는 WSA_FLAG_OVERLAPPED)

Windows I/O 모델

- Overlapped I/O
  - int WSARecv(SOCKET s, LPWSABUF lpBuffers, DWORD dwBufferCount, LPDWORD lpNumberofBytesRecvd, LPDWORD lpFlags, LPWSAOVERLAPPED lpOverlapped, LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine)
    - s : 소켓
    - lpBuffers : 받은 데이터를 저장할 버퍼
    - dwBufferCount : 버퍼의 개수
    - lpFlags : 동작 옵션(MSG_PEEK, MSG_OOB)
    - lpNumberofBytesRecvd : 받은 데이터의 크기 => NULL
    - lpOverlapped, lpCompletionRoutine : 뒤에 설명

Windows I/O 모델

- Overlapped I/O
  - LPWSAOVERLAPPED lpOverlapped

```
typedef struct WSAOVERLAPPED {
        DWORD Internal;
        DWORD InternalHigh;
        DWORD Offset;
        DWORD OffsetHigh;
        WSAEVENT hEvent;
} WSAOVERLAPPED, FAR *LPWSAOVERLAPPED;
```

    - Internal, InternalHigh, Offset, OffsetHigh : 0으로 초기화 후 사용
    - hEvent I/O가 완료 되었음을 알려주는 event 핸들
  - LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
    - callback 함수, 뒤에 설명

Windows I/O 모델

- Overlapped I/O 모델
  - Overlapped I/O가 언제 종료되었는지를 프로그램이 알아야 함
  - 두 가지 방법이 존재
    - Overlapped I/O Event모델
    - Overlapped I/O Callback모델

Windows I/O 모델

- Overlapped I/O Event 모델
  - WSARecv의 LPWSAOVERLAPPED lpOverlapped 구조체의 WSAEVENT hEvent 사용
  - 작업 결과 확인
    - WSAGetOverlappedResult()

Windows I/O 모델

- Overlapped I/O Event 모델
  - WSAGetOverlappedResult()

```
BOOL WSAGetOverlappedResult(
        SOCKET s,
        LPWSAOVERLAPPED lpOverlapped,
        LPDWORD lpcbTransfer,
        BOOL fWait,
        LPDWORD lpdwFlags);
```

    - s :socket
    - lpOverlapped : WSARecv에 넣었던 구조체
    - lpcbTransfer : 전송된 데이터 크기
    - fWait : 대기 여부
    - lpdwFlags : Recv의 lpFlag의 결과

Windows I/O 모델

- Overlapped I/O Event 모델
  - 1. WSACreateEvent()를 사용해서 이벤트 생성
  - 2. WSAOVERLAPPED구조체 변수선언 0으로 초기화 hEvent에 1의 이벤트
  - 3. WSASend(), WSARecv()
    - 2의 구조체를 WSAOVERLAPPED에
      - 중복 사용 불가능!! 호출 완료 후 재사용
    - lpCompletionROUTINE에 NULL
  - 4. WSAWaitForMultipleEvents()함수로 이벤트 감지
  - 5. WSAGetOverlappedResult()함수로 이벤트완료 확인

Windows I/O 모델 (2019 수목학기)

- Overlapped I/O Callback 모델
  - 이벤트 제한 개수 없음
  - 사용하기 편리
  - WSARecv와 WSASend의 LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine 함수 사용
  - Overlapped I/O가 끝난후 lpCompletionRoutine이 호출됨

Windows I/O 모델

- Overlapped I/O Callback 모델
  - Callback함수

```
void CALLBACK CompletionROUTINE(
        DWORD dwError,
        DWORD cbTransferred,
        LPWSAOVERLAPPED lpOverlapped,
        DWORD dwFlags);
```

  - dwError : 작업의 성공 유무
  - cbTransferred : 전송된 바이트 수
  - lpOverlapped : WSASend/WSARecv에서 사용한 구조체
  - dwflags : WSASend/WSARecv에서 사용한 flag

실습 (2019 화목)

- 예제 프로그램 :
  - http://myblog.opendocs.co.kr/archives/1204
  - 64비트 용으로 수정한 것을 게임공학부 자료실 게시판에서 다운로드.

