

# Windows Socket I/O Multiplexing

Prof. Lin Weiguo  
Copyright © 2009~2013 College of Computing, CUC

Dec 2013

# Note

---

- ▶ You should not assume that an example in this presentation is complete. Items may have been selected for illustration. It is best to get your code examples directly from the textbook and modify them to work. Use the lectures to understand the general principles.

# Outline

---

- ▶ Mechanisms for I/O Multiplexing
- ▶ Blocking with Multi-threading
- ▶ Nonblocking I/O and Polling
- ▶ I/O Multiplexing: the select Model

# Behavior of Function Calls

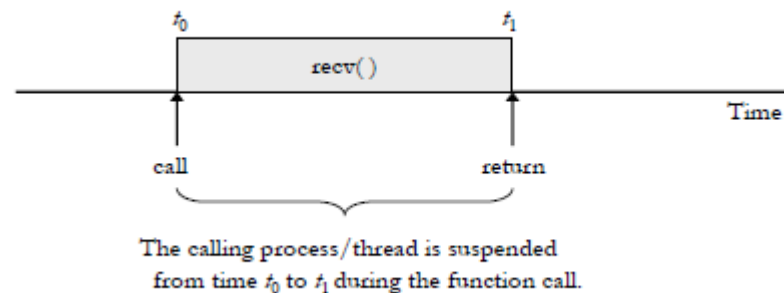
---

- ▶ **Blocking Function Call**
- ▶ **Non-Blocking Function Call**

# Blocking Function Call

- ▶ A blocking function call will not return until the function is completed or failed.
- ▶ Example:

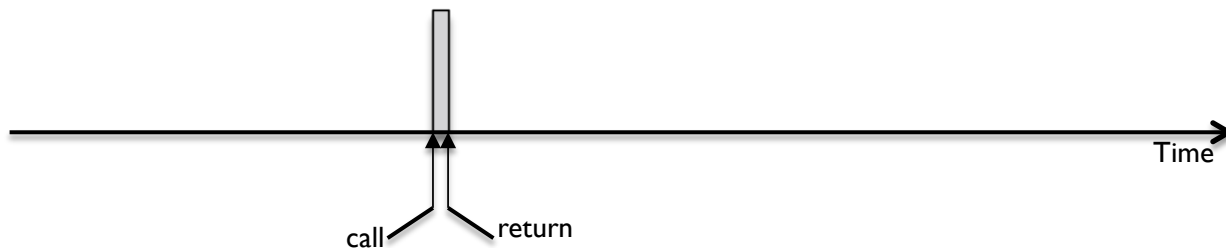
```
// Issues a blocking recv function call  
int rtv = recv(...);  
// Here either function is completed or failed.
```



# Non-Blocking Function Call

- ▶ A non-blocking function call will return immediately no matter the function can be completed or not.
- ▶ Example:

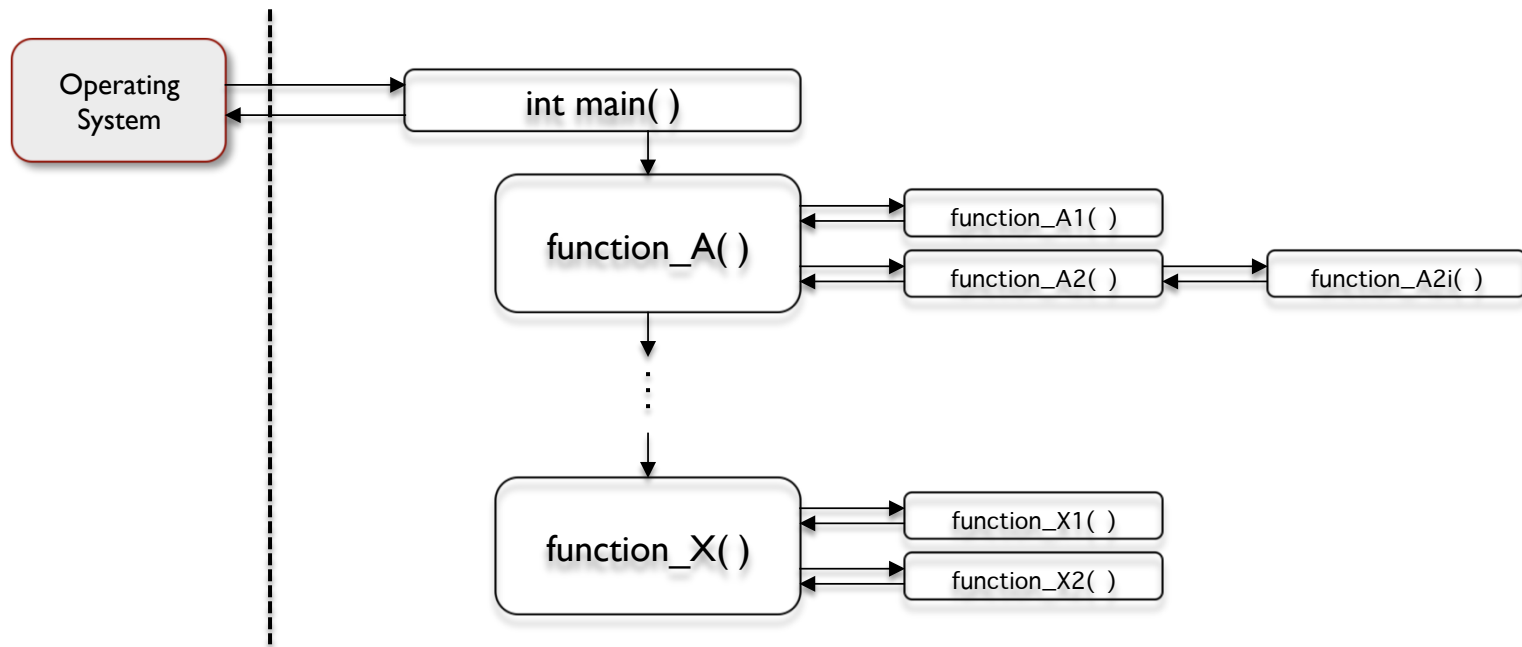
```
// Issues a non-blocking recv function call  
int rrv = recv(...);  
// recv() will return quickly regardless of  
// whether the function is completed or not.
```



Question: What if the function requires a longer time to complete?

# Synchronous Programming

- ▶ Sequential processing of program code, continuous program flow.
- ▶ Example: (function level call graph)



# Synchronous Programming on Diff OSes

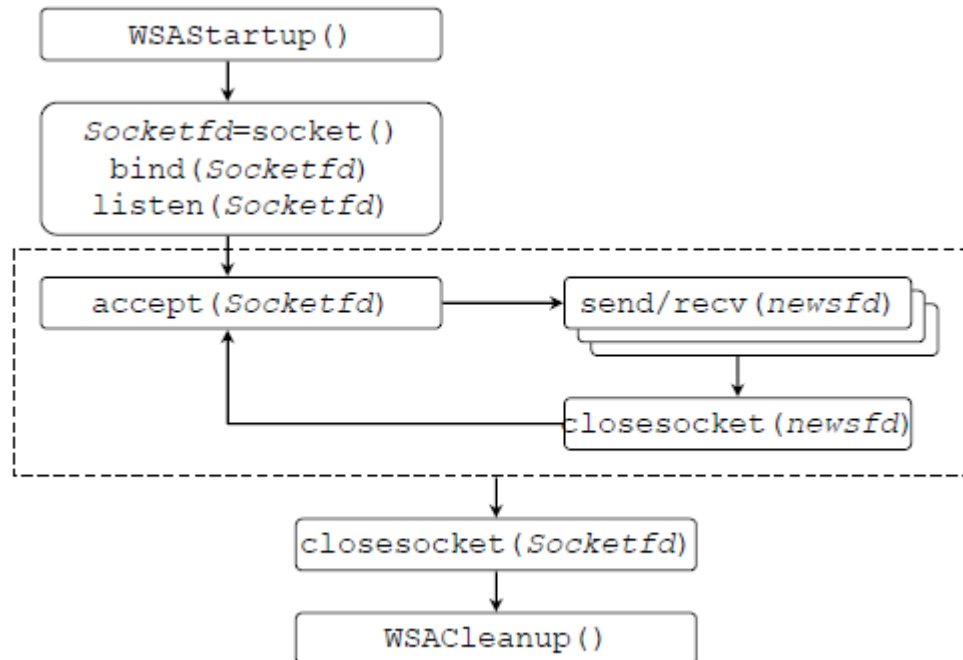
---

- ▶ Synchronous Windows programs
  - ▶ Typically only Console Mode (run in a Command Prompt window or as a Windows System Service) application are developed using the synchronous model.
- ▶ Synchronous programming is the default model in Unix and its variants.



