Socket I/Os in Windows

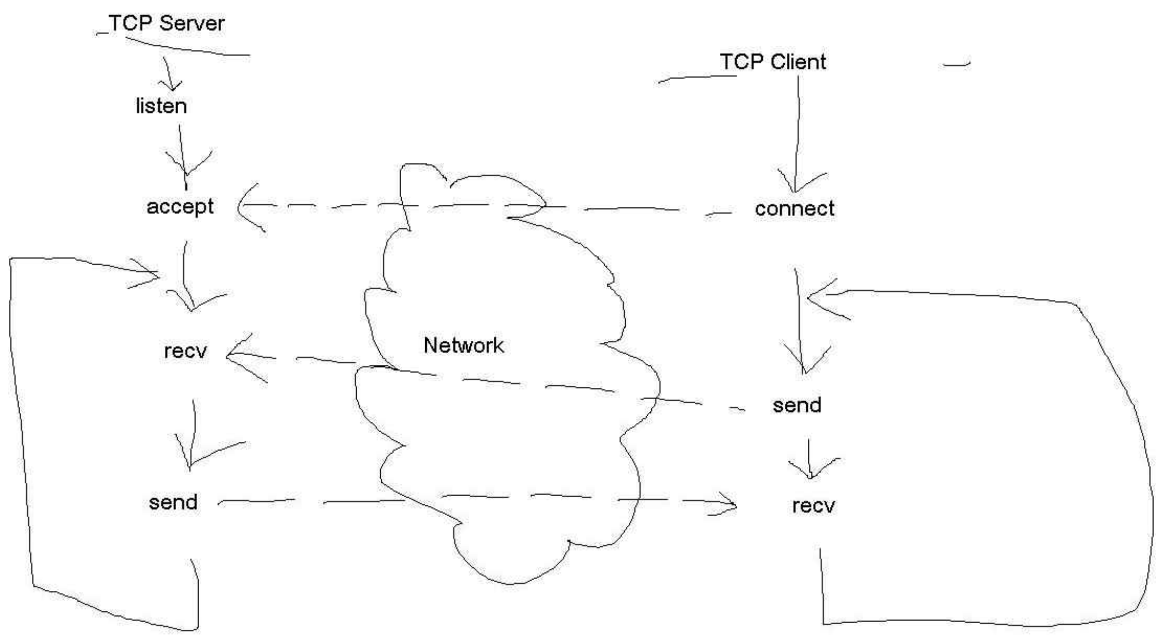
Dae-Ki Kang

Agenda

- TCP Server/Client
- Multi-Threads
- Synchronization
- Socket IO Model
- WSAAsyncSelect Model
- WSAEventSelect Model
- UDP Server/Client
- Overlapped Model
- Completion Port Model

TCP Server/Client

TCP Server/Client



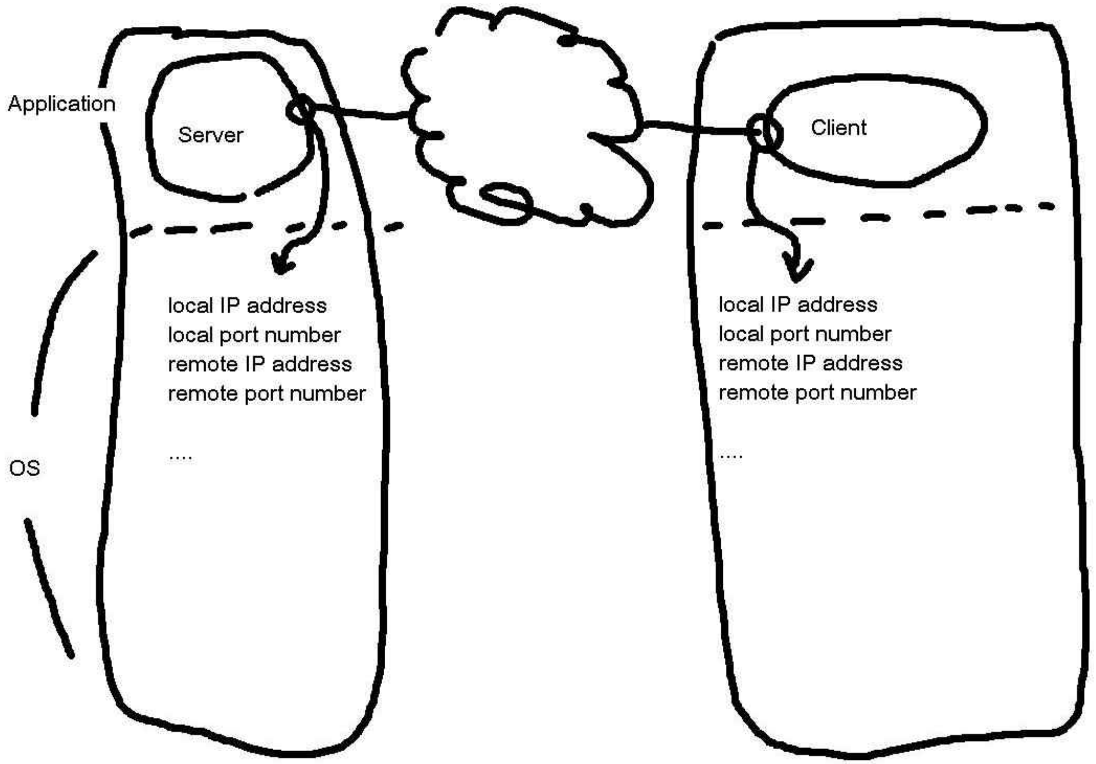
TCP Server/Client

- listen – server waits for the client
- connect – client connects to server and send data
- accept – server accepts client and recv data
- send – server sends data to client
- recv – client recv data from server

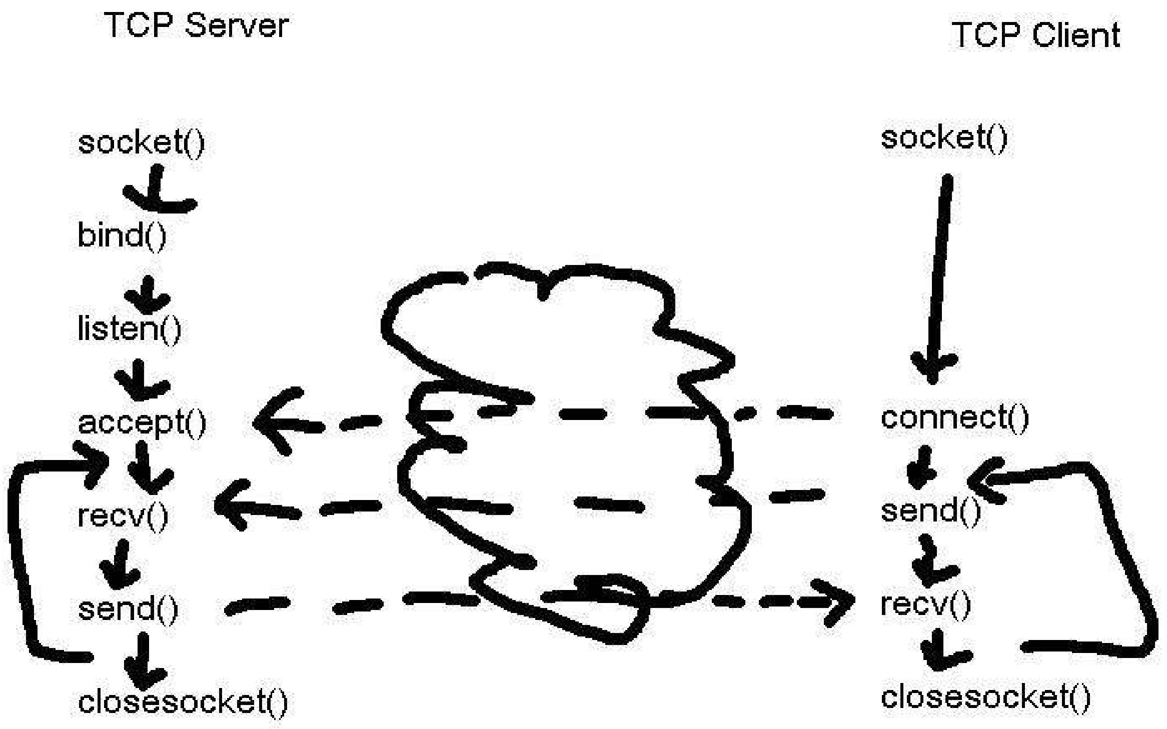
Necessary things for TCP/IP Socket

- Protocol
- Local IP and port address
- Remote IP and port address

Socket data structure



Functions



TCP Server
socket()
bind()
listen()
accept()
recv()
send()
closesocket()
TCP Client
socket()
connect()
send()
recv()
closesocket()

bind()

- TCP server function – determine server's local IP address and local port number

```
int bind (
  SOCKET s,
  const struct sockaddr* name,
  int namelen
);
```

- Success:0, Failure: SOCKET_ERROR

listen()

- TCP server function – set TCP port status to LISTENING to accept client

```
int listen (
 SOCKET s,
 int backlog
);
```

- Success:0, Failure: SOCKET_ERROR

accept()

- TCP server function – create and return a new socket to communicate with connected client with client's IP address and port number

```
SOCKET accept (
  SOCKET s,
  struct sockaddr* addr,
  int* addrlen
);
```