Windows I/O 모델

- Overlapped I/O
  - Overlapped I/O 구분

lpOverlapped	hEvent	lpCompletionRoutine	Completion 여부 식별
NULL	세팅 불가	무시됨	동기적 실행
Not-NULL	NULL	NULL	Overlapped 동작, 완료 검사 불가능
Not-NULL	Not-NULL	NULL	Overlapped 동작, Event객체로 완료 검사
Not-NULL	무시됨	Not-NULL	Overlapped 동작, completion routine을 통해서 완료 관리

Windows I/O 모델

- IOCP
  - Windows I/O모델 중 최고의 성능
  - 별도의 커널 객체를 통한 구현
    - IOCP객체를 생성해서 핸들을 받아 사용.
  - 기본적으로 Overlapped I/O CallBack
    - CallBack 함수들을 멀티쓰레드로 동시에 실행
  - IOCP객체 내부 Thread Pool사용
    - Thread생성 파괴 오버헤드 없앰
    - 적은 수의 thread로 많은 연결을 관리
  - IOCP객체 내부 Device LIst 사용
    - 등록된 소켓에 대한 I/O는 IOCP가 처리

Windows I/O 모델

- IOCP 사용이 어려운 이유
  - Overlapped I/O로만 동작
    - Overlapped I/O를 모르면 이해할 수 없음.
  - 비 직관적인 API
    - 하나의 API를 여러 용도로 사용
      - 파라미터에 따라 완전히 다르게 동작하는 API
      - API이름과 아무 관계도 없는 동작을 하는 경우가 있음.
    - 뜬금없는 API 파라미터
      - 하나의 API를 여러 용도로 사용하기 때문
      - 파라미터로 넘어 오는 정보들이 불완전함 -> 편법으로 보완 필요

Windows I/O 모델

- IOCP – 준비

```
HANDLE CreateIoCompletionPort(
        HANDLE FileHandle,
        HANDLE ExistingCompletionPort,
        ULONG_PTR CompletionKey,
        DWORD NumberOfConcurrentThreads };
```

– IOCP커널 객체 생성

```
HANDLE hIOCP = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, NULL, 0};
```

- 마지막 0 : core 개수 만큼 사용

Windows I/O 모델

- IOCP – 준비
  - IOCP객체와 소켓연결

```
HANDLE CreateIoCompletionPort(socket, hIOCP, key, 0};
```

    - key값은 unique하게 임의로 설정
    - 마지막 값은 무시
  - Worker Thread생성

```
thread { worker_thread };
```

Windows I/O 모델

- IOCP – 준비
  - Worker Thread

```
while(1) {
        GetQueuedCompletionStatus(
                hIOCP,
                &dwIOSize,
                &key,
                &lpOverlapped,
                INFINITE;
        );
        lpOverlapped에 따른 처리;
}
```

Windows I/O 모델

- IOCP – 준비
  - Worker Thread

```
BOOL GetQueuedCompletionStatus(
        HANDLE CompletionPort,
        LPDWORD lpNumberOfByte,
        PULONG_PTR lpCompletionKey,
        LPOVERLAPPED *lpOverlapped,
        DWORD dwMillisocnds);
```

  - I/O완료 상태를 report
  - Completion Port : 커널 옵젝트
  - lpNumberofByte : 전송된 데이터 양
  - lpCompletionkey : 미리 정해놓은 ID
  - lpOverlapped : Overlapped I/O 구조체