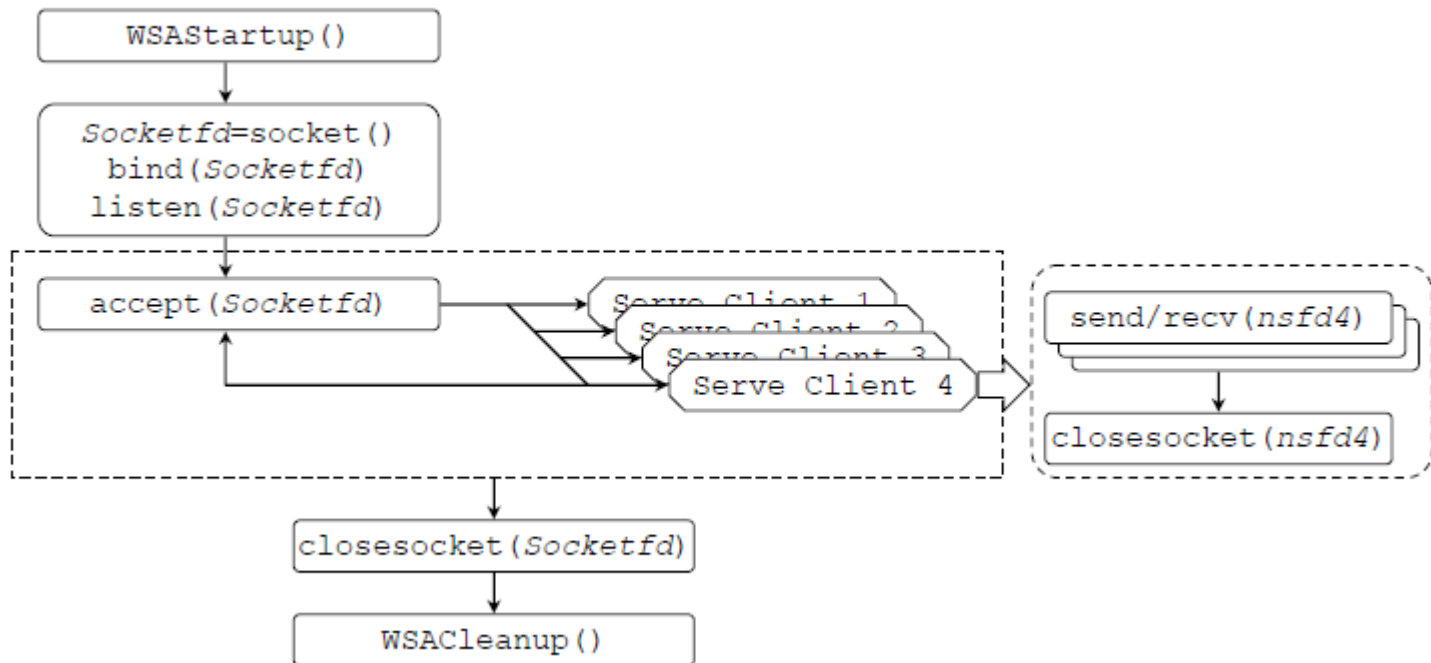
# Iterative Server

- ▶ Serves one client at a time
  - ▶ e.g., Echo Server, Comer and Stevens, Algorithm 8.1.



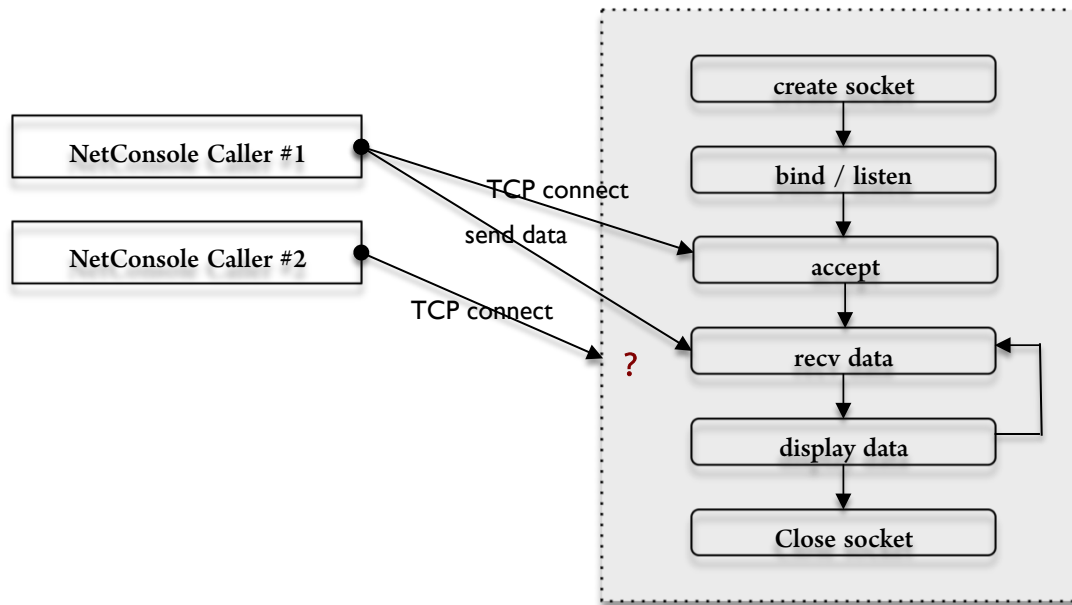
# Concurrent Server

- ▶ Able to serve multiple clients concurrently.
  - ▶ (e.g., Echo Server, Comer and Stevens, Algorithm 8.4).



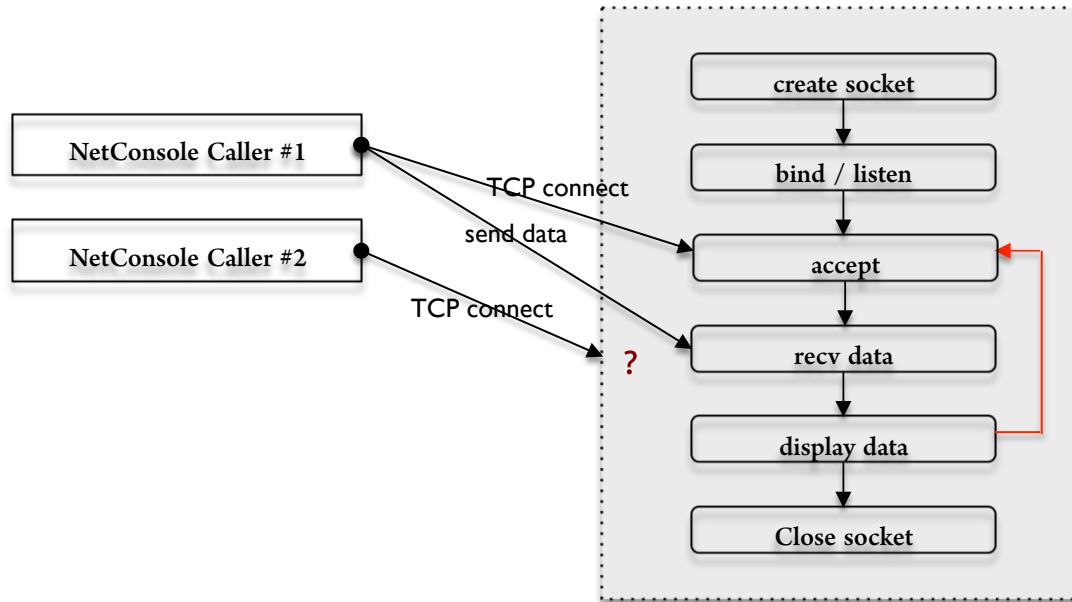
# Problem

- ▶ Problem with synchronous programming and blocking I/O model
  - ▶ Accepting a second connection:



# Problem

- ▶ Problem with synchronous programming and blocking I/O model
  - ▶ Accepting a second connection:



Question: What is the fundamental problem here?

# Mechanisms for I/O Multiplexing

---

- ▶ Synchronous Programming
  - ▶ Blocking with Multi-threading (Unix, Windows, Java)
  - ▶ Non-blocking I/O and polling (Unix, Windows)
  - ▶ The select() system call (Unix, Windows, also Java)
  - ▶ Overlapped I/O with Event Notification (Windows only)
- ▶ Asynchronous Programming
  - ▶ The WSAAsyncSelect Model
    - ▶ Message-driven I/O (Windows only)
  - ▶ The WSAEventSelect Model
    - ▶ Event-driven I/O (Windows only)
  - ▶ Overlapped I/O with Completion routines
    - ▶ Alertable I/O (Windows only)
  - ▶ Completion Ports (WinNT and later only)

---

# Blocking with Multi-threading

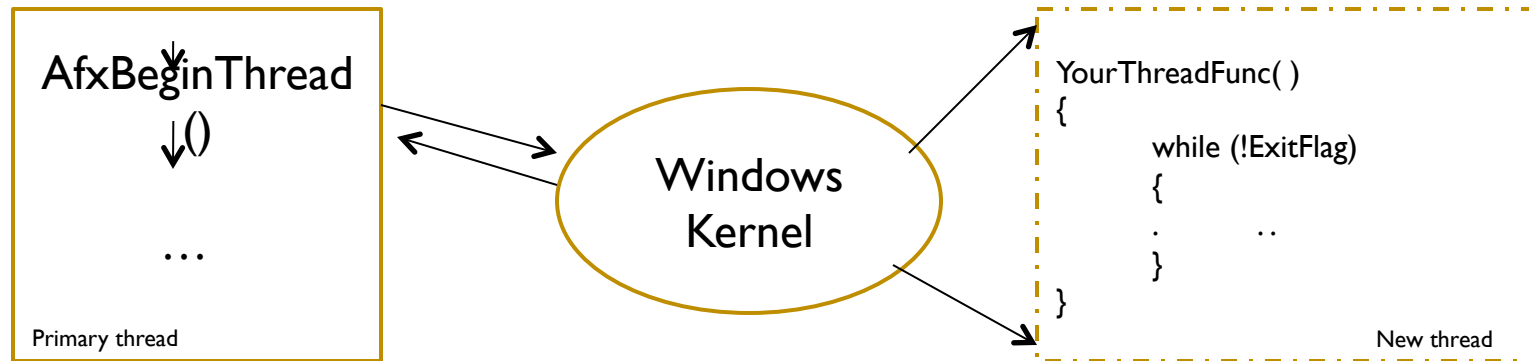
## Multithreaded Concurrent Server

# MFC: AfxBeginThread

```
CWinThread* AfxBeginThread(  
    AFX_THREADPROC pfnThreadProc,  
    LPVOID pParam,  
    int nPriority = THREAD_PRIORITY_NORMAL,  
    UINT nStackSize = 0,  
    DWORD dwCreateFlags = 0,  
    LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL  
);
```

