Lecture

Review of Last Lecture

- RFCs
- HTTP protocol
- MIME types
- Server Architecture
- Threads

Server Architecture

- Dialog-based GUI application
- Most of the processing is at back-end
- Running on TCP port 5432 decimal

HTTP Web Server Application

Initialise Windows Sockets Library

```
if(WSAStartup(MAKEWORD(1,1), &wsaData))
{
   … … …
   return 1;
}
```

HTTP Web Server Application

Get machine's hostname and IP address

```
gethostname(hostName, sizeof(hostName));
ptrHostEnt = gethostbyname(hostName);
```

Fill the socket address with appropriate values

```
serverSocketAddress.sin_family = AF_INET;
serverSocketAddress.sin_port = htons(SERVER_PORT);
… … …
memcpy(&serverSocketAddress.sin_addr.S_un.S_addr,
 ptrHostEnt->h_addr_list[0], sizeof(unsigned long));
```

HTTP Web Server Application

Create the server socket

```
serverSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if(serverSocket == INVALID_SOCKET)
{
   … … …
   WSACleanup();
   return 1;
}
```

Bind the socket

```
if(bind(serverSocket,
        (struct sockaddr *)&serverSocketAddress,
        sizeof(serverSocketAddress)))
{
   … … …
   WSACleanup();
   return 1;
}
```

HTTP Web Server Application

Put the socket in listening mode

```
if(listen(serverSocket, MAX_PENDING_CONNECTIONS))
{
   … … …
   WSACleanup();
   return 1;
}
```

Here is the time to accept client connections

HTTP Web Server Application

Create a thread that will call accept() in a loop to accept multiple client connections

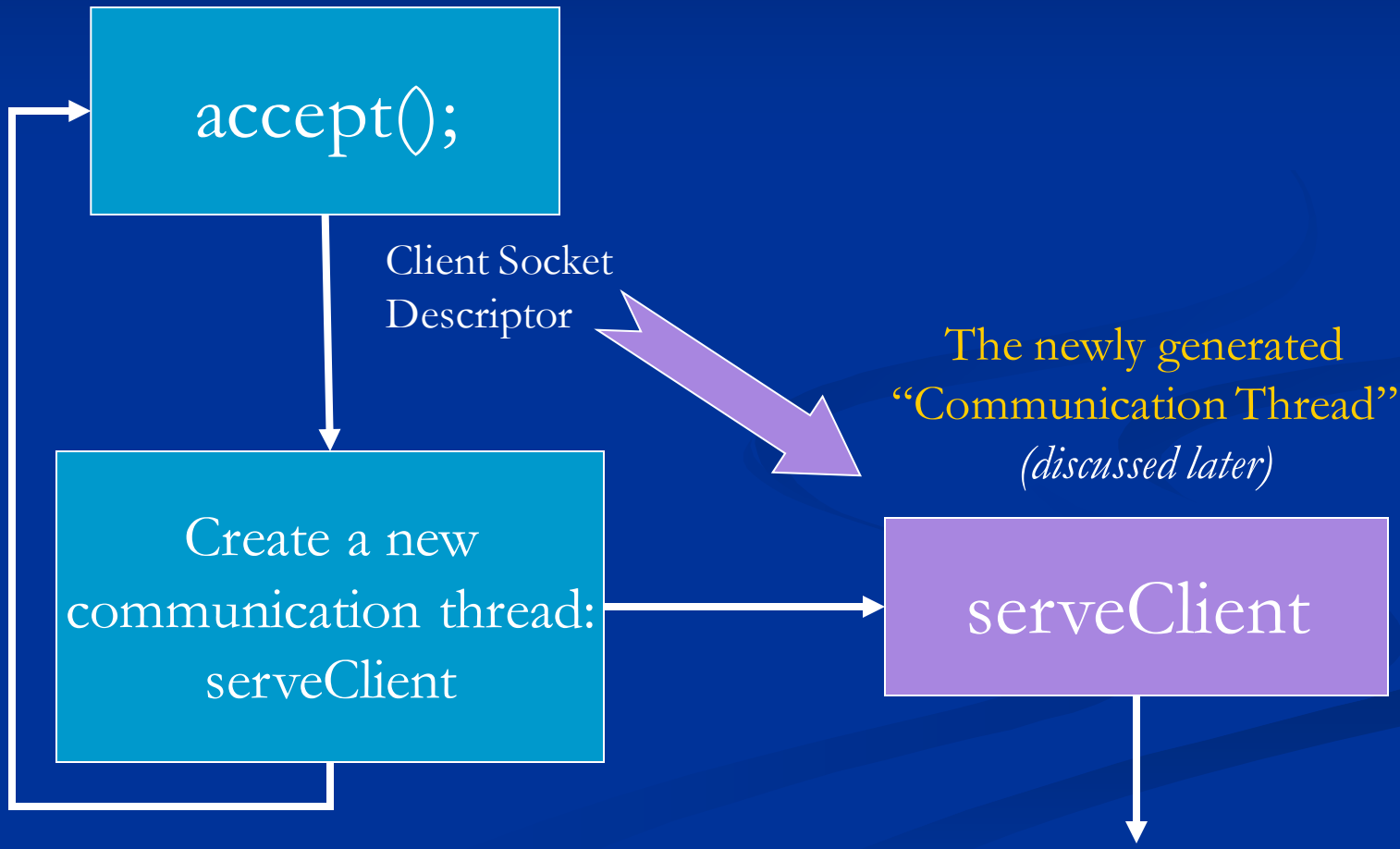
```
hAcceptingThread = CreateThread(
      NULL,
      0,
      (LPTHREAD_START_ROUTINE)
      acceptClientConnections,
      NULL,
      CREATE_SUSPENDED,
      &dwAcceptingThread);
```