Windows I/O 모델

- IOCP 그리고
  - 이벤트 추가함수

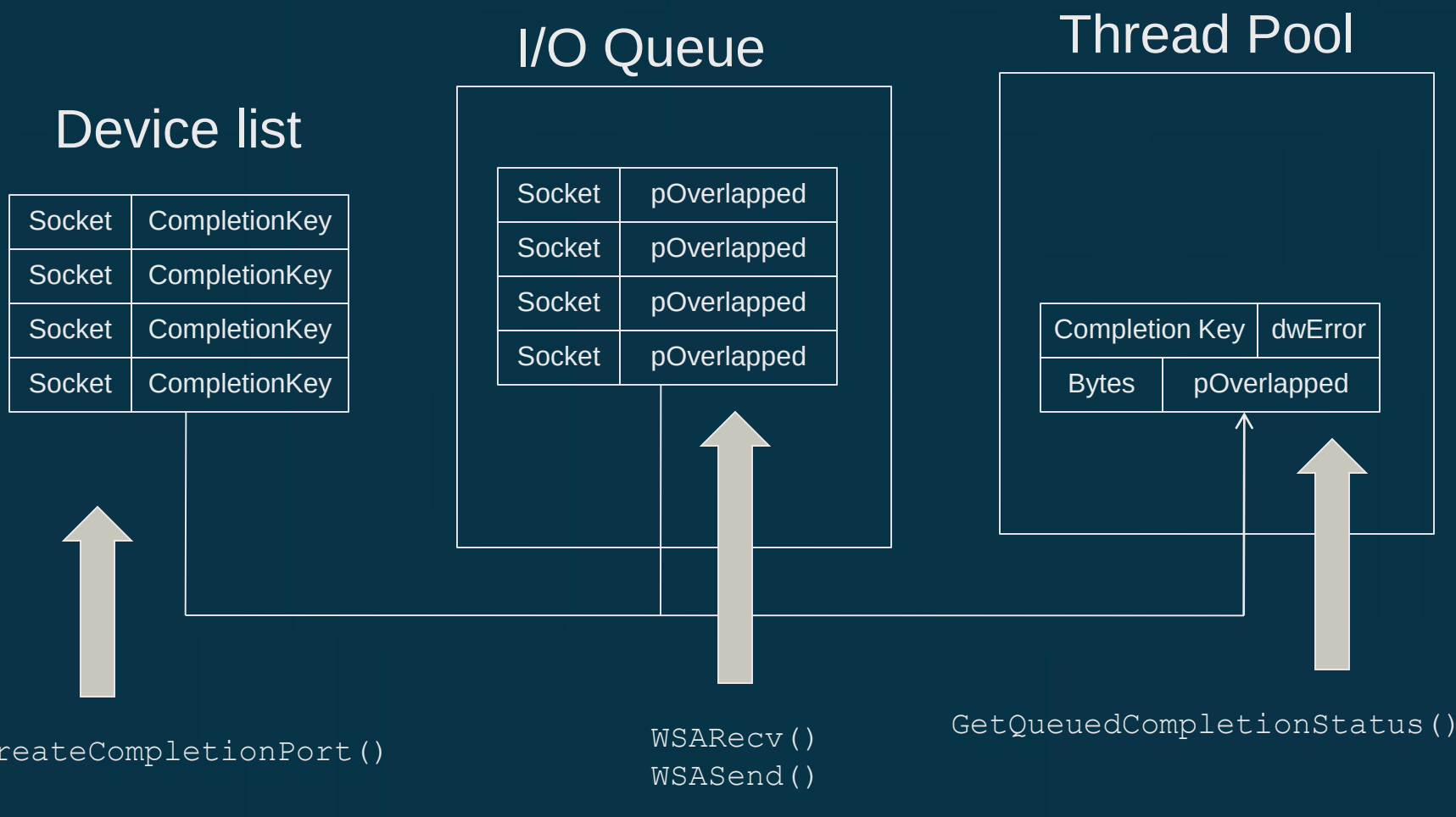
```
BOOL PostQueuedCompletionStatus(
        HANDLE CompletionPort,
        DWORD NumberOfByte,
        ULONG_PTR dwCompletionKey,
        LPOVERLAPPED lpOverlapped);
```

  - 커널의 Queue에 이벤트 추가
  - Completion Port : 커널 옵젝트
  - NumberofByte : 전송된 데이터 양
  - dwCompletionkey : 미리 정해놓은 ID
  - lpOverlapped : Overlapped I/O 구조체

Windows I/O 모델

- IOCP 그리고
  - `PostQueuedCompletionStatus()`의 용도
    - IOCP를 사용할 경우 IOCP가 main loop가 되기 때문에 socket I/O 이외에도 모든 다른 작업할 내용을 추가 할 때 쓰인다.
      - 예) timer