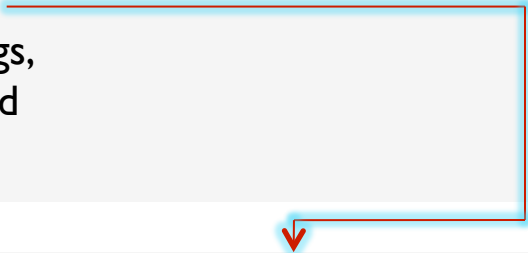
↓

```
UINT YourThreadProc(LPVOID pParam);
```



# WINAPI: CreateThread

```
HANDLE WINAPI CreateThread(
    __in_opt LPSECURITY_ATTRIBUTES lpThreadAttributes,
    __in     SIZE_T dwStackSize,
    __in     LPTHREAD_START_ROUTINE lpStartAddress,
    __in_opt LPVOID lpParameter,
    __in     DWORD dwCreationFlags,
    __out_opt LPDWORD lpThreadId
);
```



```
DWORD WINAPI ThreadProc( __in LPVOID lpParameter);
```

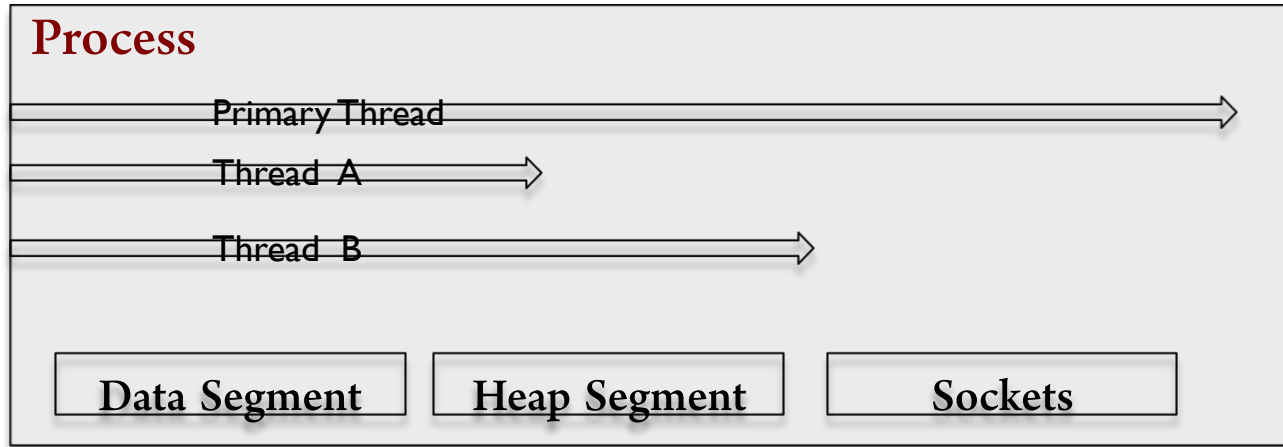
```
HANDLE Handle_Of_Thread = 0;
int Data_Of_Thread = 1;
Handle_Of_Thread = CreateThread( NULL, 0, ThreadProc, &Data_Of_Thread, 0, NULL);
if ( Handle_Of_Thread == NULL )      ExitProcess(Data_Of_Thread);
.....
CloseHandle(Handle_Of_Thread);
```



# Process and Threads

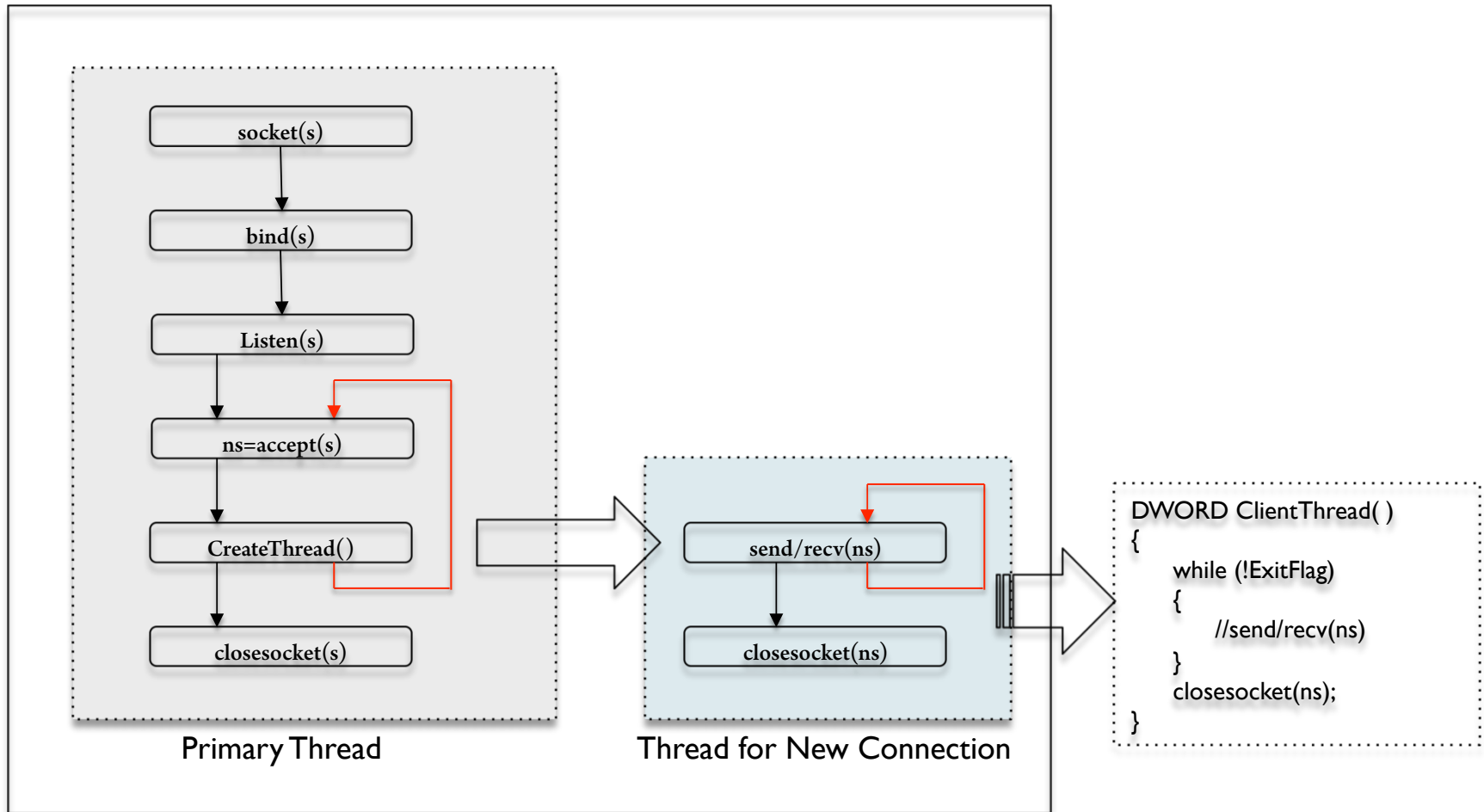
---

- ▶ Threads in the same process share all data:



- ▶ But each thread has its own stack:
  - ▶ Maximum stack size per thread is 1MB (virtual address space).
  - ▶ The maximum stack size can be configured during linking,

# Program Flow for Blocking I/O:



The application Process

# The Winsock Library

---

## ▶ Thread-Safe

- ▶ Winsock is thread-safe, Winsock internal data such as the memory pointer returned by `gethostbyname( )` will be allocated on a per-thread basis.

## ▶ Execution Order

- ▶ Order of execution for calls from multiple threads to a Winsock function such as `send( )` is unpredictable.
- ▶ Manual synchronization is required if execution order needs to be controlled.

## ▶ Blocking Function

- ▶ A blocking call such as `recv( )` will fail with error `WSAEINTR` if the socket is closed by another thread.

# What if the library is not thread-safe?

---

## ▶ Solution 1

- ▶ Don't use it or find a thread-safe equivalent.

## ▶ Solution 2

- ▶ If you know for certain that only one thread in your process will call the library then it is also ok.