HTTP Web Server Application

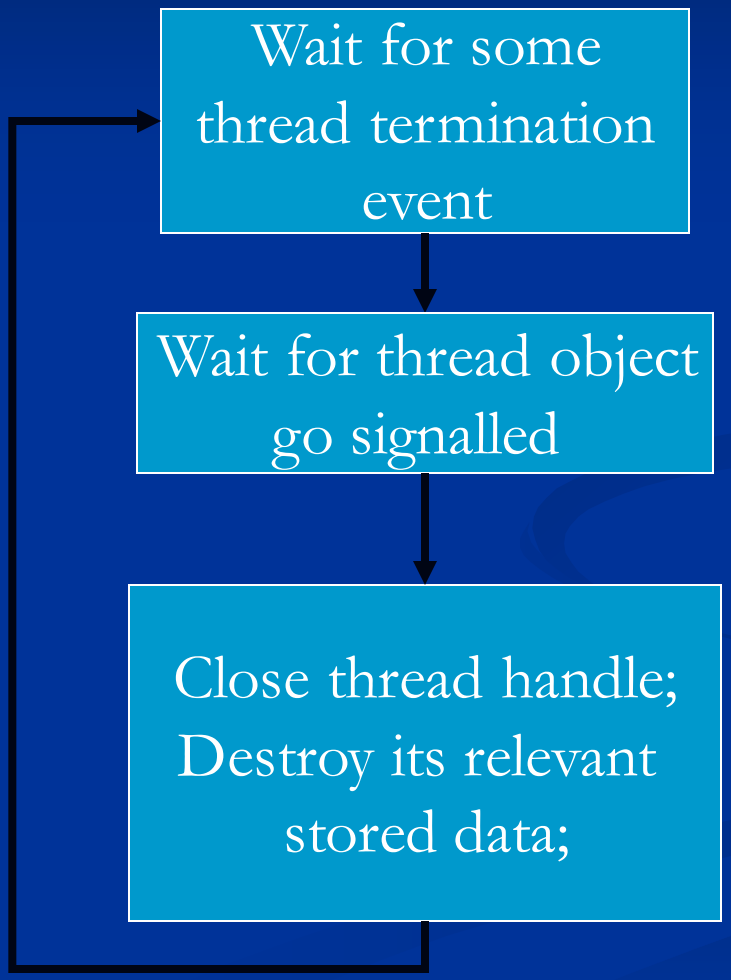
Create a thread to do termination house-keeping when some communication thread terminates

```
hTerminatingThread = CreateThread(
  NULL,
  0,
  (LPTHREAD_START_ROUTINE)
  terminateCommunicationThreads,
  NULL,
  CREATE_SUSPENDED,
  &dwTerminatingTThread);
```