- Success: new socket, Failure: INVALID_SOCKET

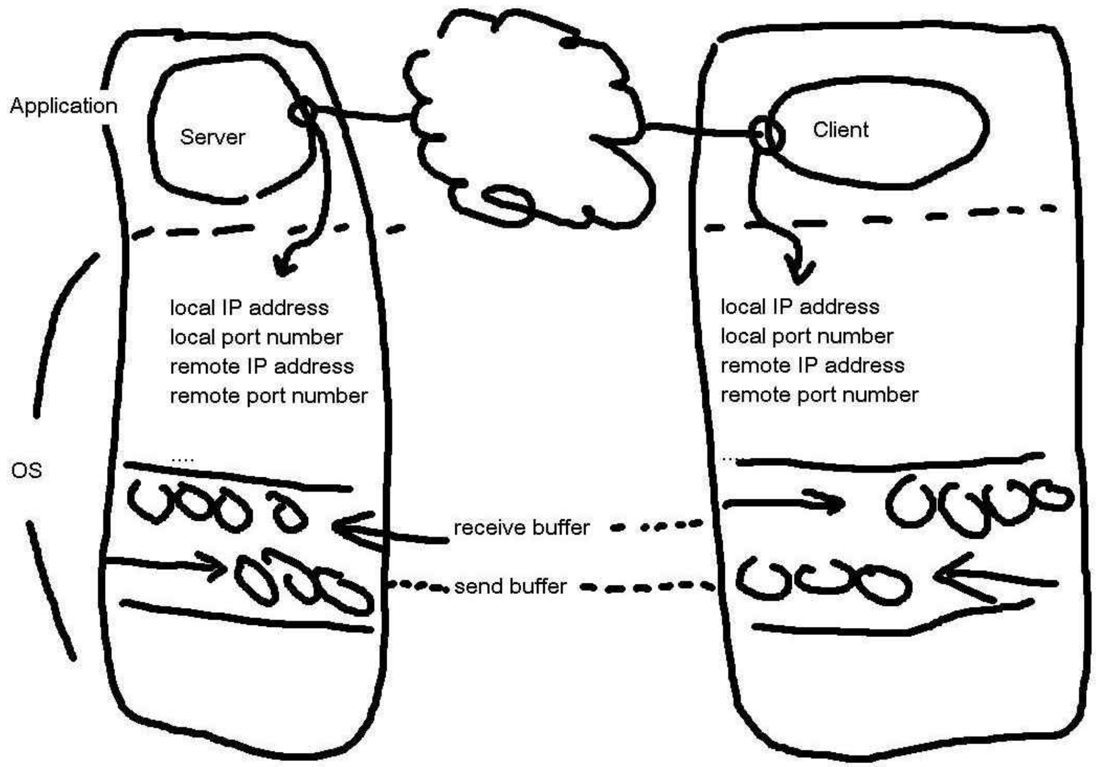
connect()

- TCP client function – connect to server and set TCP protocol connection

```
int connect (
  SOCKET s,
  const struct sockaddr * name,
  int namelen
);
```

- Success:0, Failure: SOCKET_ERROR

Data transmission



send()

- Data transmission function – copy application data to send buffer

```
int send(
  SOCKET s,
  const char * buf,
  int len,
  int flags
);
```

- Success: number of bytes, Failure: SOCKET_ERROR

recv()

- Data transmission function – copy receive buffer to application buffer

```
int recv(
  SOCKET s,
  const char * buf,
  int len,
  int flags
);
```

- Success: number of bytes or 0, Failure: SOCKET_ERROR

Considerations for message design

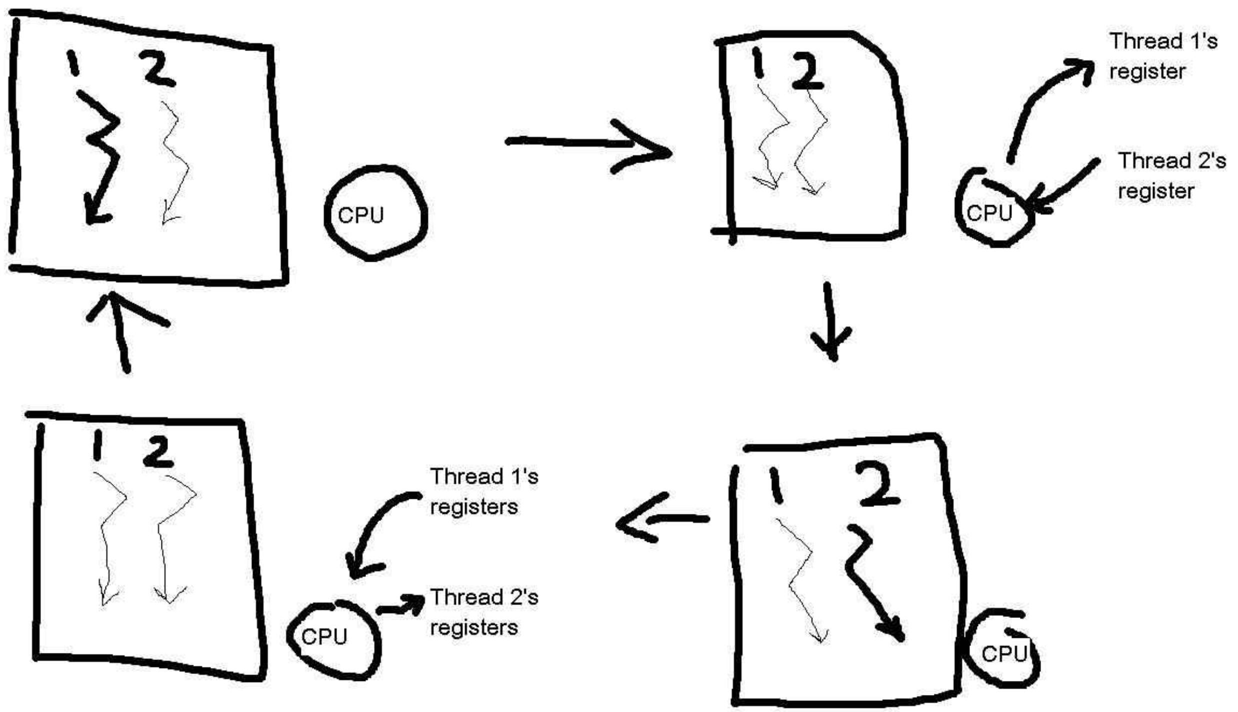
- Boundary
  - use data with fixed length
  - use special symbol for EOR (end of record)
  - use the length of data in advance and send data with variable length
- Byte order – network byte order is Big Endian!
- Member ordering
  - starting address of struct
  - #progma pack

Multi-Threads

Multi-Threads

- Process
- Thread – shares global memory inside a process
- Primary Thread
  - thread started in main() or WinMain()
  - created when process starts
- Context Switch – store and retrieval of thread's states

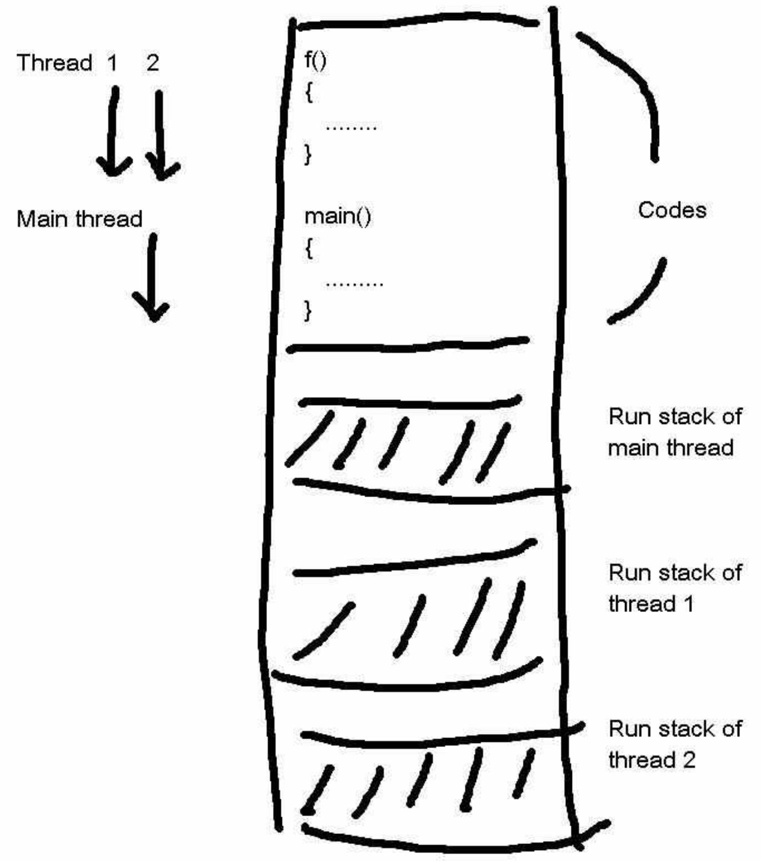
Context switching



What are needed for thread creation?

- Starting address of thread functions
- Size of stack that thread function uses

Consider 2 functions and 3 threads



CreateThread()

- create thread and returns its handle

```
HANDLE CreateThread(
 LPSECURITY_ATTRIBUTE lpThreadAttributes, //NULL
 SIZE_T dwStackSize, // 0
 LPTHREAD_START_ROUTINE lpStartAddress,
                                   // Thread functon
 DWORD dwCreationFlags,
                                   // 0 or CREATE_SUSPENDED
 LPDWORD lpThreadID // Thread ID
);
```

- Success: Thread handle, Failure: NULL

Defining thread function

```
DWORD WINAPI ThreadProc
 (LPVOID lpParameter)
{
  ……
}
```

Terminate threads

- *The thread function itself returns*
- The thread function run ExitThread() function
- Main function call TerminateThread()
- Main thread terminates → all children threads terminate

ExitThread()

```
void ExitThread(
  DWORD dwExitCode
);
```

TerminateThread()

```
BOOL TerminateThread (
  HANDLE hThread,
                    // HANDLE thread to terminate
  DWORD dwExitCode // exit code
);
```

Wait for thread

- WaitForSingleObject();
- WaitForMultipleObjects();

WaitForSingleObject()

```
DWORD WaitForSingleObject(
  HANDLE hHandle,
  DWORD dwMilliseconds
);
```

Success: WAIT_OBJECT_0 or WAIT_TIMEOUT
Failure: WAIT_FAILED

```
HANDLE hThread = CreateThread(...);
WaitForSingleObject(hThread, INFINITE);
```

WaitForMultipleObjects()

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

- Success: WAIT_OBJECT_0 ~ WAIT_OBJECT_0+nCount-1 or WAIT_TIMEOUT
- Failure: WAIT_FAILED