Windows I/O 모델



Windows I/O 모델

I/O Model	장점	단점
select	호환성	불편하고, 느리다
WSAAsyncSelect	윈도우 메시지 기반이라 친숙하다	성능이 떨이지고 윈도우가 필수
WSAEventSelect	비교적 사용하기 쉽고 성능이 좋다	64개 소켓 제한
Overlapped I/O (Event)	IO 동시작업으로 성능 향상	64개 이벤트 제한, 개별 작업 결과 확인 필요
Overlapped I/O (Callback)	64개 이벤트제한 없음. 프로그램 간단	대규모 연결에는 아직 부족
IOCP	대규모 연결 처리 가능	사용하기 어렵다

숙제 (#3)

- 게임 서버/클라이언트 프로그램 작성
  - 내용
    - 숙제 (#2)의 프로그램의 다중 사용자 버전
    - Client/Server 모델, 서버는 반드시 Overlapped I/O callback 을 사용할 것
    - 클라이언트 10개 까지 접속 가능 하게 수정
      - 옆의 클라이언트 에서도 다른 클라이언트의 말의 움직임이 보임
  - 목적
    - Windows 다중 접속 Network I/O 습득
  - 제약
    - Windows에서 Visual Studio로 작성 할 것
    - 그래픽의 우수성을 보는 것이 아님
  - 제출
    - 다음 화요일 (3월 26일) 오후 1시
    - 제목에 “2019 게임서버 [화목] 학번 이름 숙제 3번” 또는 “2019 게임서버 [수목] 학번 이름 숙제 3번”
    - Zip으로 소스를 묶어서 e-mail로 제출

IOCP(2019-화목, 수목)

- IOCP API

```
CreateIoCompletionPort();
```

  - IOCP객체 생성
  - Socket을 IOCP에 연결

```
GetQueuedCompletionStatus();
```

  - Thread를 IOCP의 thread-pool에 등록하고 멈춤
  - IOCP로 부터 I/O결과를 얻어옴

IOCP

- IOCP 서버 설계
  1. 초기화
     - IOCP 핸들 생성
  2. Worker thread 생성
     - Thread들을 IOCP thread-pool에 등록
  3. Accept thread 생성
     - Accept무한 루프
     - 새로운 소켓이 접속하면 IOCP에 연 결 후 WSARecv호출

IOCP

- Accept thread
  - 새로 접속해 오는 클라이언트를 IOCP로 넘기는 역할
  - 무한 루프를 돌면서
    - Accept() 호출
    - 새 클라이언트가 접속했으면 **클라이언트 정보 구조체**를 만든다.
    - IOCP에 소켓을 등록한다.
      - Send/recv가 IOCP를 통해 수행됨
    - WSARecv()를 호출한다.
      - Overlapped I/O recv 상태를 항상 유지해야 한다.

IOCP

- 클라이언트 객체
  - 서버는 클라이언트의 정보를 갖고 있는 객체가 필요
    - 최대 동접과 같은 개수가 필요
    - 필요한 정보 : ID, socket, 상태, 게임정보(name, HP, x, y)
- GetQueuedCompletionStatus를 받았을 때 클라이언트 객체를 찾을 수 있어야 한다.
  - IOCP에서 오고 가는 것은 completion_key와 overlapped I/O pointer, number of byte 뿐
  - Completion_key를 클라이언트 객체의 포인터로 하거나 클라이언트 객체의 ID 혹은 index로 한다.

IOCP

- **Overlapped 구조체**
  - 모든 Send, Recv는 Overlapped 구조체가 필요.
  - 하나의 구조체를 동시에 여러 호출에서 사용하는 것을 불가능
  - 소켓당 Recv 호출은 무조건 한 개여야 한다.
    - Recv 호출 용 Overlapped 구조체 한 개가 있어서 계속 재사용하는 것이 바람직 (new/delete overhead 제거)
  - 소켓당 Send 호출은 동시에 여러 개가 될 수 있다.
    - Send 버퍼도 같은 개수가 필요하다.
    - 개수의 제한이 없으므로 new/delete로 사용
      - Send 할 때 new, Send가 complete되었을 때 delete
    - 성능을 위해서는 공유 Pool을 만들어서 관리할 필요가 있다.