`acceptClientConnections` thread routine



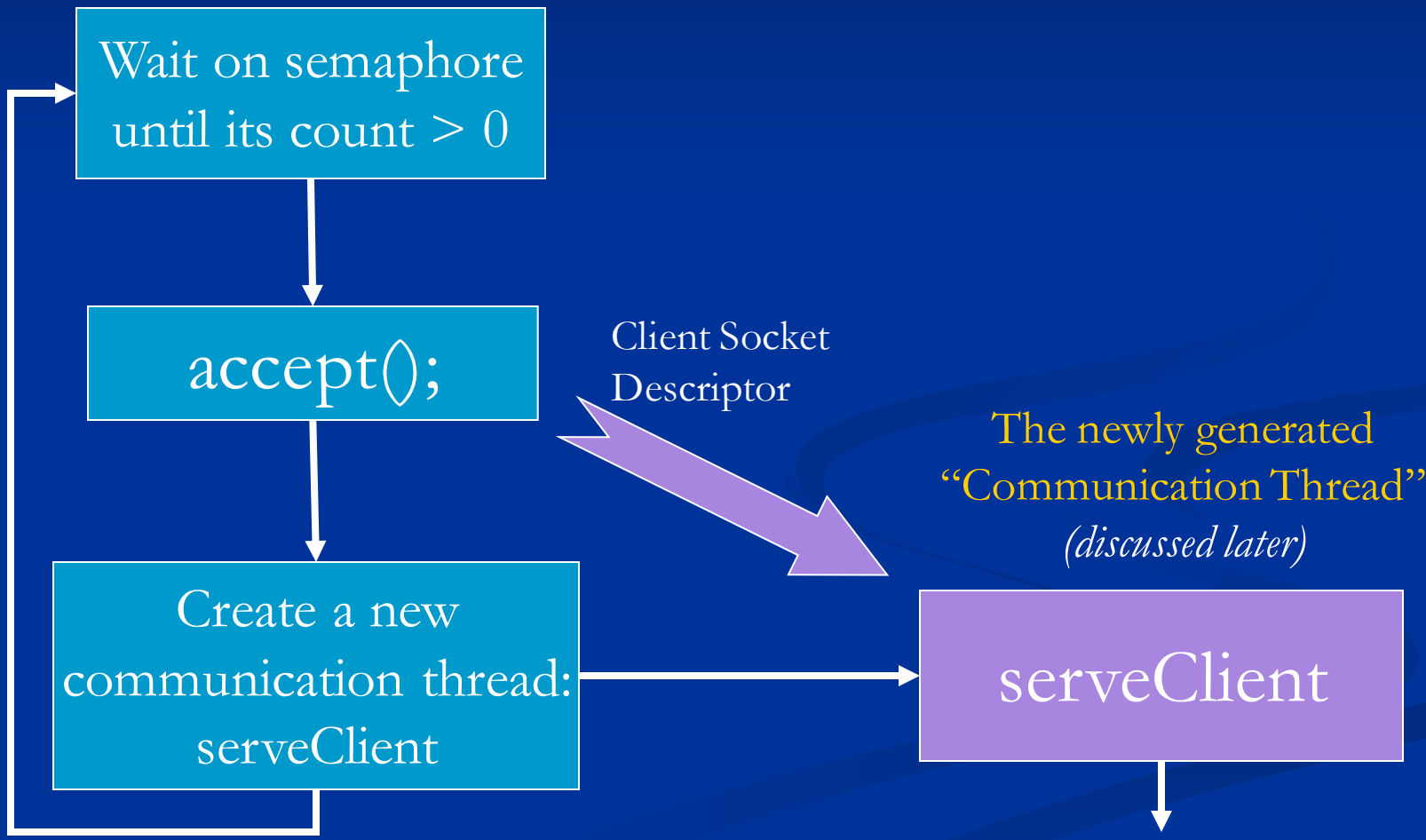
`terminateCommunicationThreads` thread routine