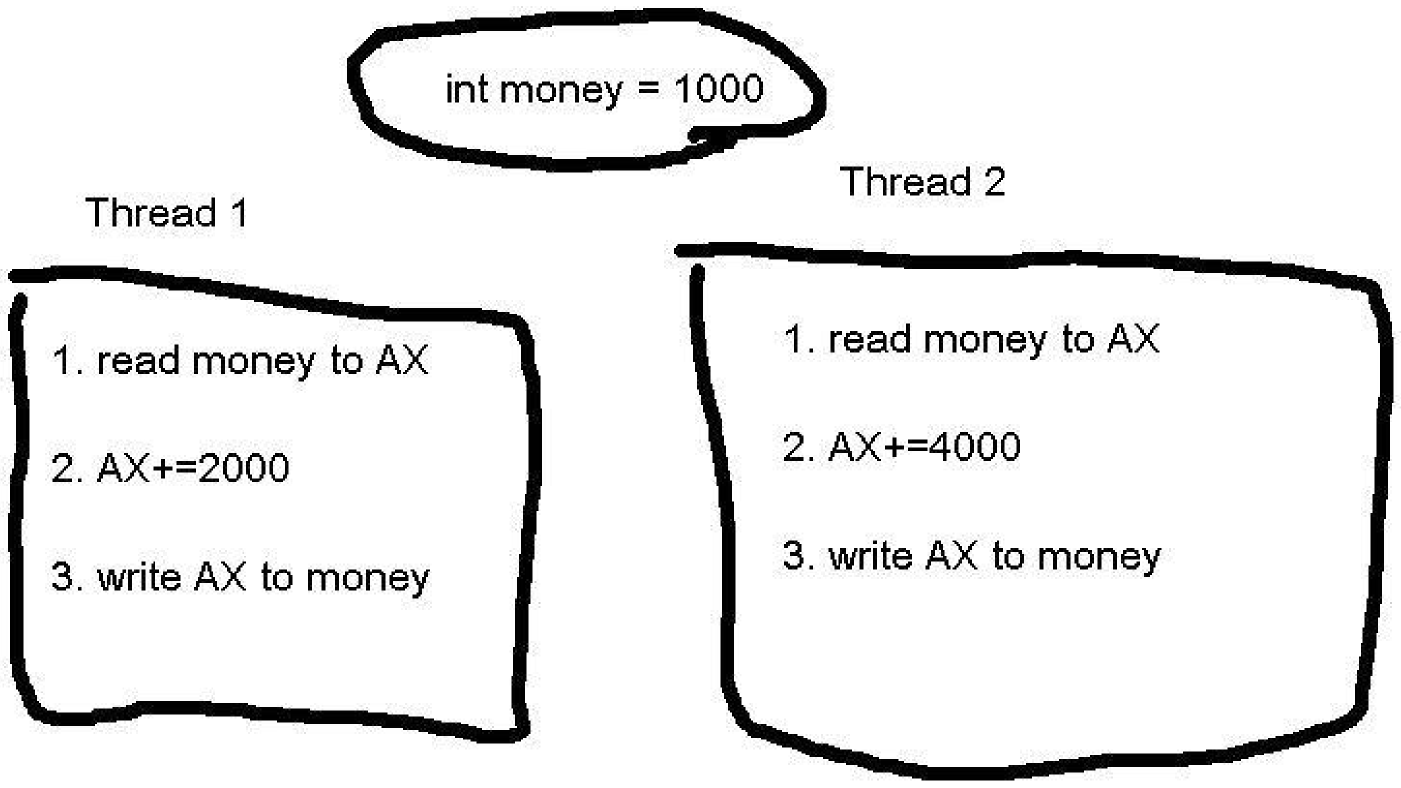
Synchronization

Synchronization

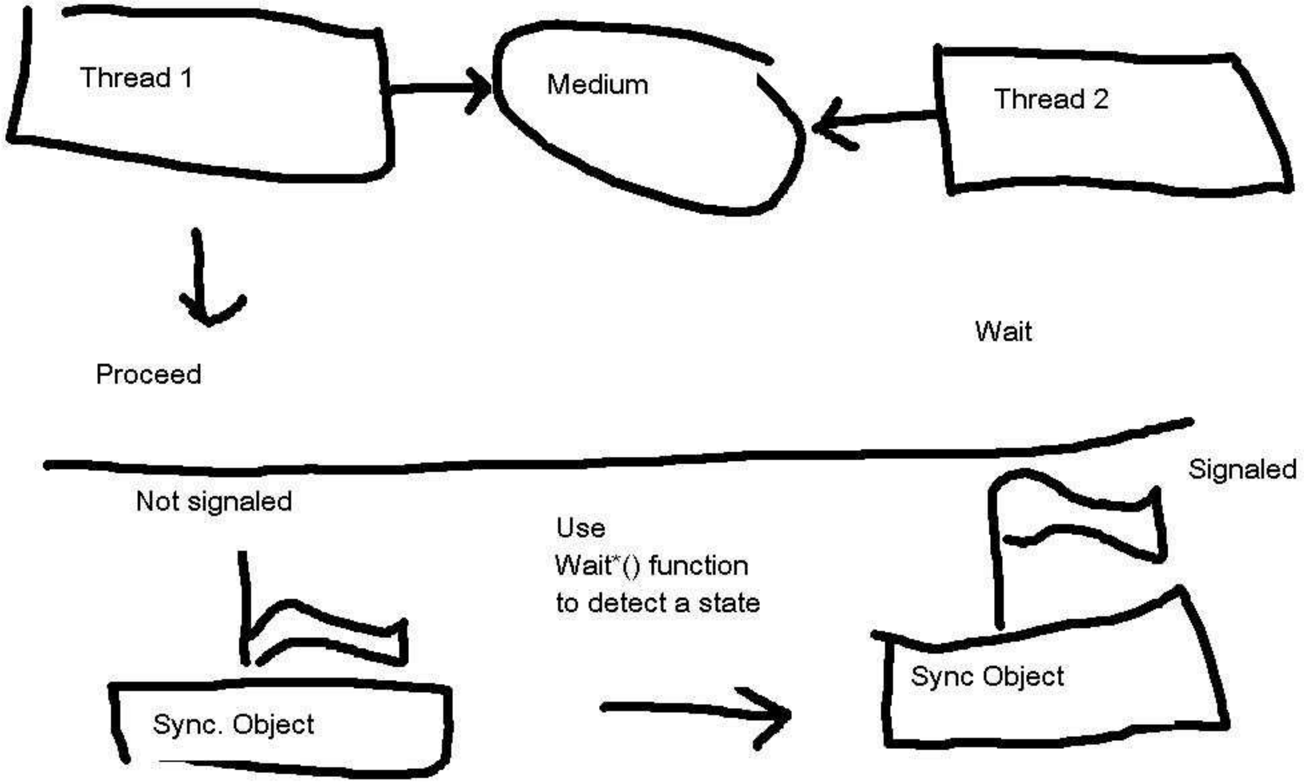
- In multithreaded program, two threads try to access shared memory

- Critical section – one thread for one shared resource (in a process)
- Mutex – one thread for one shared resource (in processes)
- Event – notify an event to other threads
- Semaphore – access control
- Waitable timer – wakes waiting thread after some time

Synchronization



Thread synchronization



Critical section

- Critical section object is in user memory area, so CS is used for threads in a process
- Faster and more efficient than others

```
#include <windows.h>
CRITICAL_SECTION cs;
// Thread 1
DWORD WINAPI Thread1(LPVOID arg)
{
 EnterCriticalSection(&cs);
 // use shared memory
 LeaveCriticalSection(&cs);
}
```

```
// Thread 2
DWORD WINAPI Thread2(LPVOID arg)
{
 EnterCriticalSection(&cs);
 // use shared memory
 LeaveCriticalSection(&cs);
}
int main()
{
 InitializeCriticalSection(&cs);
 DeleteCriticalSection(&cs);
}
```

Event object

- Used to notify events to other threads
  - auto reset event
  - manual reset event
- Create event object (not-signaled)
- One thread proceed, the other wait
- After completion, the thread change the event to be signaled
- The other thread proceed

```
BOOL SetEvent(HANDLE hEvent);  // to signaled
BOOL ResetEvent(HANDLE hEvent);  // to un-
   signaled

HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTE lpEventAttribute;
  BOOL bManualReset;
  BOOL bInitialState;
  LPSTR lpName;
);
```

Success: Event handle, Fail: NULL

Socket IO Model

Socket IO Model

- Socket mode – Blocking v. Non-blocking
- Blocking
  - accept() returns when client connects
  - send(), sendto() returns when all data transfered
  - recv(), recvfrom() returns when data arrived
- Non-blocking socket returns even if the conditions are not met

```
// create a blocking socket first
SOCKET listen_sock = socket(AF_INET,
  SOCK_STREAM, 0);
if (INVALID_SOCKET==listen_sock)
  err_quit("socket()");

// and change it to non-blocking socket
u_long on=1;
retval=ioctlsocket(listen_sock, FIONBIO, &on);
if (SOCKET_ERROR==retval)
  err_quit("ioctlsocket");
```

Socket IO Model

- When socket functions are used for non-blocking socket that is not ready, the function return FAILURE stuff
- To check for the error code, call WSAGetLastError()
- Most error code is WSAEWOULDBLOCK, which means the conditions are not met
- Then the program have to call the socket function again after sometime

Non-blocking socket

- Pros
  - the program can continue other work
  - without multithreads, the program can process multiple socket IOs
- Cons
  - has to check for WSAEWOULDBLOCK for all socket functions, which makes the program complicated
  - more CPU usage

Server model

- Repetitive server processes one client at a time and then process the next
  - Less resource
  - Longer wait time
- Concurrent server processes multiple client in parallel
  - Shorter wait time ?a client which takes long time does not affect others
  - Use multi-processor, multi-process, or multi-threads, so more resource usage

Ideal server and ideal socket

- Ideal server
  - all client can connect
  - server always responds to the client immediately and send data in high speed
  - minimize the system resource usage
- Ideal socket IO model
  - minimal blocking in socket function call
  - concurrent processing of IO task and other tasks
  - minimum number of threads
  - minimize user-mode & kernel mode switch
  - minimize internal temporary data transfer