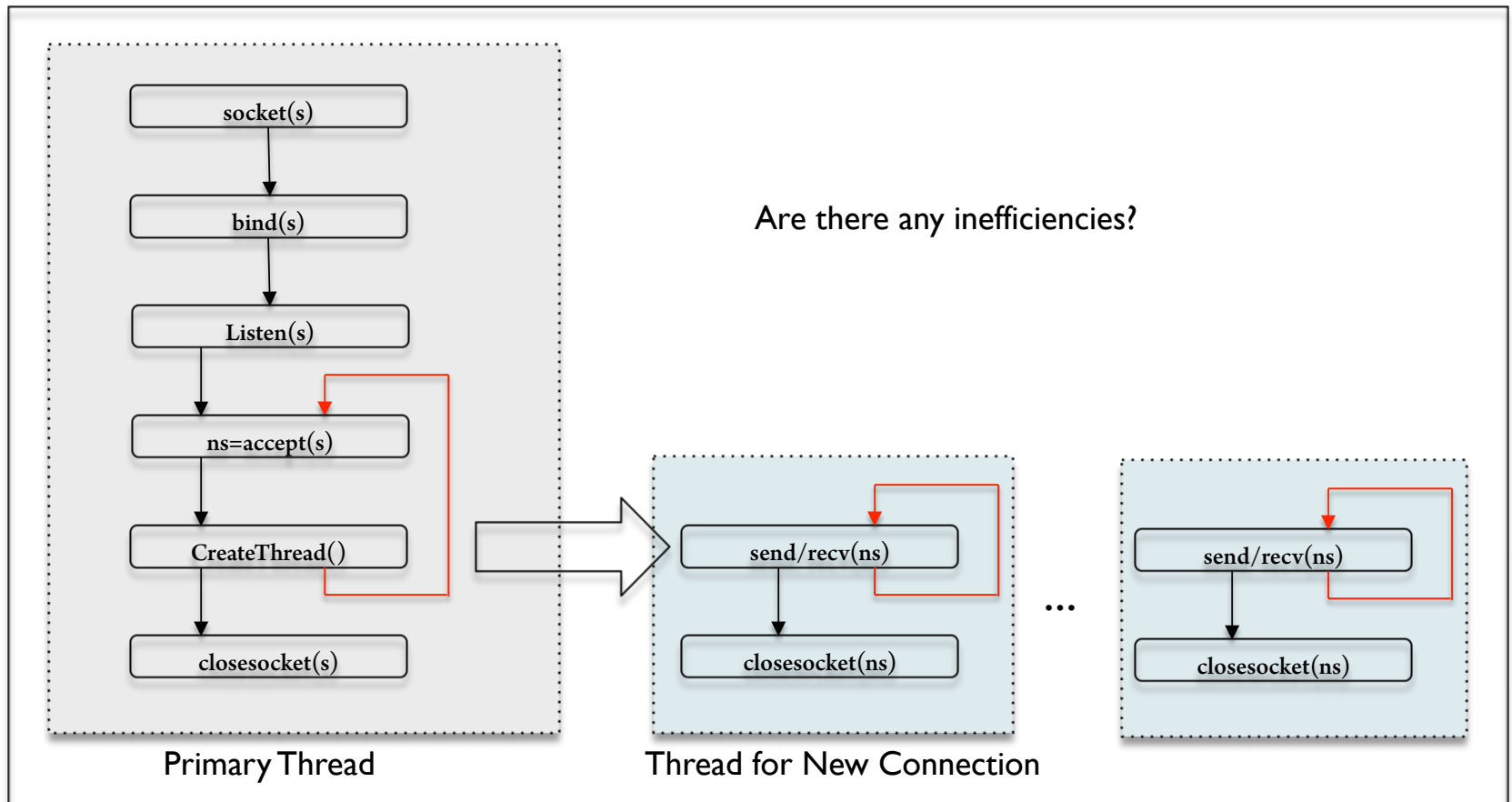
## ▶ Solution 3

- ▶ You can always manually control access to the library.
- ▶ e.g. using Critical Section:

```
// ...  
EnterCriticalSection(&critSecForNonThreadSafeLib);  
int x = FunctionInANonThreadSafeLib();  
LeaveCriticalSection(&critSecForNonThreadSafeLib);  
// ...
```

# Thread Management

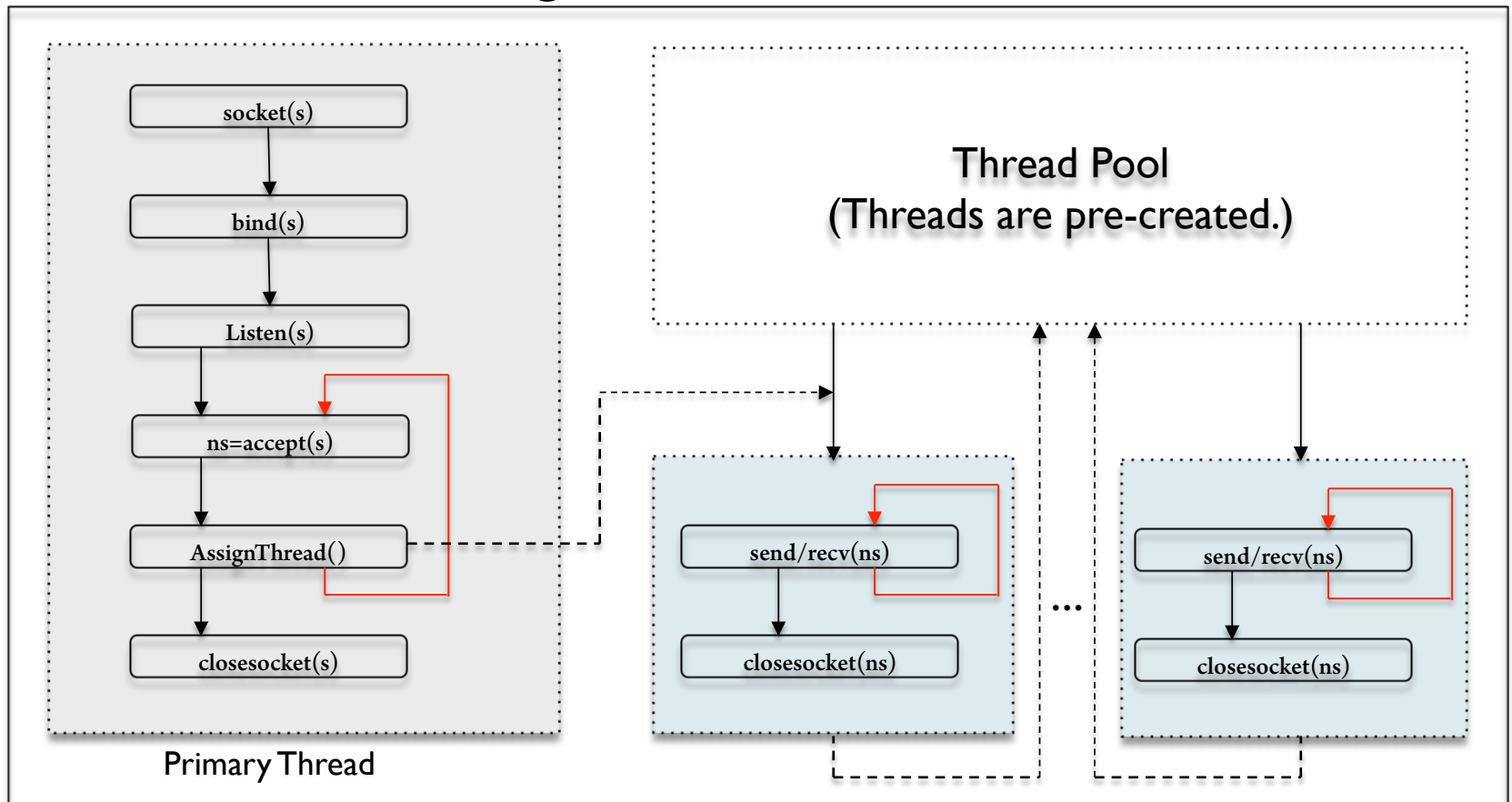
## ► Multithreaded Concurrent Server Revisited



