Preliminaries II
Dmitri Loguinov
Texas A&M University
January 21, 2016
Agenda

• Windows sockets
  – Clients
• HTTP basics
• Multi-threading
  – Threads
  – Synchronization
  – Producer-consumer
Windows Sockets

• Sockets are interfaces to the TCP/IP protocol stack
  – More on TCP and IP later in the semester
  – HTTP (hw1) uses TCP
  – Sockets identified by their handle

• Communication using sockets is accomplished by a set of system calls to Winsock
  – Winsock is the Windows implementation of sockets
  – Parts are identical to Berkeley sockets in Unix

• TCP sockets can be used in two modes:
  – Client (socket actively establishes outgoing connections)
  – Server (socket listens for incoming connections)
Windows Sockets 2

- **IP address**: uniquely identifies the host to be contacted
  - 4-byte number written with a dot between each byte
  - How to assign IP 128.194.135.60 to an integer in C++?
- **Localhost** has IP address 127.0.0.1
- What if multiple network applications need to be run simultaneously on one host?
- **Solution**: ports
  - Each socket is bound to unique port
  - Socket = OS handle, port = externally visible identifier
  - The OS forwards incoming messages to sockets based on ports they are bound to
Windows Sockets 3

• Ports are 2-byte unsigned integers
  – Port 0 reserved, 1-1023 are system; 1024-65535 user

• Some well-known ports
  – HTTP: 80 (often moved to 8000 or 8080)
  – Telnet: 23
  – SSH: 22
  – SMTP: 25 (often moved to 465, 587)

• See http://www.iana.org/assignments/port-numbers

• When issuing a connect, the OS implicitly binds the socket to the next available port
  – Clients do not need to worry about their port numbers
  – Binding is mandatory for servers
Example (Windows)

- Use “netstat –a” to see open ports on your host

<table>
<thead>
<tr>
<th>Proto</th>
<th>Local Address</th>
<th>Foreign Address</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>viper:echo</td>
<td>viper:0</td>
<td>LISTENING</td>
</tr>
<tr>
<td>TCP</td>
<td>viper:discard</td>
<td>viper:0</td>
<td>LISTENING</td>
</tr>
<tr>
<td>TCP</td>
<td>viper:daytime</td>
<td>viper:0</td>
<td>LISTENING</td>
</tr>
<tr>
<td>TCP</td>
<td>viper:qotd</td>
<td>viper:0</td>
<td>LISTENING</td>
</tr>
<tr>
<td>TCP</td>
<td>viper:chargen</td>
<td>viper:0</td>
<td>LISTENING</td>
</tr>
<tr>
<td>TCP</td>
<td>viper:epmap</td>
<td>viper:0</td>
<td>LISTENING</td>
</tr>
<tr>
<td>TCP</td>
<td>viper:microsoft-ds</td>
<td>viper:0</td>
<td>LISTENING</td>
</tr>
<tr>
<td>TCP</td>
<td>viper:netbios-ssn</td>
<td>viper:0</td>
<td>LISTENING</td>
</tr>
<tr>
<td>TCP</td>
<td>viper:3713</td>
<td>imap.cs.tamu.edu:pop3</td>
<td>TIME_WAIT</td>
</tr>
<tr>
<td>TCP</td>
<td>viper:3717</td>
<td>google.com:http</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>TCP</td>
<td>viper:3718</td>
<td>google.com:http</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>TCP</td>
<td>viper:38209</td>
<td>dsl-113-77-11-99.comcast.net:12876</td>
<td>ESTABLISHED</td>
</tr>
</tbody>
</table>

P2P application (skype, BitTorrent) or possibly hacker
Windows Sockets 4

- Winsock requires initialization (unlike Unix)
  - This should be done before any other winsock calls
  - Once per program execution

```c
#include <Winsock2.h>
WSADATA wsaData;
WORD wVersionRequested = MAKEWORD(2,2);
if (WSAStartup(wVersionRequested, &wsaData) != 0)
{
    printf("WSAStartup error %d\n", WSAGetLastError());
    WSACleanup();
    return;
}
```
Agenda