Socket I/O model in Windows

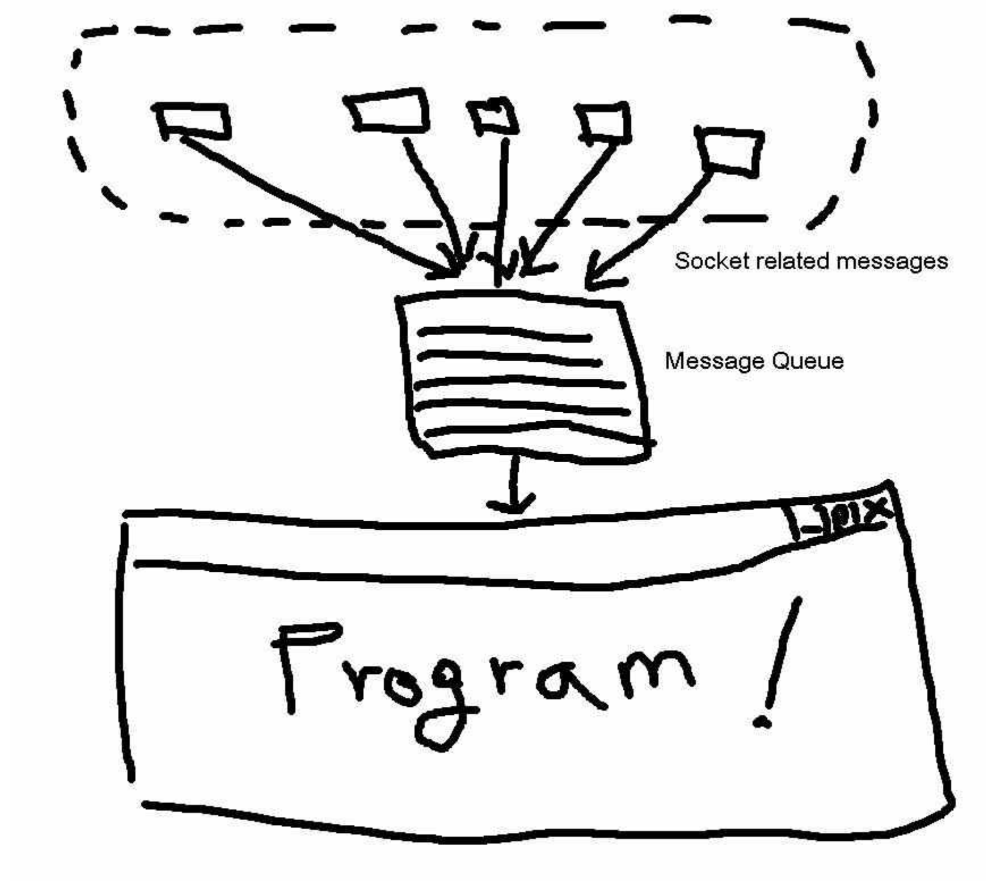
Socket IO model	Windows CE	Windows (client version)	Windows (server version)
Select	CE 1.0+	Window 95+	Windows NT+
WSAAsyncSelect	N/A	Window 95+	Windows NT+
WSAEventSelect	CE.NET 4.0+	Window 95+	Windows NT 3.51+
Overlapped	CE.NET 4.0+	Window 95+	Windows NT 3.51+
Completion Port	N/A	Windows NT 3.5+(except Win 95/98/Me)	

WSAAsyncSelect Model

WSAAsyncSelect Model

- WSAAsyncSelect() is most crucial
- Process network events related with sockets in terms of Window Message – can process multiple sockets without using multi-threads

How it works?



Socket I/O procedure in WSAAsyncSelect model

1. Register a window message and a network event using WSAAsyncSelect() – For example, register that when the system can send or receive data using the socket, invoke a certain window message
2. When the network event registered occurs, a window message is invoked, and window procedure is called
3. Window procedure process appropriate socket functions based on the message received

WSAAsyncSelect()

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

- Success:0, Failure: SOCKET_ERROR

```
#define WM_SOCKET(WM_USER+1)
                              // user defined window message
…
WSAAsyncSelect(s,hWnd,WM_SOCKET,FD_READ|FD_WRIT
   E)
```

Network event constants

Network Event	Meaning (produce Windows Message when…)
FD_ACCEPT	When a client connects
FD_READ	When data can be read
FD_WRITE	When data can be written
FD_CLOSE	When the client disconnects
FD_CONNECT	When the client finish connection
FD_OOB	When OOB data is arrived OOB data = Out of Band data (or “urgent data” in TCP)

Window procedure

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT
   msg, WPARAM wParam, LPARAM lParam)
{
 …
}
```

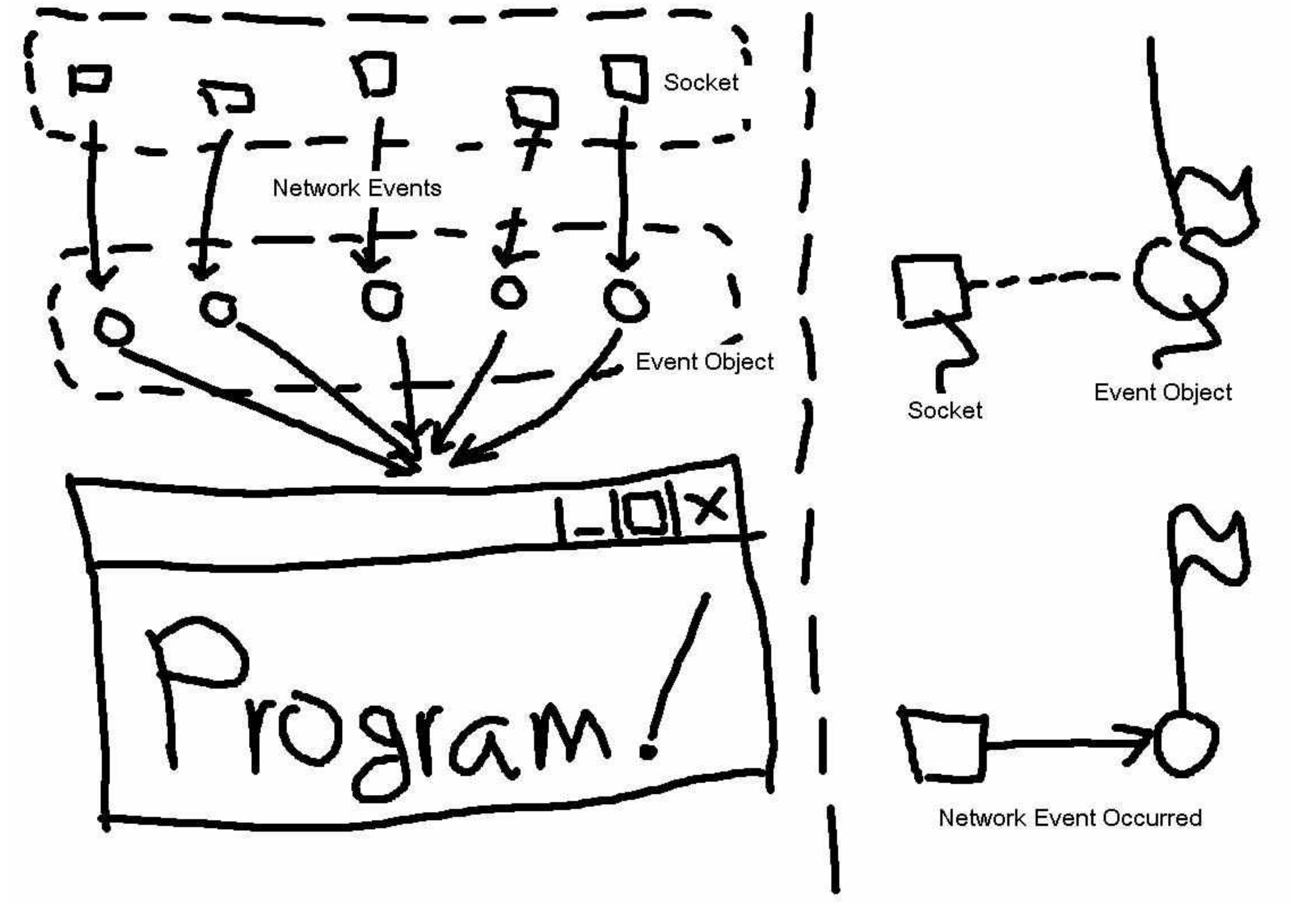
- hWnd - window that generates the message
- msg – user-defined message registered when WSAAsyncSelect() was invoked
- wParam – socket that causes network event
- lParam – lower 16 bits: network event, higher 16 bits: error code

WSAEventSelect Model

WSAEventSelect Model

- WSAEventSelect() is an important function
- Detecting network event using event object
  - create an event object for each socket
  - observe the event objects to process multiple sockets without using multu-threads

How it works


Socket
Network Events
Event Object
Program!
Socket
Event Object
Network Event Occurred

Necessary features

- Event object creation/desctruction:
  - WSACreateEvent(), WSACloseEvent()
- Pairing socket and event object
  - WSAEventSelect()
- Noticing signal state of event object
  - WSAWaitForMultipleEvents()
- Finding out network event detail
  - WSAEnumNetworkEvents()

Socket IO procedure in WSAEventSelect model

1. Create an event object using WSACreateEvent() whenever a socket is created
2. Pair socket and event object using WSAEventSelect() and register network event to process – for example, register signal state of event object when the system can send/receive data through socket
3. Call WSAWaitForMultipleEvents() to wait for the event object to be signaled. When the registered network event is occurred, the event object related with the socket become signal state
4. Find out the occurred network event using WSAEnumNetworkEvents() and call appropriate socket functions to process the event

Creation/destruction of event object

WSAEVENT WSACreateEvent();

- Success: event object handle, Failure: WSA_INVALID_EVENT

BOOL WSACloseEvent(WSAEVENT hEvent);

- Success: TRUE, Failure: FALSE

Pairing socket and event

WSAEVENT WSACreateEvent();

- Success: Event Object Handle, Failure: WSA_INVALID_EVENT

BOOL WSACloseEvent(WSAEVENT hEvent);

- Success: TRUE, Failure: FALSE

Noticing signal state of event object

```
DWORD WSAWaitForMultipleEvents(
    DWORD cEvents,
    const WSAEVENT* lphEvents,
    BOOL fWaitAll,
    DWORD dwTimeout,
    BOOL fAlertable
);
```

- Success: WSA_WAIT_EVENT_0 ~ WSA_WAIT_EVENT_0+cEvents-1 or WSA_WAIT_TIMEOUT
- Failure: WSA_WAIT_FAILED

Finding out network event detail