The application Process

# Thread Management

## ► A Thread Pool Design:



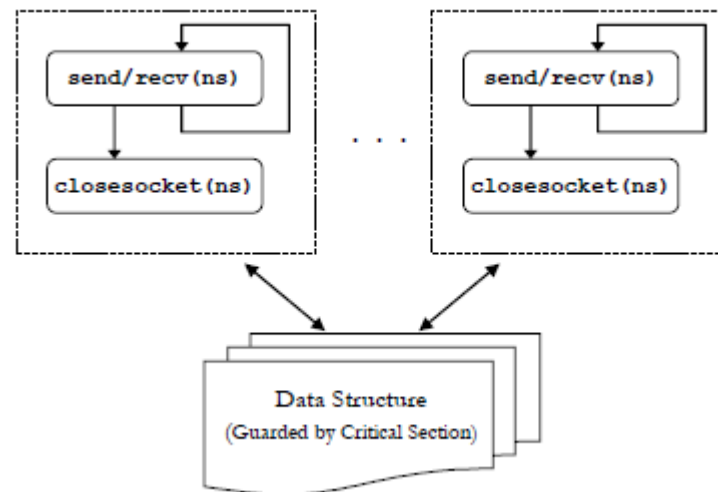
The application Process

# Thread Management

---

## ► Symmetric Multithreading

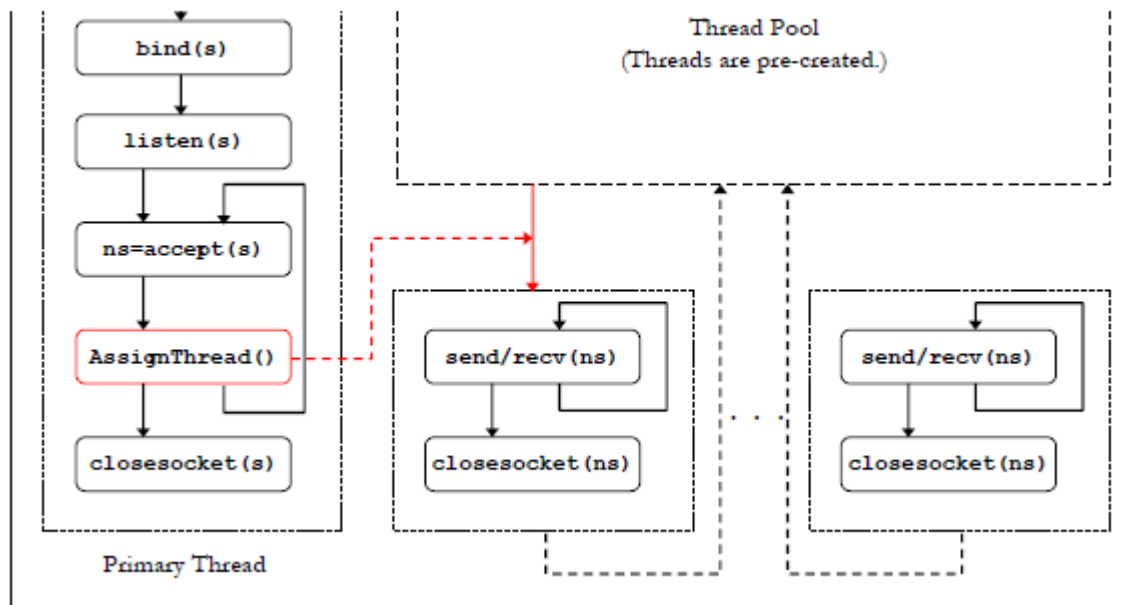
- Threads are homogeneous and perform the same function.
- E.g., the session threads in the concurrent server.
- Mutual exclusion can be used to prevent corruption of shared data.
- Minimizes data sharing can enhance thread execution efficiency.



# Thread Management

## ► Asymmetric Multithreading

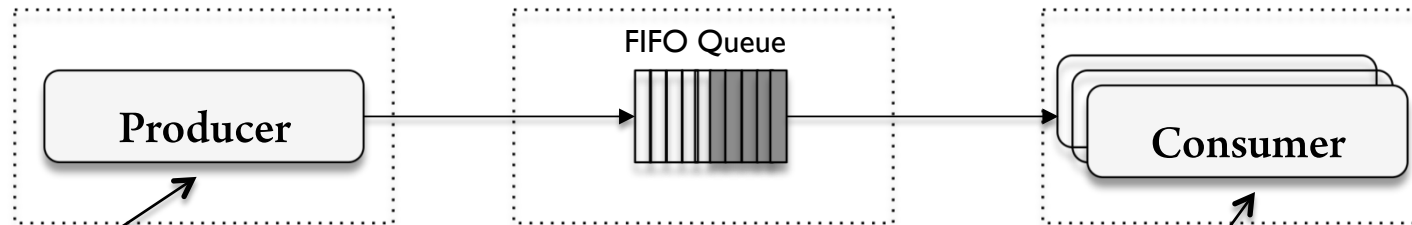
- Threads have different functions.
- E.g., the primary thread v.s. the session thread.
- How to coordinate the threads?





# Threads Synchronization

## ▶ The Producer-Consumer Model



The producer runs in a separate thread.

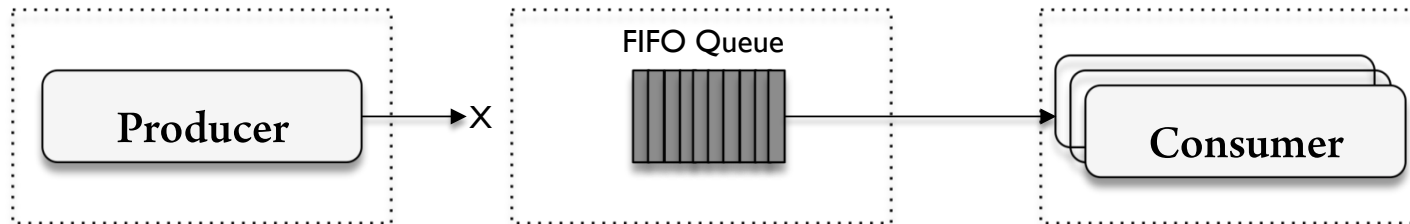
The consumer(s) also runs in separate thread(s).

## ▶ Concurrent Server Example:

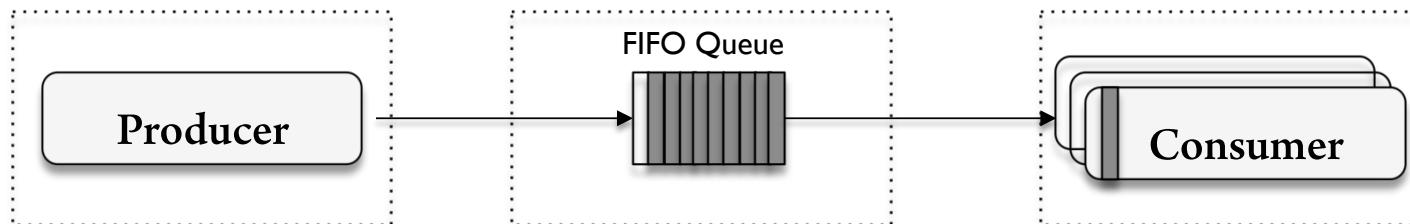
- ▶ **Producer** – the primary thread that accept() connections;
- ▶ **Consumer** – the session threads that serve clients;
- ▶ **Queue/Items** – the thread pool.

# The Producer-Consumer Model

- ▶ Producer needs to be *suspended* if the queue is full.

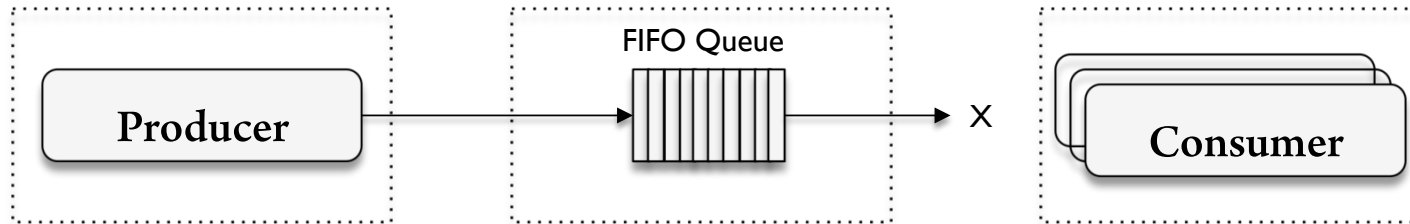


- ▶ And it needs to be *waked up* once the queue has vacant space.

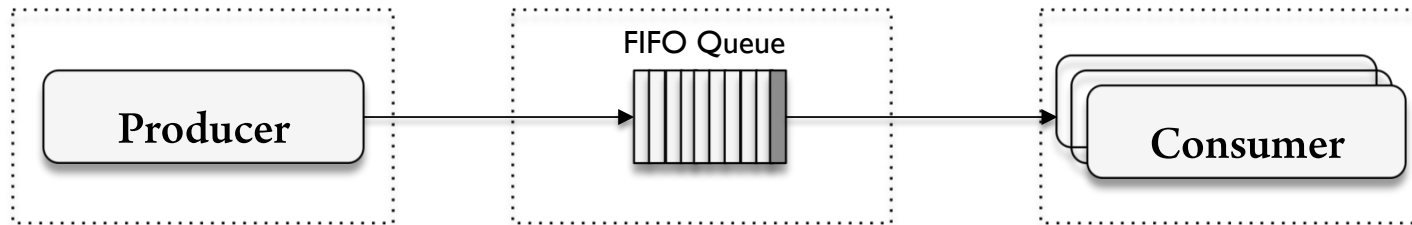


# The Producer-Consumer Model

- ▶ Consumer needs to be *suspended if the queue is empty*.



- ▶ And it needs to be *waked up once the queue becomes non-empty*.



# using CSemaphore

---

- ▶ Semaphore is a thread synchronization object that allows accessing the resource for a count between zero and maximum number of threads.
  - ▶ If the Thread enters the semaphore, the count is incremented.
  - ▶ If the thread completed the work and is removed from the thread queue, the count is decremented.
  - ▶ When the thread count goes to zero, the synchronization object is non-signaled. Otherwise, the thread is signaled.

# CSemaphore

---

- Constructs a named or unnamed CSemaphore object.