IOCP

- Overlapped I/O pointer를 확장
  - Overlapped I/O 구조체 자체는 쓸만한 정보가 없다.
  - 따라서 정보들을 더 추가할 필요가 있다.
    - 뒤에 추가하면 IOCP는 있는지 없는지도 모르고 에러도 나지 않는다. (pointer만 왔다 갔다 하므로)
  - 꼭 필요한 정보
    - 지금 이 I/O가 send인지 recv인지????
    - I/O Buffer의 위치 (Send할 때 버퍼도 같이 생성되므로)

IOCP

- WorkerThread
  - 무한루프
    - GetQueuedCompletionStatus를 부른다.
    - 에러처리/접속종료처리를 한다.
    - Send/Recv처리를 한다.
      - 확장 Overlapped I/O 구조체를 유용하게 사용한다.
      - Recv
        - 패킷이 다 왔나 검사 후 다 왔으면 패킷 처리
        - 여러 개의 패킷이 한번에 왔을 때 처리
        - 계속 Recv호출
      - Send
        - Overlapped 구조체, 버퍼의 free(혹은 재사용)

IOCP

- 버퍼관리
  - Recv
    - 하나의 소켓에 대해 Recv호출은 언제나 하나이기 때문에 하나의 버퍼를 계속 사용할 수 있다.
    - 패킷들이 중간에 잘려진 채로 도착할 수 있다.
      - 모아 두었다가 다음에 온 데이터와 붙여주어야 한다.
      - 남은 데이터를 저장해 두는 버퍼 필요, 또는 Ring Buffer를 사용할 수도 있다.
    - 패킷들이 여러 개 한꺼번에 도착할 수 있다.
      - 잘라서 처리해야 한다.

IOCP

- 버퍼관리
  - Send
    - 하나의 Socket에 여러 개의 send를 동시에 할 수 있다.
      - MULTI-THREAD!
      - overlapped구조체와 WSABUF는 중복 사용 불가능!
    - Windows는 send를 실행한 순서대로 내부 버퍼에 넣어놓고 전송한다.
    - 내부 버퍼가 차서 Send가 중간에 잘렸다면??
      - 나머지를 다시 보내면 된다.
      - 다시 보내기 전 다른 쓰레드의 Send가 끼어들었다면??
      - 해결책
        - 모아서 차례 차례 보낸다. send 데이터 버퍼 외에 패킷 저장 버퍼를 따로 둔다. (성능 저하)
        - 또는, 이런 일이 벌어진 소켓을 끊어버린다.

IOCP 서버 구현

- 먼저 할 일
  - 다중 접속 관리
    - 클라이언트 접속 시 마다 ID 부여
  - 패킷 포맷 및 프로토콜 정의
    - 기본 패킷 포맷
      - 길이 (Byte) + 타입 (Byte) + Data (….)
    - Client -> Server
      - 이동 패킷
    - Server -> Client
      - 위치 지정, ID 접속 알림, ID 로그아웃 알림

IOCP 서버 구현

- 먼저 할 일
  - 패킷 처리 루틴 작성

```
bool PacketProcess(const unsigned char* pBuf,
                                        int client_id);
```

IOCP 서버 구현

- Recv의 구현

Start :
    모든 데이터를 처리했으면 Goto 종료
    남는 데이터로 패킷을 완성할 수 있느가?
        예 : 패킷 버퍼 완성 , 패킷 처리 함수 호출
        아니오 : 남는 데이터 모두 패킷 버퍼로 전송
    goto Start
종료:
    Recv를 호출

IOCP 서버 구현

- Recv의 구현
  - overapped 구조체의 확장

```
struct stOverEx{
   WSAOVERLAPPED m_wsaOver;
   WSABUF        m_wsaBuf;
   unsigned char m_IOCPbuf[MAX_BUF_SIZE];  // IOCP send/recv 버퍼
   enumOperation m_eOperation;             // Send중인가Recv중인가.
};
```