```
int WSAEnumNetworkEvent(
    SOCKET s,
    WSAEVENT hEventObject,
    LPWSAMETWORKEVENTS lpNetworkEvents
);
```

- Success:0, Failure: SOCKET_ERROR

UDP Server/Client

TCP and UDP: commonalities

- Using port # and address
- Check data error

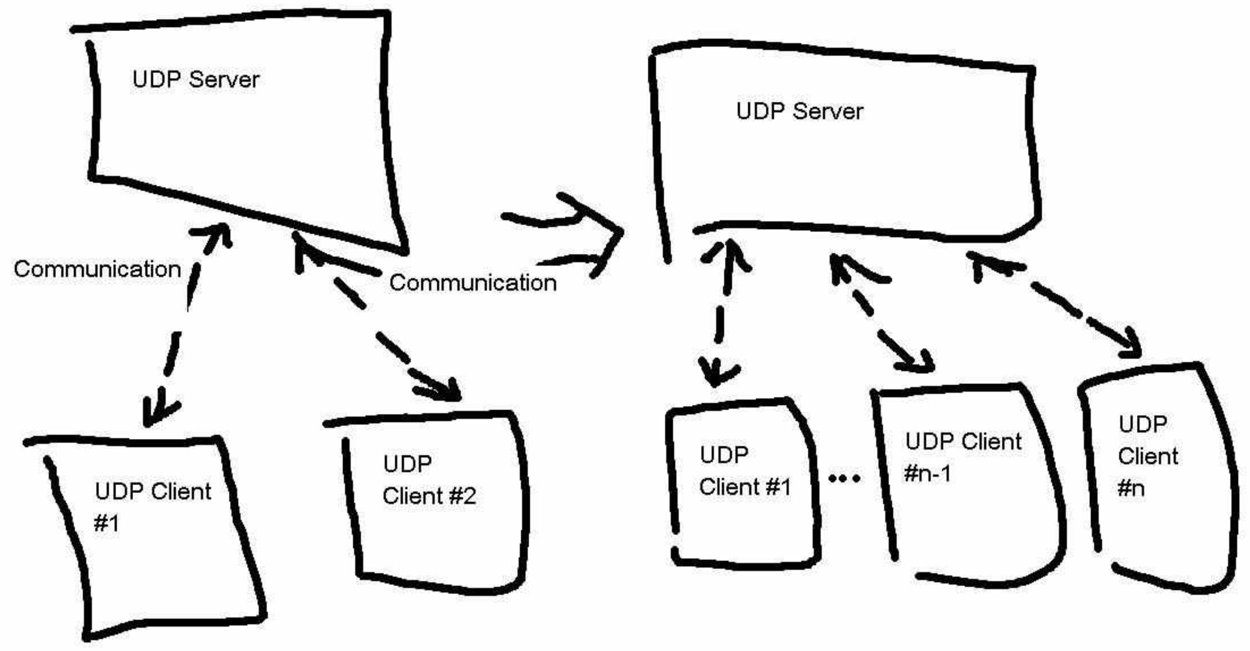
TCP and UDP: differences

	TCP	UDP
1	Connection-oriented protocol – data communication after connection	Connectionless protocol – communication without connection
2	No data boundary – byte stream service	Data boundary – datagram
3	Reliable data communication - resend data	Not reliable data communication – do not resend data
4	1-to-1 communication (unicast)	1-to-1 communication (unicast) 1-to many (broadcast) many-to-many (multicast)

UDP’s characteristics

- Usually, connect() function not used
- User applications don’t have to check for data boundaries
- If needed, user applications have to guarantee reliable communication (since, reliable communication was not performed in protocol-level)
- Easy to implement many-to-many communication

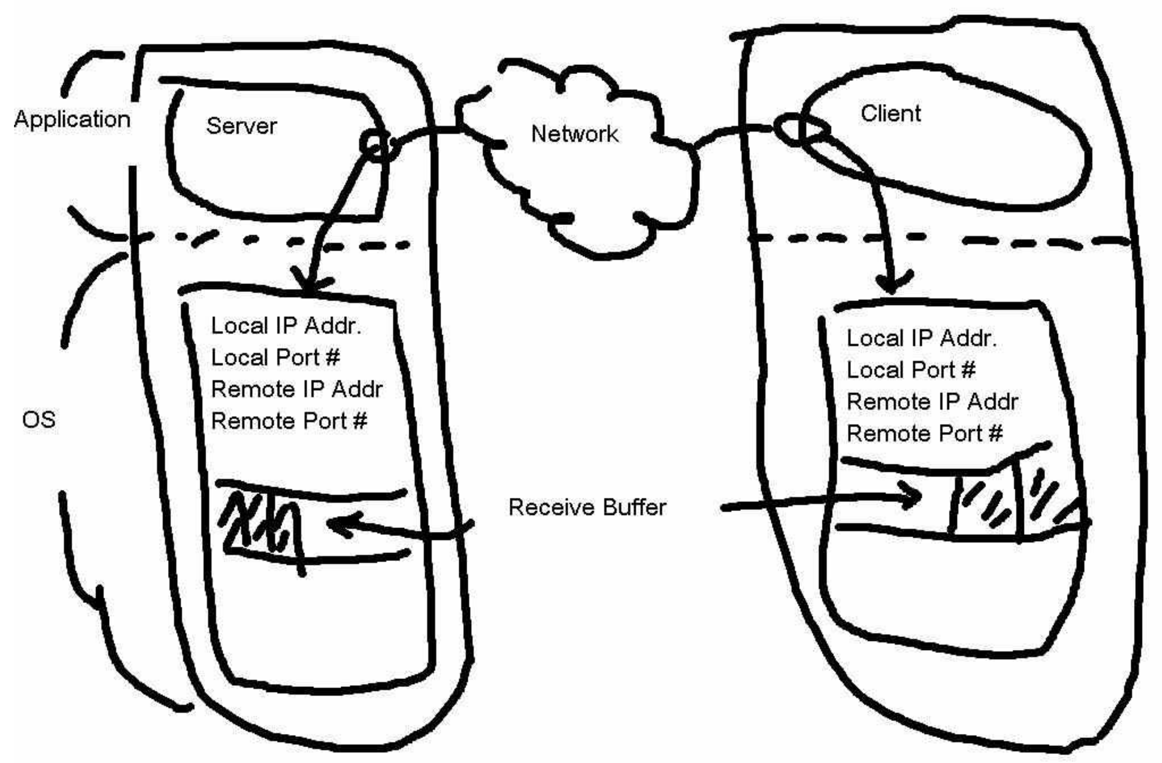
How UDP server/client work?



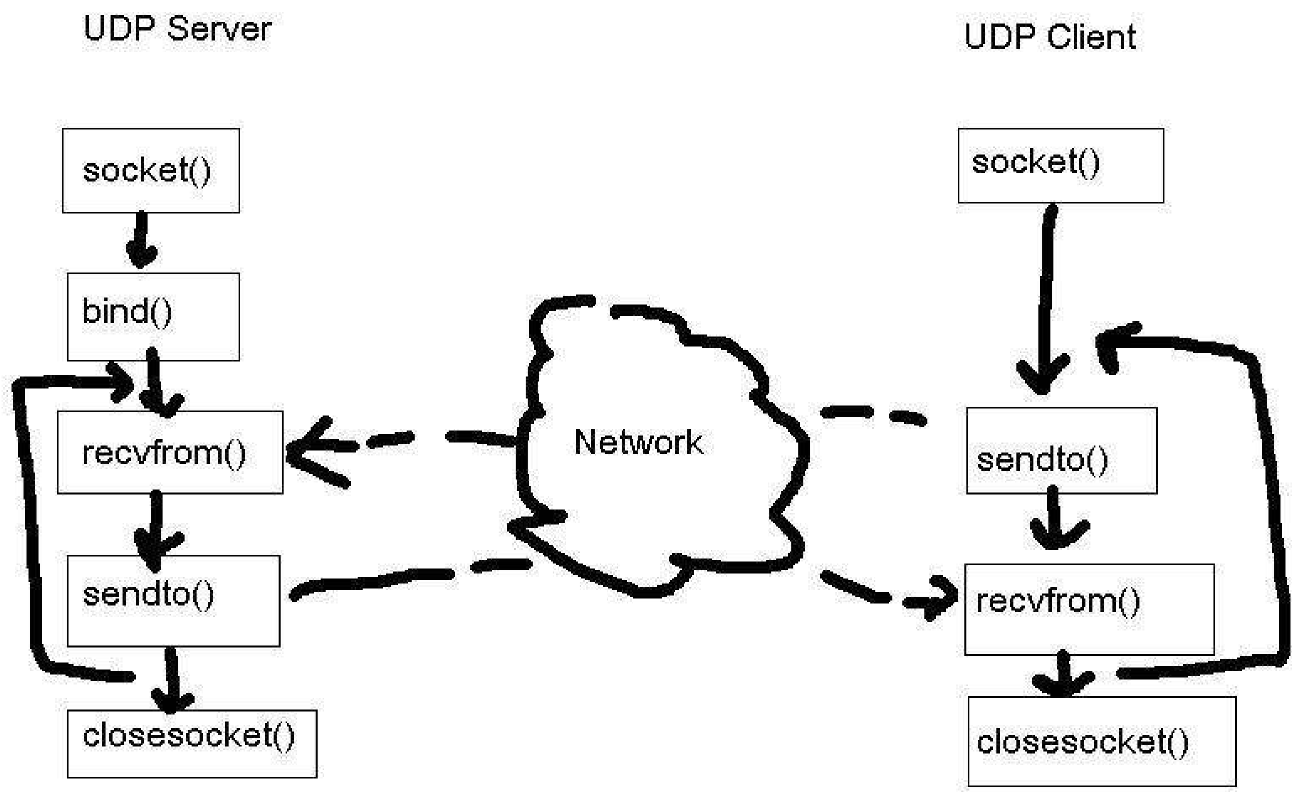
UDP/IP socket communication involves

- Protocol for socket creation
- Local IP and address
- Remote IP and address

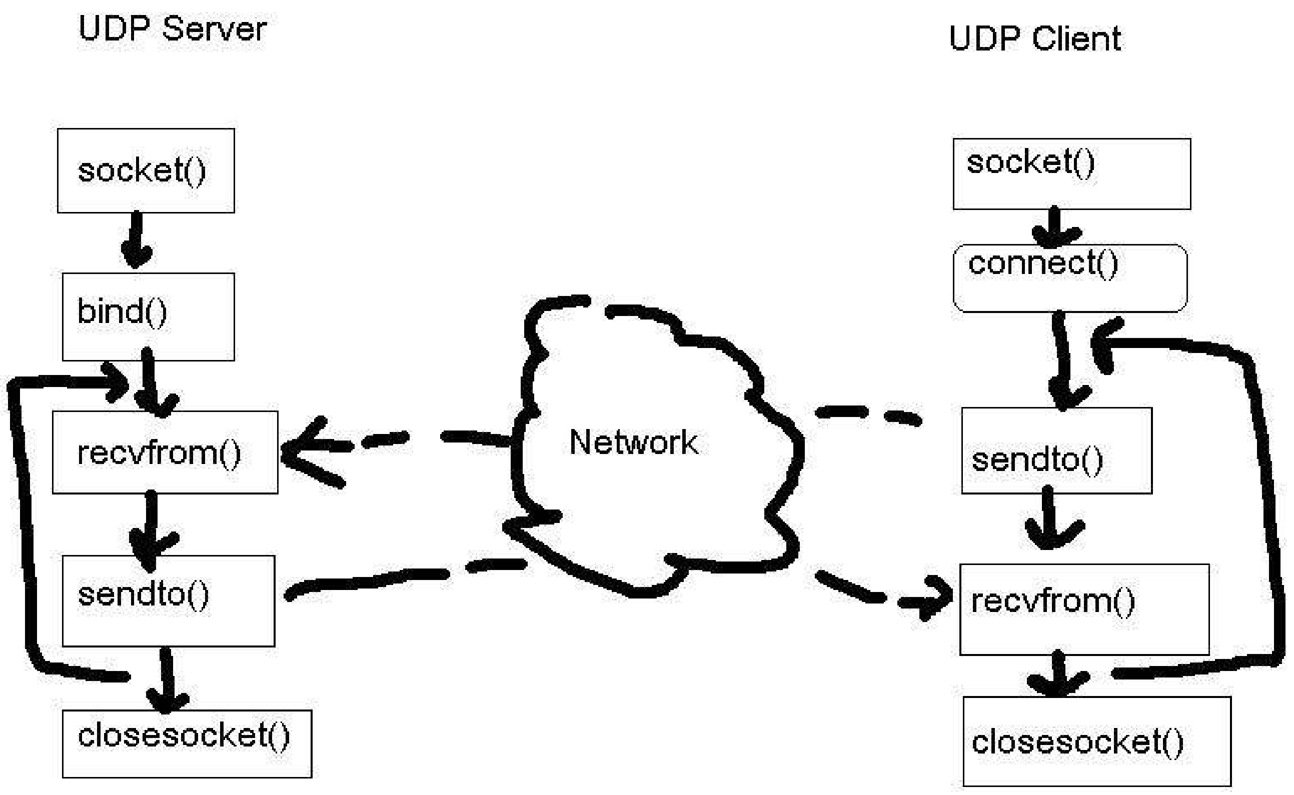
Socket data structure



UDP server/client architecture



UDP server/client architecture



sendto()

- Data communication function
- Use lower protocol (i.e. UDP/IP) to send application data
- If local IP and local port are not set when sendto() is called, the system decide them automatically