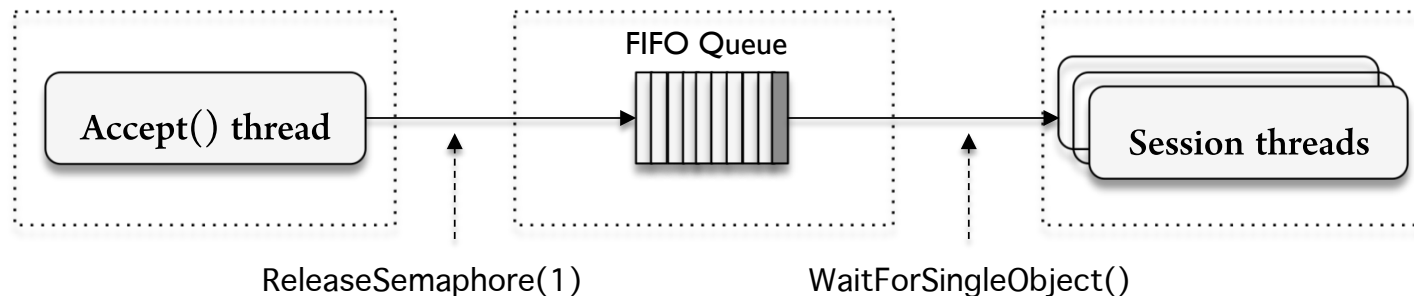
```
CSemaphore(  
    LONG lInitialCount = 1,    //The initial usage count for the semaphore.  
                                // Each time we lock the semaphore, it will decrement the  
                                // reference count by 1, until it reaches zero.  
    LONG lMaxCount = 1,       // Max threads that can simultaneously access.  
    LPCTSTR pstrName = NULL,  
    LPSECURITY_ATTRIBUTES lpsaAttributes = NULL  
);
```

```
CSemaphore g_sema(4,4);  
UINT ThreadFunction1(LPVOID lParam)  
{  
    CSingleLock lock(&g_sema);  
    lock.Lock();  
    // Process ...  
    lock.Unlock();  
    return 0;  
}
```

# Win32 Semaphore Synchronization Object

```
HANDLE g_Semaphore;  
// Create a semaphore with initial and max.  
g_Semaphore = CreateSemaphore( NULL, 4, 4, NULL);  
DWORD dwWaitResult;  
//take ownership, WaitForSingleObject decrements the semaphore's count by one.  
dwWaitResult = WaitForSingleObject( g_Semaphore, 0L);  
  
// Check the ownership  
// Release Semaphore  
ReleaseSemaphore( g_Semaphore, 1, NULL);
```

## ▶ Semaphore – number of accepted connections



# Using Semaphore for Mutual Exclusion:

---

```
// sema.cpp
#include <windows.h>
#include <stdlib.h>
#include <iostream.h>
volatile INT count;
HANDLE semaphore;
void CountThread(INT iterations)
{
    INT i;
    INT x;
    for (i=0; i<iterations; i++)
    {
        WaitForSingleObject(semaphore,INFINITE);
        x=count;
        x++;
        count=x;
        ReleaseSemaphore(semaphore, 1,NULL);
    }
}
```

# Using Semaphore for Mutual Exclusion:

---

```
const INT numThreads=4;
void main(void)
{
    CWinThread* pThread;
    HANDLE handles[numThreads];
    INT i;
    semaphore = CreateSemaphore(0, 4, 4, 0);
    for (i=0; i<numThreads; i++)
    {
        pThread =AfxBeginThread(CountThread,
                                (VOID *) 25000, THREAD_PRIORITY_NORMAL ,0, CREATE_SUSPENDED);
        if (pThread) {
            HANDLE hThread;
            DuplicateHandle(GetCurrentProcess(), pThread->m_hThread, GetCurrentProcess(), &hThread, 0, TRUE,
                            DUPLICATE_SAME_ACCESS);
            handles[i]= hThread;
            pThread->ResumeThread();
        }
    }
    // wait for all threads to finish execution
    WaitForMultipleObjects(numThreads, handles, TRUE, INFINITE);
    //use WaitForMultipleObjects(numThreads, handles, FALSE, INFINITE) to find out which thread died
    CloseHandle(semaphore);
    cout << "Global count = " << count << endl;
}
```

---




# Multithreading with GUI

---

## ► Define a sending thread

```
class CMyDlg : public CDialog
{
    // ... codes omitted ...
    // Thread-Driven Send //
    static UINT ThreadSendProc(LPVOID lpInstance);
    int StartThreadedSend();
    int StopThreadedSend();
    HANDLE hThread;
    // ... codes omitted ...
};
```



Thread procedure for sending packets.  
Note the use of the static qualifier.

# Multithreading with GUI

---

## ▶ Start the sending thread

```
int CMyDlg::StartThreadedSend()
{
    CWinThread* pThread;
    AfxBeginThread(CMyDlg::ThreadSendProc,
        (LPVOID)this, // passing this pointer to new thread
        THREAD_PRIORITY_NORMAL);
    if (!pThread){
        AfxMessageBox(_T("Error starting new sending thread!"));
    }
    return 0;
}
```

# Multithreading with GUI

---

## ▶ The Sending thread proc

```
UINT CMyDlg::ThreadSendProc(LPVOID lpInstance)
{
    CMyDlg *pDlg = (CMyDlg *)lpInstance;
    // [refresh_interval] //
    pDlg ->iRefreshInterval = pDlg ->m_EditRefreshInterval;
    // ... codes omitted //
}
```

- ▶ Access to class members must be through the explicitly-passed this pointer (pDlg) because the function is static.

# Multithreading with GUI

---

- ▶ **Passing pointer or handle of a MFC objects?**
  - ▶ pass the pointer to your object, not the hWnd, so that the new thread can get both. This is not thread-safe, since you might try to use this pointer to call methods of your CMyDlg object, and you'll reach some failing assertions!
  - ▶ post or send messages to the native window having hWnd as handle, and these messages will be processed in the thread that created the dialog. This might be slower, but safer.

```
CWinThread* pThread =  
    AfxBeginThread(  
        ComputeThreadProc, // The address of the thread function  
        GetSafeHwnd(), // 32-bit value that passes to the thread function  
        THREAD_PRIORITY_NORMAL); // The desired priority of the thread
```

# Multithreading with GUI

---

## ► Passing a struct pointer

```
int CMyDlg::StartThreadedSend()
{
    ThreadInfo * threadInfo = new ThreadInfo;
    threadInfo->RefreshInterval=m_EditRefreshInterval;
    ...
    threadInfo->thisHandle = GetSafeHwnd();
    AfxBeginThread(ThreadProc, (LPVOID) threadInfo, 0);
    return 0;
}
```

```
UINT CMyDlg ::ThreadProc( LPVOID pParam ) {
    ThreadInfo * threadInfo = (ThreadInfo *) pParam;
    int m_RefreshInterval= threadInfo->RefreshInterval;
    HWND hDlgWnd = threadInfo->thisHandle;
    //update UI
    ::PostMessage(hDlgWnd ,WM_COMPLETE, (WPARAM) BufferSize, (LPARAM) hShareBuff);
    return 0;
}
```

---

# Nonblocking I/O and Polling

# Changing to nonblocking mode

---

- ▶ When a socket is created, by default it is a blocking socket. Under blocking mode socket I/O operations, connect and accept operations all block until the operation in queue is completed.
- ▶ To change the socket operation mode from blocking mode to nonblocking mode, you can either use
  - ▶ `WSAAsyncSelect`,
    - ▶ maps socket notifications to Windows messages and is the best model for a single threaded GUI application.
  - ▶ `WSAEventSelect`, or
    - ▶ uses `WSAEnumNetworkEvents` to determine the nature of the socket notification on the signaling event and maps socket notifications by signaling an event. This is a useful model for non-GUI applications that lack a message pump, such as a Windows NT service application.
  - ▶ the `FIONBIO` command in the `ioctlsocket` API call.
    - ▶ puts the socket into nonblocking mode as well. But you need to poll the status of the socket by using the `select` API.

# What has been changed for nonblocking ?

## ▶ **WSAGetLastError()=WSAEWOULDBLOCK**

**WSAEWOULDBLOCK (10035) Resource temporarily unavailable.**

- WinSock description: The socket is marked as non-blocking (non-blocking operation mode), and the requested operation is not complete at this time.
- Detailed descriptions:
  - connect(): the operation is underway, but as yet incomplete.
  - closesocket(): occurs on a non-blocking socket with non-zero timeout set with setsockopt() SO\_LINGER. The behavior may vary: some WinSocks might complete in background, and others may require another call to closesocket to complete. Do not set non-zero timeout on non-blocking sockets to avoid this ambiguity (see Chapter 9 for more information).
  - send() or sendto(): out of buffer space, so try again later or wait until FD\_WRITE notification (WSAAsyncSelect()) or select() writefds is set.
  - all other functions: retry the operation again later since it cannot be satisfied at this time.
- Developer suggestions:

Every application that uses non-blocking sockets must be prepared for this error on any call to the functions mentioned below. For instance, even if you request to send() a few bytes of data on a newly created TCP connection, send() could fail with WSAEWOULDBLOCK (if, say, the network system has a TCP slow-start algorithm implemented). The WSAAsyncSelect() FD\_WRITE event is specifically designed to notify an application after a WSAEWOULDBLOCK error when buffer space is available again so send() or sendto() should succeed.



# Behavior of Nonblocking functions

---

## ▶ The **send** function

- ▶ If no buffer space is available within the transport system to hold the data to be transmitted, **send will block unless the socket has been placed in nonblocking mode.**
- ▶ On nonblocking stream oriented sockets, the number of bytes written can be between 1 and the requested length, depending on buffer availability on both the client and server computers. The *select*, *WSAAsyncSelect* or *WSAEventSelect* functions can be used to determine when it is possible to send more data.
- ▶ Return value:
  - ▶ `SOCKET_ERROR` is returned,
  - ▶ Error code `WSAEWOULDBLOCK` can be retrieved by calling `WSAGetLastError()` .