  - 클라이언트 정보에 추가될 내용

```
class ClientInfo{
   …
   stOverEx      m_over;
   unsigned char m_recv_packet_buf[MAX_PACKET_SIZE];// 패킷이 조립되는버퍼
   int           m_prev_size;                       // 이전에 받아 놓은 패킷의 양
};
```

IOCP 서버 구현

- Recv의 구현

```
unsigned char *buf_ptr = pOverEx->m_IOCPbuf;
int restDataSize = dwIoSize;
int packet_size = 0;
if (0 != client->m_prev_size) packet_size = (int) client->m_recv_packet_buf[0];
while(restDataSize) {
          if (0 == packet_size) packet_size = (int) buf_ptr[0]; // 패킷사이즈 확정
          int required = packet_size - client->m_prev_size;
          if (restDataSize < required) { // 더 이상 패킷을 만들 수 없다. 루프를 중지한다.
                    memcpy(client->m_recv_packet_buf + client->m_prev_size,
                              buf_ptr,  restDataSize);
                    client->m_prev_size += restDataSize;
                    break;
          } else { // 패킷을 완성할 수 있다.
                    memcpy(client->m_recv_packet_buf + client->m_prev_size,
                              buf_ptr,  required);
                    bool ret = PacketProcess(client->m_recv_packet_buf, client);
                    client->m_prev_size = 0;
                    restDataSize -= required;
                    buf_ptr += required;
                    packet_size = 0; // 다음 패킷의 크기가 확실하지 않다.
          }
}
WSARecv(…);
```

IOCP 서버 구현

- WSASend의 사용
  - Send는 여러 Thread에서 동시 다발적으로 발생
  - Send하는 overlapped 구조체와, buffer는 send가 끝날 때 까지 유지되어야 한다.
    - 개수를 미리 알 수 없으므로 Dynamic하게 관리해야 한다.
  - 부분 send인 경우
    - 버퍼가 비워지지 않은 경우
    - 에러처리하고 끊어버려야 한다.
      - 현재 운영체제의 메모리가 꽉 찬 경우
      - 이러한 일이 벌어지지 않으려면 send하는 데이터의 양을 조절해야 한다.

IOCP 서버 구현

- Worker Thread에서의 Send의 구현
  - Overapped구조체와 Buffer를 해제 시켜야 한다.
    - 메모리 재사용.
    - 모든 자료구조를 확장 Overalapped 구조체에 넣었으므로.

```
if (dwIoSize < pOverEx->m_IOCPbuf[0]) Disconnect(client);
delete pOverlappedEx;
```

IOCP 서버 구현

- 체스에서 클라이언트 객체

```
class ChessClient{
        ClientInfo      m_ClientInfo;
        wchar_t         m_name[20];
        BYTE            m_x;
        BYTE            m_y;
};

class ClientInfo{
        int             m_id;
        SOCKET          m_socket;
        stOverlappedEx  m_RecvOverEx;
        unsinged char   m_packet_buf[MAX_PACKET_SIZE];
        int             m_prev_size;
};

ChessClient    g_clients[10];
```

IOCP 서버 구현

- 프로토콜 정의 TIP
  - #pragma pack(push, 1) 사용

```
#pragma pack(push, 1)
struct SCdisconnect {
          BYTE      size, type, id;
}; // 이 플레이어가 접속을 끊었으므로 화면에서 제거

struct SClogin {
          BYTE      size, type, id, x, y;
          wchar_t   name[20];
}; // id와 정보, 시작위치를 처음 접속한 클라이언트에 전송

struct SCputplayer {
          BYTE      size, type, id, x, y;
          wchar_t   name[20];
}; // 새로운 플레이어가 주위에 나타났음을 알려 줌

struct SCposition {
          BYTE      size, type, id, x, y;
}; // 이 플레이어의 위치가 이곳으로 변경됨

#pragma pack(pop)
```

IOCP 서버 구현 (2019-수목)

- Thread Pool 생성
  - 쓰레드 만들기
  - 쓰레드를 GQCS를 사용해 IOCP에 대기상태로 넘겨 주기
- IOCP 실습 시작