```
int sendto (
    SOCKET s,
    const char* buf,
    int len,
    int flags,
    const struct sockaddr* to,
    int tolen
);
```

- Success: # of sent in byte, Failure: SOCKET_ERROR

sendto() usage

```
// initialize socket struct with receiver address
SOCKADDR_IN serveraddr;
...
// declare buffer for data to send
char buf[BUFSIZE];

// store data to buffer

// send data
retval = sendto(sock, buf, strlen(buf), 0, (SOCKADDR
   *)&serveraddr, sizeof(serveraddr));
if (SOCKET_ERROR == retval) error_processing();
printf("%d bytes sent\n", retval);
```

recvfrom()

- data communication function
- copy received data to application buffer

```
int recvfrom(
   SOCKET s,
   char* buf;
   int len,
   int flags,
   struct sockaddr* from,
   int* fromlen
);
```

- Success: # of received in byte, Failure: SOCKET_ERROR

recvfrom() usage

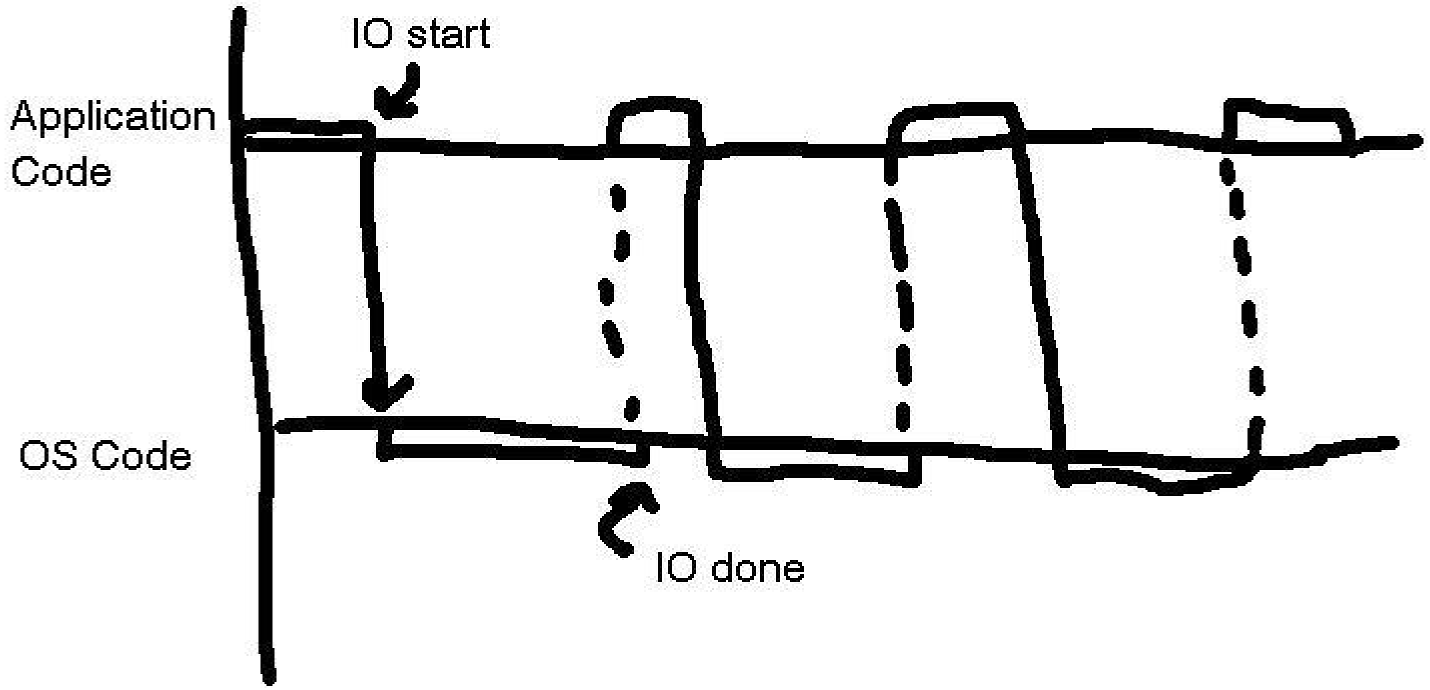
```
// declare a variable for sender address
SOCKADDR_IN peeraddr;
int addrlen;
// declare buffer to store received data
char buf[BUFSIZE];
// receive data
addrlen=sizeof(peeraddr);
retval=recvfrom(sock, buf, BUFSIZE, 0,
   (SOCKADDR*)&peeraddr, &addrlen);
if (SOCKET_ERROR == ret_val) error_processing();
printf("%d bytes received\n",retval);
```

Overlapped Model

Synchronous IO

- After IO function call, applications wait until IO is done
- IO functions will return after IO task, and applications process the results or proceed to the next task

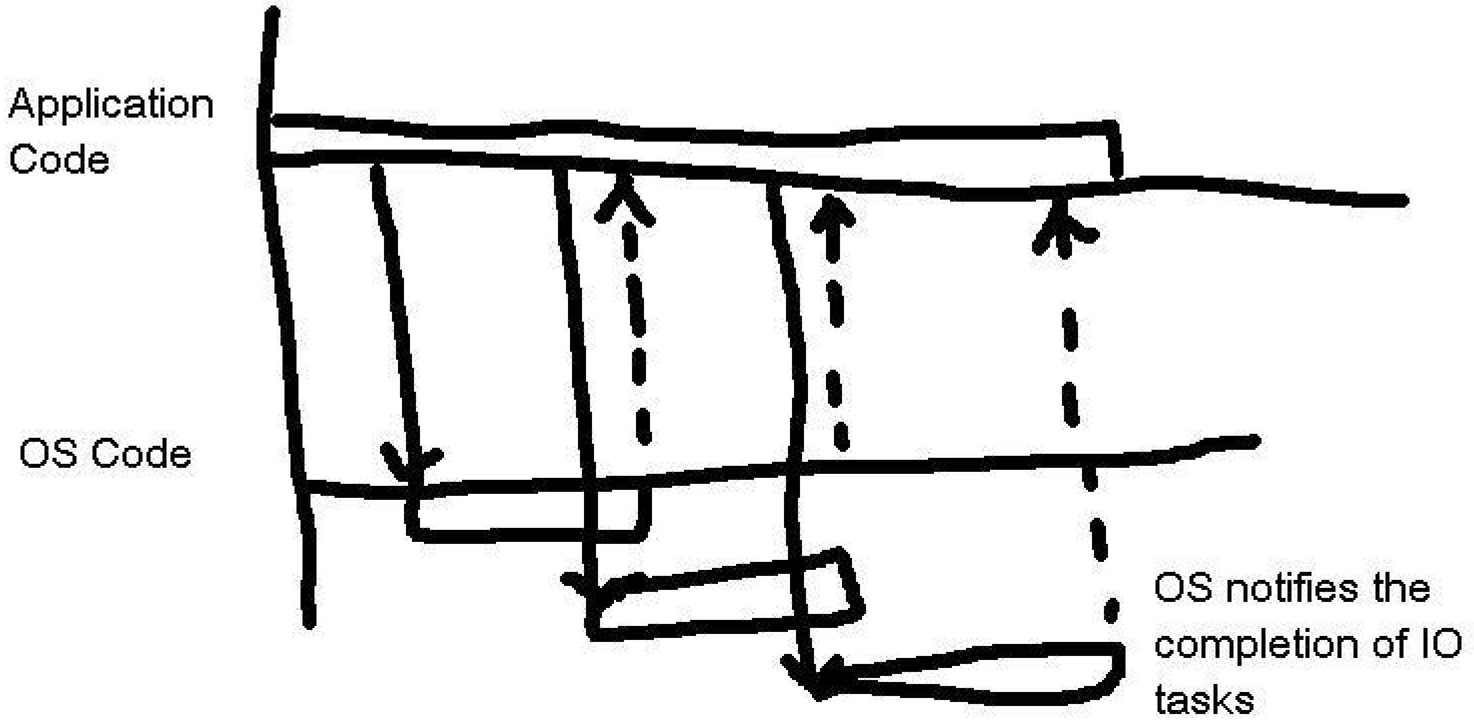
Synchronous IO



Asynchronous IO

- After IO function call, applications do their work
- When IO tasks are done, OS will notify it to applications, and the applications will stop their work and process the IO results

Asynchronous IO



Socket IO Models

- Synchronous IO + Asynchronous Notification
  - Select Model
  - WSAAsyncSelect Model
  - WSAEventSelect Model
- Asynchronous IO + Asynchronous Notification
  - Overlapped Model (I)
  - Overlapped Model (II)
  - Completion Port Model

Overlapped Model's Procedure

1. Create a socket that supports asynchronous IO
2. Call socket functions that supports asynchronous IO
3. OS asynchronously notifies the completion of socket IO and the application processes the results

Overlapped Models with asynchronous notification

Model	Description
OverlappedModel (I)	After the completion of socket IO, OS will change the status of the event object. Application observes the event object to detect the completion of IO tasks.
OverlappedModel (II)	After the socket IO completion, OS calls the function registered by the application. Generally, we call such functions ‘callback functions’. In overlapped model, we call them completion routines.

Input Function