# Behavior of Nonblocking functions

---

## ▶ The **recv** function

- ▶ If no incoming data is available at the socket, the **recv** call **blocks** and waits for data to arrive according to the blocking rules defined for `WSARecv` with the `MSG_PARTIAL` flag not set **unless the socket is nonblocking**.
- ▶ In this case, a value of **`SOCKET_ERROR`** is returned with the error code set to **`WSAEWOULDBLOCK`**. The **`select`**, **`WSAAsyncSelect`**, or **`WSAEventSelect`** functions can be used to determine when it is possible to accept a coming connection request..

# Behavior of Nonblocking functions

---

## ▶ The **connect** function

- ▶ With a nonblocking socket, the connection attempt cannot be completed immediately. In this case, connect will return `SOCKET_ERROR`, and `WSAGetLastError` will return `WSAEWOULDBLOCK`.
- ▶ In this case, there are three possible scenarios:
  - ▶ Use the `select` function to determine the completion of the connection request by checking to see if the socket is writeable.
  - ▶ If the application is using `WSAAsyncSelect` to indicate interest in connection events, then the application will receive an `FD_CONNECT` notification indicating that the connect operation is complete (successfully or not).
  - ▶ If the application is using `WSAEventSelect` to indicate interest in connection events, then the associated event object will be signaled indicating that the connect operation is complete (successfully or not).

# Behavior of Nonblocking functions

---

## ▶ The **accept** function

- ▶ The **accept** function can block the caller until a connection is present if no pending connections are present on the queue, and the socket is marked as blocking.
- ▶ If the socket is marked as nonblocking and no pending connections are present on the queue, **accept** returns an error and a specific error code **WSAEWOULDBLOCK** can be retrieved by calling **WSAGetLastError** .
- ▶ In this case , the socket is marked as nonblocking and no connections are present to be accepted. The **select**, **WSAAsyncSelect**, or **WSAEventSelect** functions can be used to accept a coming connection.

# Nonblocking I/O and Polling

## ► Turns a socket into non-blocking mode:

```
void ConcurrentListenerUsingNonBlockingIOandPolling(char *address, char *port)
{
    /// Step 1: Prepare address structures. ///
    sockaddr_in *TCP_Addr = new sockaddr_in;
    memset (TCP_Addr, 0, sizeof(struct sockaddr_in));
    TCP_Addr->sin_family = AF_INET;
    TCP_Addr->sin_port = htons(atoi(port));
    TCP_Addr->sin_addr.s_addr = INADDR_ANY;
    /// Step 2: Create a socket for incoming connections. ///
    SOCKET Sockfd = socket(AF_INET, SOCK_STREAM, 0);
    bind(Sockfd, (struct sockaddr *)TCP_Addr, sizeof(struct sockaddr_in));
    listen(Sockfd, 5);
    unsigned long ul_val = 1;
    if (ioctlsocket(Sockfd, FIONBIO, &ul_val) == SOCKET_ERROR) {
        printf("\nioctlsocket() failed. Error code: %i\n", WSAGetLastError());
        return;
    }
    // ...
}
```

# Nonblocking I/O and Polling

## ► accept() will become non-blocking:

```
/// Step 3: Setup the data structures for multiple connections. ///
```

```
const int maxSockets = 10;  
SOCKET socketHandles[maxSockets]; // Array for the socket handles  
bool socketValid[maxSockets]; // Bitmask to manage the array  
int numActiveSockets = 1;  
for (int i=1; i<maxSockets; i++) socketValid[i] = false;  
socketHandles[0] = Sockfd; socketValid[0] = true;  
/// Step 4: Poll all active sockets for data/accept. ///
```

```
while (1) {  
    if (socketValid[0]) { // Check for incoming connection first.  
        SOCKET newsfd = accept(Sockfd, 0, 0);  
        if (newsfd != SOCKET_ERROR) { // accept() succeeded  
            // Append the new entry to the socketHandles[] array//  
            socketHandles[numActiveSockets] = newsfd;  
            socketValid[numActiveSockets] = true;  
            ++numActiveSockets;  
            if (numActiveSockets == maxSockets) { // Suspend accept  
                socketValid[0] = false;  
            }  
        } else if (WSAGetLastError() != WSAEWOULDBLOCK) {  
            printf("\naccept() failed. Error code: %i\n", WSAGetLastError());  
            return;  
        }  
    }  
}
```

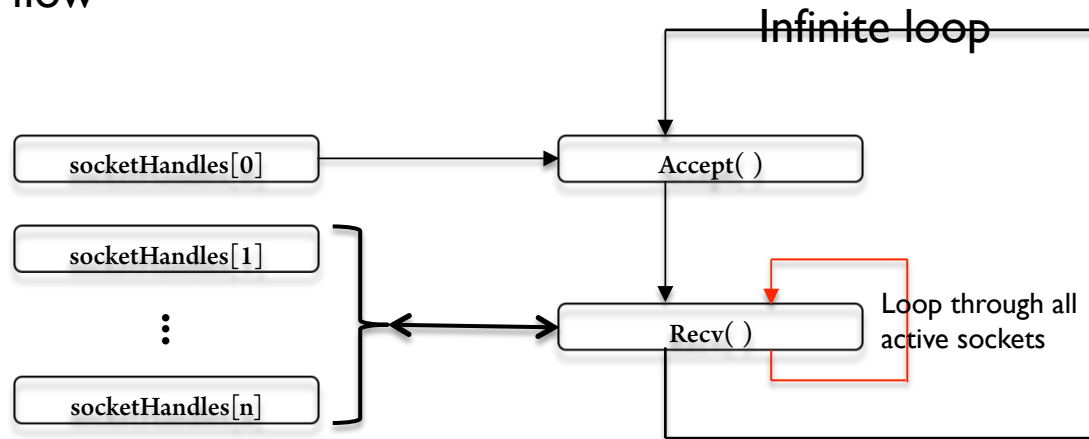
# Nonblocking I/O and Polling

## ► Poll the rest of the active sockets:

```
// ...
for (int i=1; i<numActiveSockets; i++) {
    if (socketValid[i]) {
        // Receive data and display to stdout. //
        char buf[256]; int len=255;
        int numread = recv(socketHandles[i], buf, len, 0);
        if (numread == SOCKET_ERROR) {
            if (WSAGetLastError() != WSAEWOULDBLOCK) {
                printf("\nrecv() failed. Error: %i\n", WSAGetLastError());
                return;
            }
            continue;
        } else if (numread == 0) { // connection closed
            // Pack the socket array
            closesocket(socketHandles[i]);
            --numActiveSockets;
            socketHandles[i] = socketHandles[numActiveSockets];
            socketValid[numActiveSockets] = false;
            if (numActiveSockets == (maxSockets-1)){
                socketValid[0] = true;
            } else {
                printf("[%i]: ", i);   for (int x=0; x<numread; x++) putchar(buf[x]);
            }
        }
    }
} // end-for
```

# Nonblocking I/O and Polling

## ▶ Program flow



Question: What is the tradeoff of polling non-blocking sockets?

- ▶ Blocking sockets are easier to use from a conceptual standpoint but become difficult to manage when dealing with multiple connected sockets or when data is sent and received in varying amounts and at arbitrary times.
- ▶ On the other hand, non-blocking sockets are more difficult because more code needs to be written to handle the possibility of receiving a `WSAEWOULDBLOCK` error on every Winsock call.



---

# I/O Multiplexing: the select Model

# I/O Multiplexing: the select Model

- ▶ The select function blocks for I/O operations until the conditions specified as parameters are met. The function prototype for select is as follows:

```
int select(  
    int nfds,           // [in] Ignored in windows, for compatibility with Unix.  
    fd_set* readfds,    // [in, out] Set of sockets for read events.  
    fd_set* writefds,   // [in, out] Set of sockets for write events.  
    fd_set* exceptfds,  // [in, out] Set of sockets for exceptions/errors.  
    const struct timeval* timeout // [in] Max time to block.  
);  
// returns the total number of socket handles that are ready and contained in the fd\_set structures, zero if the  
// time limit expired, or SOCKET_ERROR if an error occurred.
```

- ▶ The readfds set identifies sockets that meet one of the following conditions:
  - ▶ Data is available for reading.
  - ▶ Connection has been closed, reset, or terminated.
  - ▶ If listen has been called and a connection is pending, the accept function will succeed.
- ▶ The writefds set identifies sockets in which one of the following is true:
  - ▶ Data can be sent.
  - ▶ If a non-blocking connect call is being processed, the connection has succeeded.
- ▶ Finally, the exceptfds set identifies sockets in which one of the following is true:
  - ▶ If a non-blocking connect call is being processed, the connection attempt failed.
  - ▶ OOB data is available for reading.