• Windows sockets
  - Clients
• HTTP basics
• Multi-threading
  - Threads
  - Synchronization
  - Producer-consumer
Clients

- Steps to writing a TCP client:
  - Open a socket
  - Determine the IP address of the server in URL
  - Initiate connection with the server on correct port
  - Send request
  - Receive response
  - Close socket

- **Task 1**: open/close a TCP socket
  - Sockets are initially unbound (i.e., no port associated)

```c
SOCKET sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (sock == INVALID_SOCKET)
{
    printf("socket() error %d\n", WSAGetLastError());
    WSACleanup();
    return;
}
...
closesocket (sock);
```
• **Task 2**: determine the IP address of the server in URL
  • First assume the server is specified by an IP address
    - Convert to 4-byte int using `inet_addr("128.194.135.72")`
  • If this fails, then the server is given by its hostname
    - Use the domain name system (DNS)
      - DNS resolves **fully-qualified domain names** (FQDN) such as
        www.tamu.edu to their IP addresses (165.91.22.70)
  • DNS lookup performed through a system call
    - `struct hostent* remote = gethostbyname("www.tamu.edu");`
    - Returns 4-byte IP addresses inside the structure
  • **Reverse** DNS lookups map IPs to FQDN
    - Use function `gethostbyaddr()`
**Clients 3**

- **Task 3**: connect socket to server on correct port

```c
struct sockaddr_in server;
server.sin_family = AF_INET; // IPv4
server.sin_addr = ... // from inet_addr or gethostbyname
server.sin_port = ... // port #

if (connect (sock, (struct sockaddr*) &server,
            sizeof(struct sockaddr_in)) == SOCKET_ERROR)
{
    printf ("Connection error: %d\n", WSAGetLastError());
    return;
}
```

- Main caveat is that all numbers must be in network byte order (MSB first)
  - Forward (host-to-network): htons(), htonl()
  - Reverse (network-to-host): ntohs(), ntohl()
- `inet_addr` and `gethostbyname` internally perform this, so usually only port # needs explicit conversion
• **Task 4:** send request conforming to correct protocol

```c
char *sendBuf = new char [requestLength];
// place request into buf
if (send (sock, sendBuf, requestLen, 0) == SOCKET_ERROR)
{
    printf ("Send error: %d\n", WSAGetLastError());
    return;
}
```

• **Task 5:** receive response into recvBuf
  - Data arrives in chunks from function recv(), needs to be appended to a character buffer
  - Size of message and each chunk is unknown a-priori
    - Recv() must be called repeatedly until it returns 0 bytes
    - Use a pointer that moves along receive buffer
    - Buffer starts from 4-8 KB and is resized dynamically to accommodate longer messages
Agenda

- Windows Sockets
  - Clients
- HTTP basics
- Multi-threading
  - Threads
  - Synchronization
  - Producer-consumer
HTTP Basics

• General URL format:
  - Optional elements shown in square brackets
    
    scheme://[user:pass@]host[:port][/path][?query][#fragment]
  
• No need to parse username/password in this homework, but you must strip off the fragment and extract the port number from the host
  - If the path is not present, must use root “/” in its place
• HTTP request is [/path][?query]:

<table>
<thead>
<tr>
<th>URL</th>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://google.com">http://google.com</a></td>
<td>/</td>
</tr>
<tr>
<td></td>
<td>/</td>
</tr>
<tr>
<td><a href="http://tamu.edu?test=1/blah">http://tamu.edu?test=1/blah</a></td>
<td>/?test=1/blah</td>
</tr>
<tr>
<td><a href="http://tamu.edu?tes:t=1/blah">http://tamu.edu?tes:t=1/blah</a></td>
<td>/?tes:t=1/blah</td>
</tr>
</tbody>
</table>

1) Find # using strchr() and truncate
2) Find first /, :, ? using strchr()
3) Design an algorithm to use these 3 pointers to determine host, port, request
HTTP Basics 2

- HTTP request
  - Begins with the **method** line, followed by (field: value) pairs
  - Ends with an empty line

- Methods
  - GET and HEAD, same syntax

- HTTP responses
  - Status line begins with HTTP/
  - Status codes are 3-digit integers