```
int WSASend
(
   SOCKET s,
   LPWSABUF lpBuffers,
   DWORD dwBufferCount,
   LPDWORD lpNumberOfBytesSent,
   DWORD dwFlags,
   LPWSAOVERLAPPED lpOverlapped,
   LPWSAOVERLAPPED_COMPLETION_ROUTINE
      lpCompletionRoutine
);
```

- Success: 0, Fail: SOCKET_ERROR

Output Function

```
int WSARecv
(
   SOCKET s,
   LPWSABUF lpBuffers,
   DWORD
   LPDWORD lpNumberOfBytesReceived,
   LPDWORD lpFlags,
   LPWSAOVERLAPPED lpOverlapped,
   LPWSAOVERLAPPED_COMPLETION_ROUTINE
      lpCompleetionRoutine
);
```

- Success:0, Fail: SOCKET_ERROR

Data Structure

```
typedef struct _WSABUF
{
   u_long len; // length in byte
   char FAR* buf; // buffer’s starting address
} WSABUF, *LPWSABUF;
typedef struct _WSAOVERLAPPED
{
   DWORD Internal;
   DWORD InternalHigh;
   DWORD Offset;
   DWORD OffsetHigh;
   WSAEVENT hEvent;
} WSAOVERLAPPED, *LPWSAOVERLAPPED;
```

Overlapped Model (I)’s Socket IO Procedure

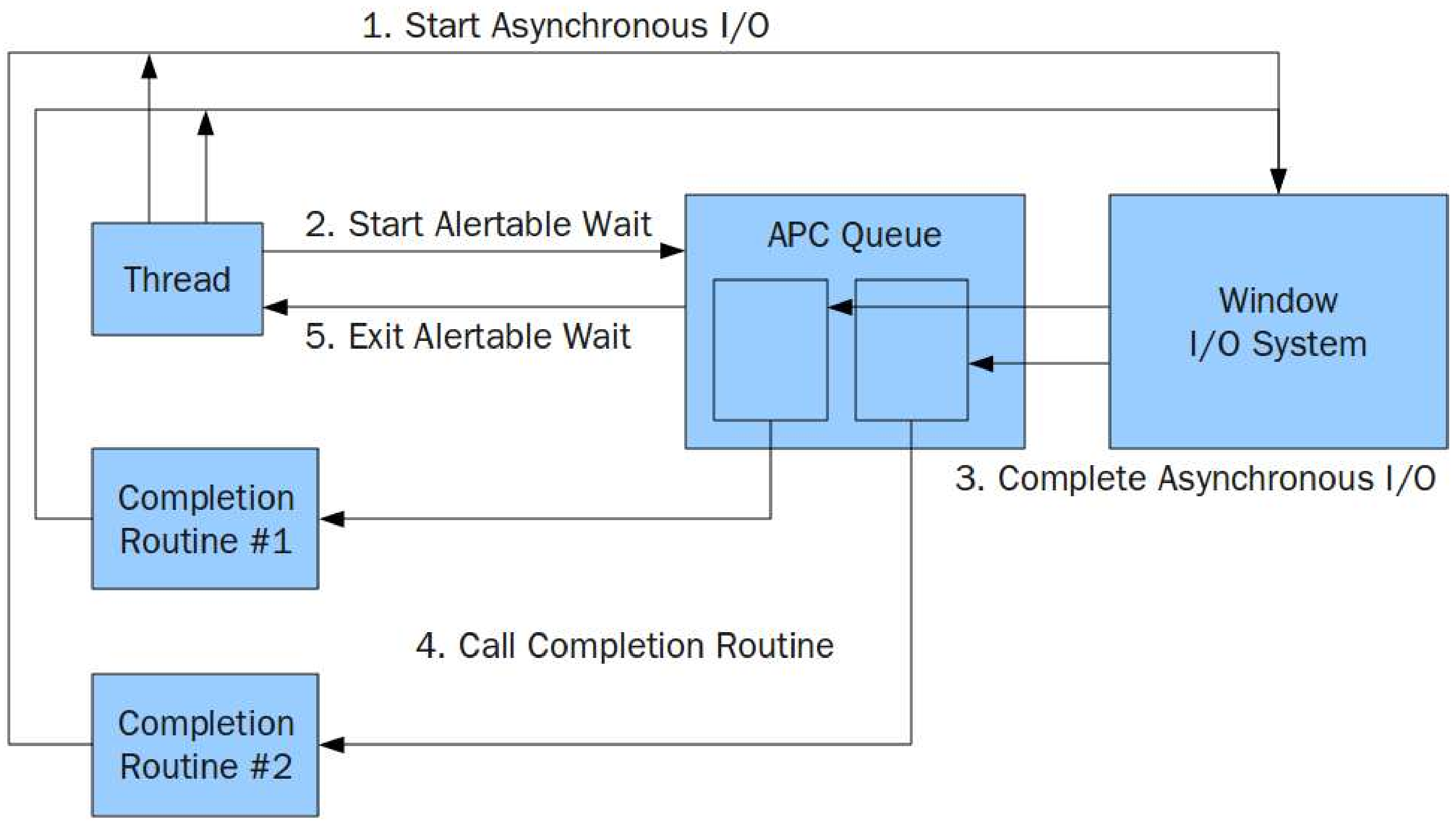
1. Create a socket of asynchronous IO. Then, call WSACreateEvent() function to create the corresponding event object
2. Call socket functions with asynchronous IO. Put event object handle value in hEvent variable of WSAOVERLAPPED struct. If the IO task is not completed soon, the socket function will return error with code WSA_IO_PENDING. After IO task completion, the OS will make the event object signaled
3. Call WSAWaitForMultipleEvents() to wait for the object signaled
4. After the asynchronous IO task completed, WSAWaitForMultipleEvents() will be returned, the main thread calls WSAGetOverlappedResult() to check asynchronous IO results and to process data
5. When a new socket is created repeat 1~4, otherwise repeat 2~4

WSAGetOverlappedResult()

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

- Success: TRUE, Fail: FALSE

Overlapped Model (II)



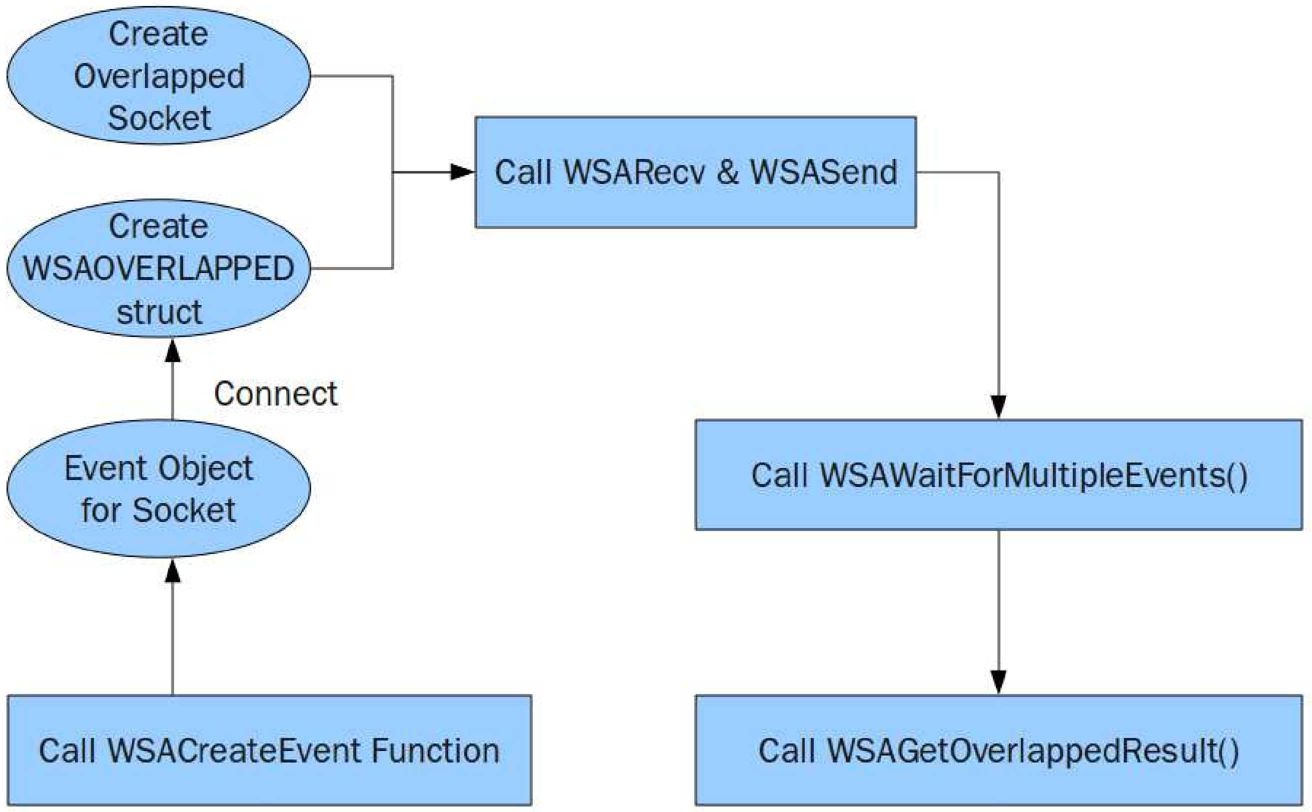
Overlapped Model (II)’s Socket IO Procedure

1. Create a socket of asynchronous IO.
2. Call socket functions with asynchronous IO with the start address of completion routine as a parameter. If the asynchronous IO is not completed soon, the socket function returns error with error code WSA_IO_PENDING.
3. Set the thread that invoked asynchronous IO function as an alertable wait state with one of the functions, WaitForSingleObjectEx(), WaitForMultipleObjectsEx(), SleepEx(), and WSAWaitForMultipleEvents().
4. After the asynchronous IO task completed, the OS will call the completion routine to check asynchronous IO results and to process data.
5. After the completion routine, the thread exit from the alertable wait state.
6. When a new socket is created repeat 1~5, otherwise repeat 2~5

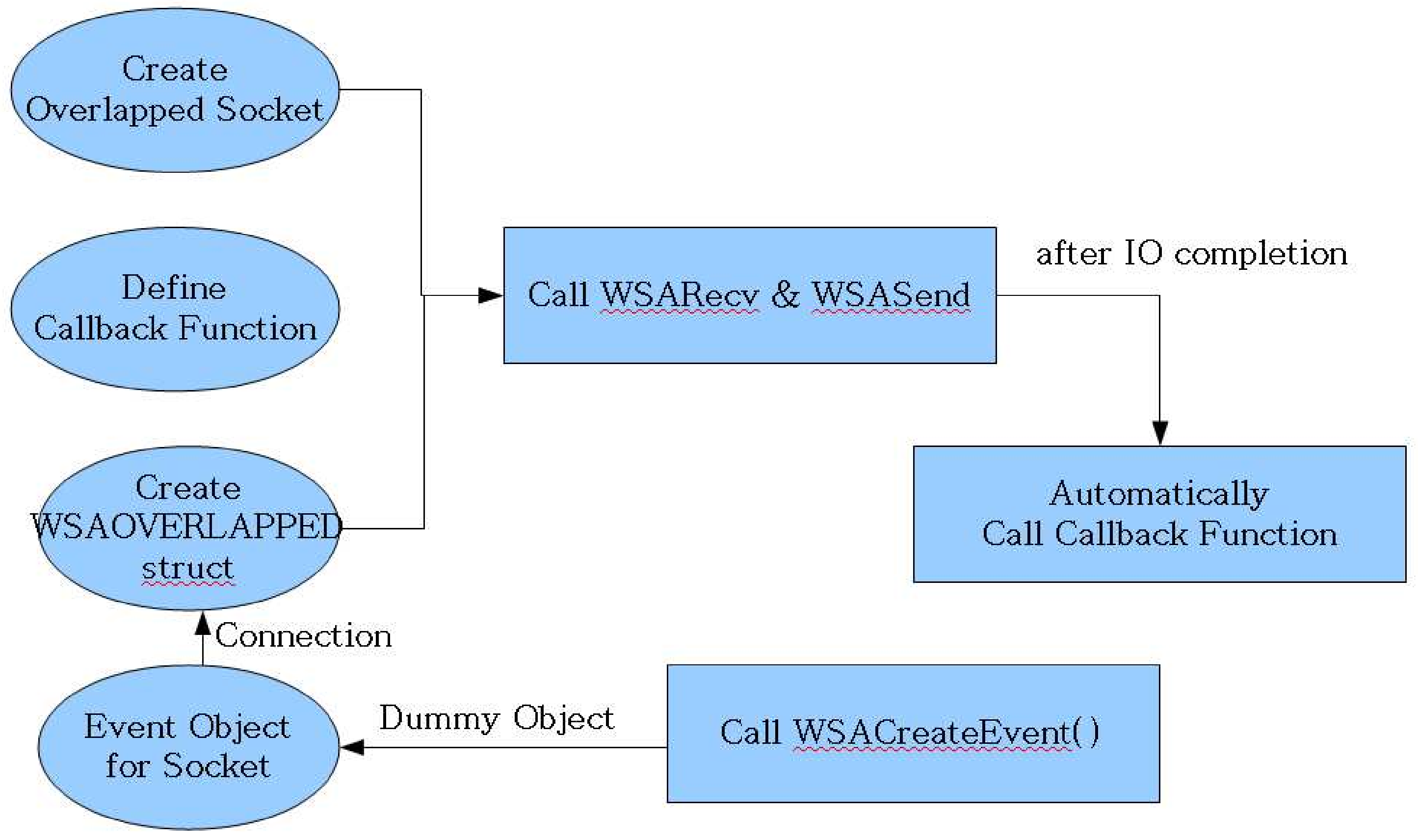
CompletionRoutine

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

Event kernel object based Overlapped IO


Create Overlapped Socket
Create WSAOVERLAPPED struct
Connect
Event Object for Socket
Call WSACreateEvent Function
Call WSARecv & WSASend
Call WSAWaitForMultipleEvents()
Call WSAGetOverlappedResult()

CompletionRoutine based Overlapped IO

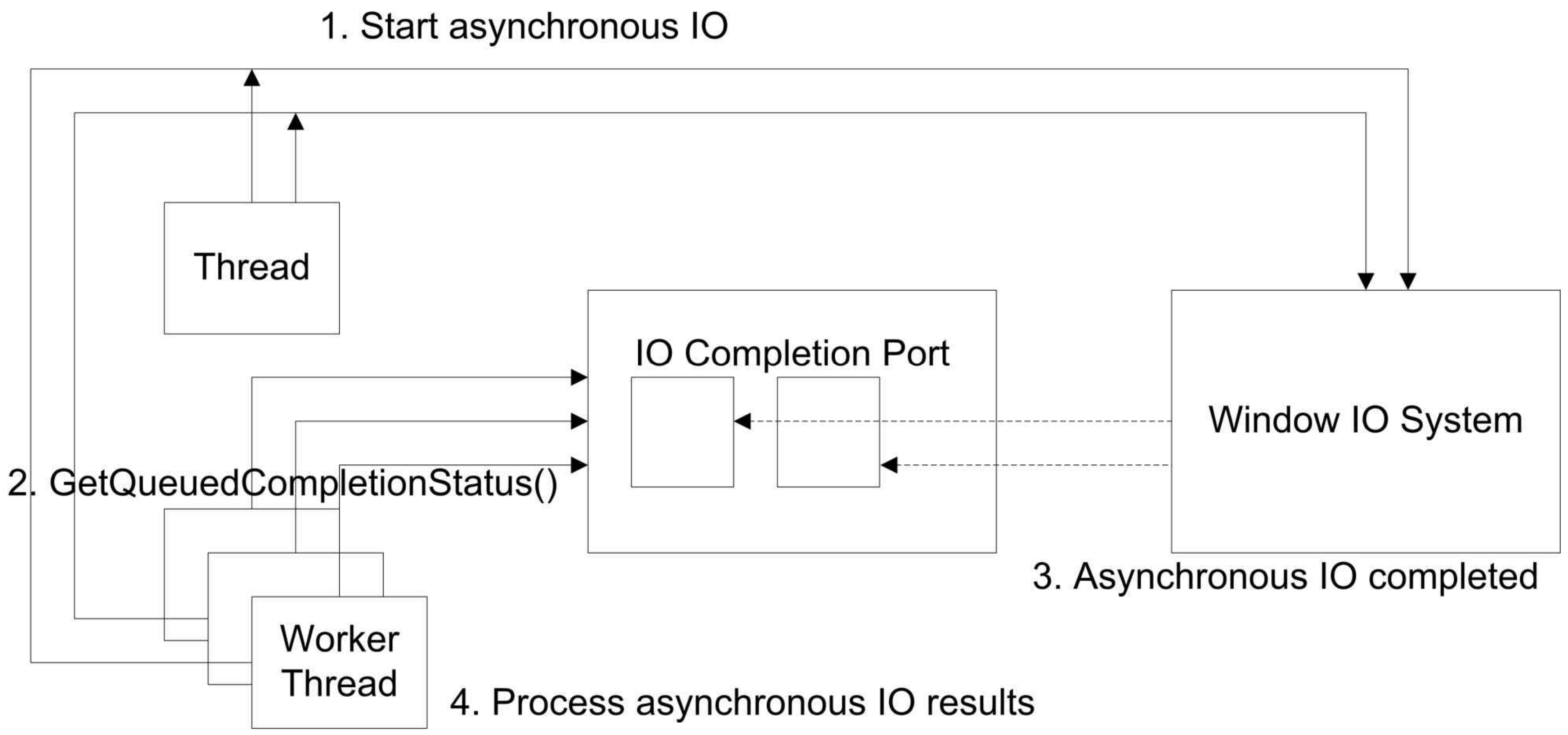


Completion Port Model

Completion Port Model

- IO completion port has results of async IO and handle of thread to process them
- IO completion port v. APC queue
  - Creation and Destroyal
  - Access control
  - Asyns IO processing method

How it works.


1. Start asynchronous IO
Thread
IO Completion Port
Window IO System
2. GetQueuedCompletionStatus()
3. Asynchronous IO completed
Worker Thread
4. Process asynchronous IO results

Socket IO Procedure

1. Using CreateIoCompletionPort(), create IO completion port.
2. Create worker threads proportional to the # of CPUs. All worker threads call GetQueuedCompletionStatus() and are in wait status.
3. Create a socket that supports asynchronous IO. To store the asynchronous IO result to IO completion port, the thread calls CreateIoCompletionPort() to connect the socket and IO completion port.
4. Call an asynchronous IO function. If the IO is not completed soon, the socket function returns error with WSA_IO_PENDING.
5. After the asynchronous IO task completed, the OS stores the results to IO completion port and wakes one thread from the wait queue. The woke thread process the results.
6. For new created socket, repeat 3~5, otherwise repeat 4~5.

CreateIOCompletionPort()