# I/O Multiplexing: the select Model

---

## ▶ fd\_set structure:

```
typedef struct fd_set {  
    u_int fd_count; // Number of sockets in the set.  
    SOCKET fd_array[FD_SETSIZE]; // Array of sockets that are in the set.  
} fd_set;
```

- ▶ Timeout: how long the select will wait for I/O to complete.
  - ▶ If timeout is a null pointer, select will block indefinitely until at least one descriptor meets the specified criteria

```
typedef struct timeval {  
    long tv_sec; // seconds  
    long tv_usec; // microseconds  
} timeval;
```

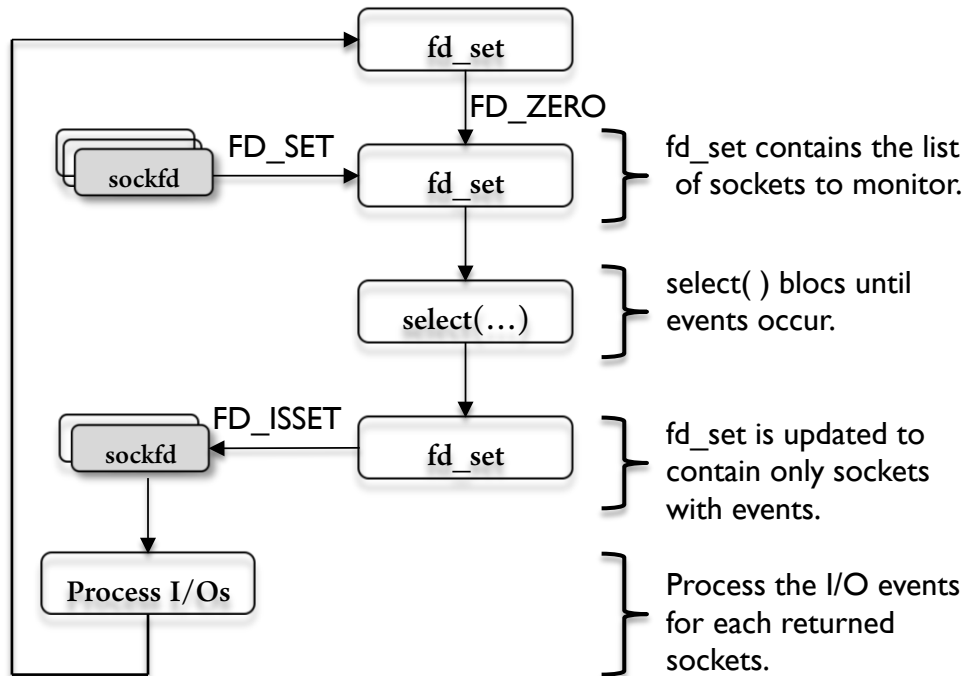
# I/O Multiplexing: the select Model

---

- ▶ **Macros to manipulate and check the fd\_set sets for I/O activity.**
  - **FD\_ZERO(\*set)** Initializes set to the empty set. A set should always be cleared before using.
  - **FD\_CLR(s, \*set)** Removes socket s from set.
  - **FD\_ISSET(s, \*set)** Checks to see if s is a member of set and returns TRUE if so.
  - **FD\_SET(s, \*set)** Adds socket s to set.
- ▶ **Steps describe the basic flow of an application that uses select with one or more socket handles:**
  1. Initialize each fd\_set of interest by using the FD\_ZERO macro.
  2. Assign socket handles to each of the fd\_set sets of interest by using the FD\_SET macro.
  3. Call the select function and wait until I/O activity sets one or more of the socket handles in each fd\_set set provided. When select completes, it returns the total number of socket handles that are set in all of the fd\_set sets and updates each set accordingly.
  4. Using the return value of select, your application can determine which application sockets have I/O pending by checking each fd\_set set using the FD\_ISSET macro.
  5. After determining which sockets have I/O pending in each of the sets, process the I/O and go to step 1 to continue the select process.

# I/O Multiplexing: the select Model

## ► Operational Scenario:



```
SOCKET s;  
fd_set fdread;  
int ret;  
// Create a socket, and accept a connection  
// Manage I/O on the socket  
while(TRUE)  
{  
    // Always clear the read set before calling select()  
    FD_ZERO(&fdread);  
    // Add socket s to the read set  
    FD_SET(s, &fdread);  
  
    if ((ret = select(0, &fdread, NULL, NULL, NULL)) ==  
        SOCKET_ERROR)  
    {  
        // Error condition  
    }  
    if (ret > 0)  
    {  
        // application should check to see whether the  
        // socket is part of a set.  
        if (FD_ISSET(s, &fdread))  
        {  
            // A read event has occurred on socket s  
        }  
    }  
}
```

# Concurrent Server

## ► Implements Listener using select():

```
void ConcurrentListenerUsingSelect (char *address, char *port)
{
    /// Step 1: Prepare address structures. ///
    sockaddr_in *TCP_Addr = new sockaddr_in;
    memset (TCP_Addr, 0, sizeof(struct sockaddr_in));
    TCP_Addr->sin_family = AF_INET;
    TCP_Addr->sin_port = htons(atoi(port));
    TCP_Addr->sin_addr.s_addr = INADDR_ANY;
    /// Step 2: Create a socket for incoming connections. ///
    SOCKET Sockfd = socket(AF_INET, SOCK_STREAM, 0);
    bind(Sockfd, (struct sockaddr *)TCP_Addr, sizeof(struct sockaddr_in));
    listen(Sockfd, 5);
    /// Step 3: Setup the data structures for multiple connections. ///
    const int maxSockets = 10; // At most 10 concurrent clients
    SOCKET socketHandles[maxSockets]; // Array for the socket handles
    bool socketValid[maxSockets]; // Bitmask to manage the array
    int numActiveSockets = 1;
    for (int i=1; i<maxSockets; i++) socketValid[i] = false;
    socketHandles[0] = Sockfd;
    socketValid[0] = true;
```

continue ...

# Concurrent Server

## ► Implements Listener using select():

```
/// Step 4: Setup the select() function call for I/O multiplexing. ///
fd_set fdReadSet;
while (1) {
    // Setup the fd_set //
    FD_ZERO(&fdReadSet);
    for (int i=0; i<maxSockets; i++) {
        if (socketValid[i] == true) {
            FD_SET(socketHandles[i], &fdReadSet);
        }
    }
    // Block on select() //
    int ret;
    if ((ret = select(numActiveSockets, &fdReadSet, NULL, NULL, NULL)) == SOCKET_ERROR)
    {
        printf("\nselect() failed. Error code: %i\n", WSAGetLastError());
        return;
    }
}
```

continue ...

# Concurrent Server

## ► Implements Listener using select():

```
// Process the active sockets //
for (i=0; i<maxSockets; i++) {
    if (!socketValid[i]) continue; // Only check for valid sockets.
    if (FD_ISSET(socketHandles[i], &fdReadSet)) { // Is socket i active?
        if (i==0) { // [0] is the server listening socket
            SOCKET newsfd = accept(Sockfd, 0, 0); // accept new connection //
            // Find a free entry in the socketHandles[] //
            int j = 1;
            for (; j<maxSockets; j++) {
                if (socketValid[j] == false) {
                    socketValid[j] = true;
                    socketHandles[j] = newsfd;
                    ++numActiveSockets;
                    if (numActiveSockets == maxSockets) {
                        // Ignore new accept()
                        socketValid[0] = false;
                    }
                    break;
                }
            }
        } //else other [i] are data transmission sockets
    }
}
```

continue ...



# Concurrent Server

## ► Implements Listener using select():

```
else { // sockets for recv()
    // Receive data and display to stdout. //
    char buf[256]; int len=255;
    int numread = recv(socketHandles[i], buf, len, 0);
    if (numread == SOCKET_ERROR) {
        printf("\nrecv() failed. Error code: %i\n",
            WSAGetLastError());
        return;
    } else if (numread == 0) { // connection closed
        // Update the socket array
        socketValid[i] = false;
        --numActiveSockets;
        if (numActiveSockets == (maxSockets-1))
            socketValid[0] = true;
        closesocket(socketHandles[i]);
    } else {
        printf("[%i]: ", i);
        for (int x=0; x<numread; x++) putchar(buf[x]);
    }
}
if (--ret == 0) break; // All active sockets processed.
}
}
```

continue ...

# Remarks on select()

## ▶ Setting up the fd\_set in every iteration:

/// Step 4: Setup the select() function call for I/O multiplexing. ///

```
fd_set fdReadSet;
// Setup the fd_set //
FD_ZERO(&fdReadSet);
for (int i=0; i<maxSockets; i++) {
    if (socketValid[i] == true) {
        FD_SET(socketHandles[i], &fdReadSet);
    }
}
```

## ▶ Finding out which socket has event for processing:

```
// Process the active sockets //
for (i=0; i<maxSockets; i++) { // Scan through all valid sockets
    if (!socketValid[i]) continue; // Only check for valid sockets.
    if (FD_ISSET(socketHandles[i], &fdReadSet)) { // Is socket i active?
        if (i==0) { // the socket for accept(); [0] is the server listening socket
            // accept new connection //
            // ...(omitted)
        }...
    }
}
```

# Summary

---

- ▶ The advantage of using select is the capability to multiplex connections and I/O on many sockets from a single thread. This prevents the explosion of threads associated with blocking sockets and multiple connections.
- ▶ The disadvantage is the maximum number of sockets that may be added to the `fd_set` structures.
  - ▶ By default, the maximum is defined as `FD_SETSIZE`, which is defined in `WINSOCK2.H` as 64. To increase this limit, an application might define `FD_SETSIZE` to something large. This define must appear before including `WINSOCK2.H`. Also, the underlying provider imposes an arbitrary maximum `fd_set` size, which typically is 1024 but is not guaranteed to be.
  - ▶ Finally, for a large `FD_SETSIZE`, consider the performance hit of setting 1000 sockets before calling select followed by checking whether each of those 1000 sockets is set after the call returns.

# Refs

---

- ▶ [Description of CWnd derived MFC objects and multithreaded applications in Visual C++](#)
- ▶ MSDN: [CreateThread](#) function
- ▶ MSDN: [ioctlsocket](#), [setsockopt](#) Function
- ▶ MSDN: [WSAAsyncSelect](#), [WSAEventSelect](#) Function
- ▶ MSDN: [select](#) Function
- ▶ CUHK [lerg4180](#): Network Software Design and Programming