**minimal request**

```plaintext
GET /courses/ HTTP/1.0
Host: irl.cs.tamu.edu
Connection: close

GET /courses/ HTTP/1.0
Host: irl.cs.tamu.edu
Connection: close
```

**HTTP response**

```plaintext
HTTP/1.0 200 OK
Cache-Control: private
Content-Type: text/html
Server: Microsoft-IIS/7.0
X-Powered-By: ASP.NET
MicrosoftOfficeWebServer: 5.0_Pub
MS-Author-Via: MS-FP/4.0
Date: Thu, 17 Jan 2013 09:22:34 GMT
Connection: close
Content-Length: 16367

<html>
<head>
<meta http-equiv="Content-Language" content="en-us">
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
...```
Agenda

• Windows Sockets
  - Clients
• HTTP basics
• Multi-threading
  - Threads
  - Synchronization
  - Producer-consumer
Multi-Threading

- Threads execute concurrently as part of a process
- Benefits:
  - Allows for parallelism in a multiprocessor/multicore system
  - If a blocking call is made in one thread, other threads can continue executing
- Issues:
  - Memory is shared between threads, concurrent access requires proper synchronization
  - Order of execution of threads is non-deterministic
- Homework note: pass shared parameters to threads using a dedicated class instead of using global variables (see 463-sample.zip on course site)
Multi-Threading 2

• Reasons for using multiple threads in hw #1
  – Web servers respond slowly (1-10 seconds/request)
  – While a thread is suspended waiting for connect() and recv(),
    other threads should be allowed to work

• Multiple threads achieve significant speed-up
  – You could run thousands of threads, but limit your testing to
    10 until you know it works correctly

• Common synchronization mechanisms
  – **Mutex** (mutual exclusion): allows only one thread access to
    critical section; others must wait
  – **Semaphore**: allows up to N concurrent threads
  – **Event**: binary (i.e., ON or OFF) signal
Multi-Threading 3

- Mutex usage
  - Any data structure or resource (e.g., screen or disk) modified by parallel threads needs to be protected
  - If not, inconsistencies (data corruption) may result

```c
HANDLE m = CreateMutex (...);
DWORD ret = WaitForSingleObject(m, INFINITE); // lock
if (ret != WAIT_OBJECT_0)
{
    printf("Wait returned %d, error %d waiting on mutex\n", ret, GetLastError());
    exit (-1);
}
// critical section here ...

if (ReleaseMutex (m) == FALSE) // unlock
{
    printf("Error %d releasing mutex\n", GetLastError());
    exit (-1);
}
```

- Read MSDN for additional details
Multi-Threading 4

- A semaphore has a numerical value $s$ attached to it
- Wait on semaphore (operation P)
  - If $s = 0$, the semaphore suspends the calling thread
  - If $s > 0$, the thread is allowed access and $s$ is set to $s - 1$
- Release semaphore (operation V)
  - If threads are waiting, unblock one of them and run it
  - Otherwise, increment $s = s + 1$

```c
HANDLE sema = CreateSemaphore (...);
DWORD ret = WaitForSingleObject(sema, INFINITE); // wait
if (ret != WAIT_OBJECT_0)
    // report error
// critical section...
if (ReleaseSemaphore (sema, ...) == FALSE) // release
    // report error
```
Multi-Threading

- Unbounded producer-consumer

```c
Producer()
{
    while (true)
    {
        // make items x[1],..., x[N]
        m.Lock ();
        for (i = 0; i < N; i++)
            Q.push (x[i]);
        m.Unlock ();
        // Windows allows batch
        // release
        sema.Release (N);
    }
}
```

```c
Consumer()
{
    while (true)
    {
        sema.Wait ();
        m.Lock ();
        // no need to check Q.size
        x = Q.pop ();
        m.Unlock ();
        // consume x outside
        // the critical section
    }
}
```

- Bounded PC requires another semaphore that counts the number of empty slots in the queue
  - Hw #1 $\rightarrow$ unbounded PC
  - Hw #3 $\rightarrow$ bounded PC