```
HANDLE CreateIOCompletionPort(
    HANDLE FileHandle,
    HANDLE ExistingCompletionPort,
    ULONG CompletionKey,
    DWORD NumberOfConcurrentThreads
);
```

- Success: IOCompletionPortHandle, Fail: NULL

GetQueuedCompletionStatus()

```
BOOL GetQueuedCompletionStatus(
    HANDLE CompletionPort,
    LPDWORD lpNumberOfBytes,
    LPDWORD lpCompletionKey,
    LPOVERLAPPED* lpOverlapped,
    DWORD dwMilliseconds
);
```

- Success: non-zero value, Fail: 0

Summary of Socket IO Models

- Synchronous IO + Asynchronous notification
  - select model
    - highly portable (can be used in Linux)
    - worst performance
    - need to use multiple threads to process 64+ sockets
  - WSAAsyncSelect model
    - Use socket event as Windows Message
    - Well coupled with GUI application
    - One window procedure processes Windows Message and Socket message --> can degrade performance
  - WSAEventSelect model
    - Mixture of select model and WSAAsyncSelect model
    - Comparable good performance
    - window procedure is not needed
    - need to use multiple threads to process 64+ sockets

Summary of Socket IO Models

- Asynchronous IO + Asynchronous notification
  - Overlapped model #1
    - Good performance using asynchronous IO
    - Need to use multiple threads to process 64+ sockets
  - Overlapped model #2
    - Good performance using asynchronous IO
    - Cannot use terminate routine for all asynchronous socket functions
  - Completion port model
    - Best performance using asynchronous IO and completion port
    - Very complicated when compared with simplest socket IO (blocking socket and one thread)
    - Can be used in Windows NT family