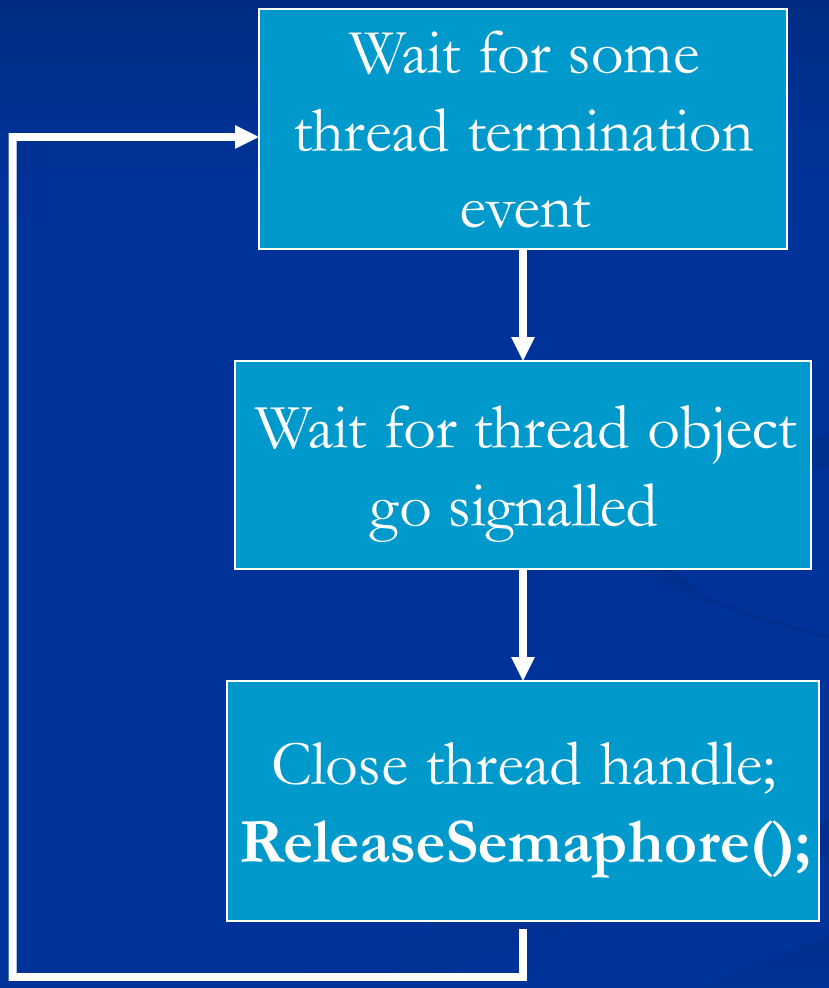
Application Variables

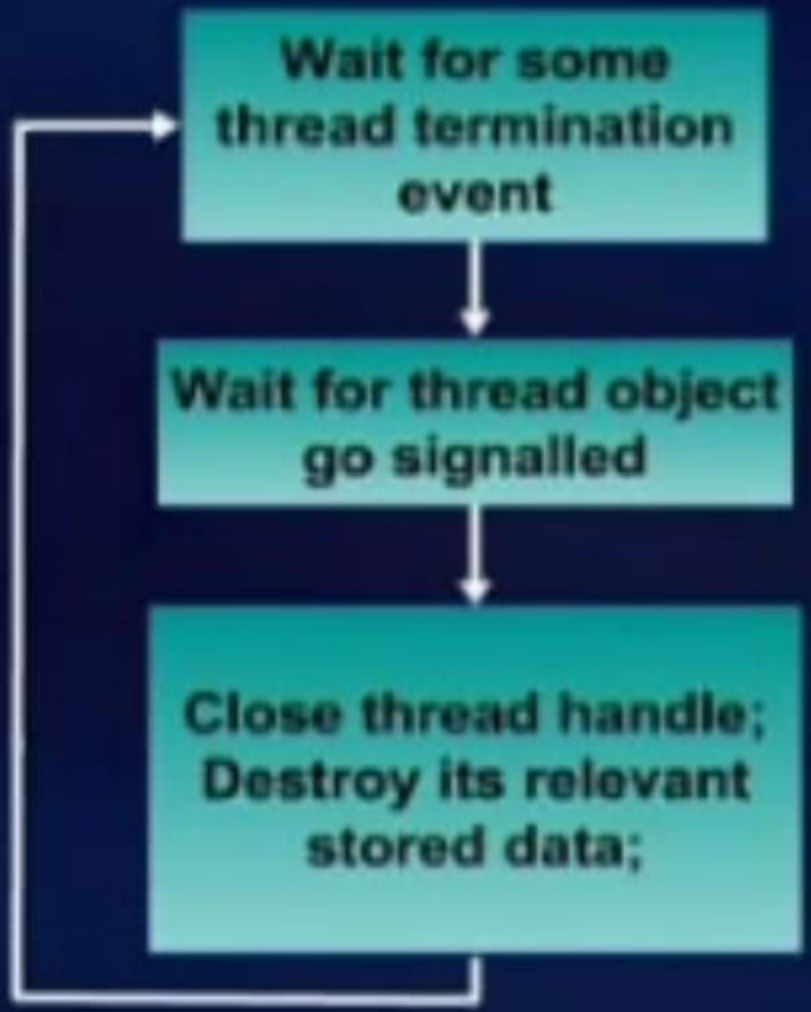
```
#define MAX_CLIENTS 5

SOCKET clientSockets[MAX_CLIENTS];

HANDLE hCommunicationThreads[MAX_CLIENTS];
DWORD dwCommunicationThreads[MAX_CLIENTS];

HANDLE hAcceptingThread;
DWORD dwAcceptingThread;

HANDLE hTerminatingThread;
DWORD dwTerminatingTThread;
```

`acceptClientConnections` thread routine

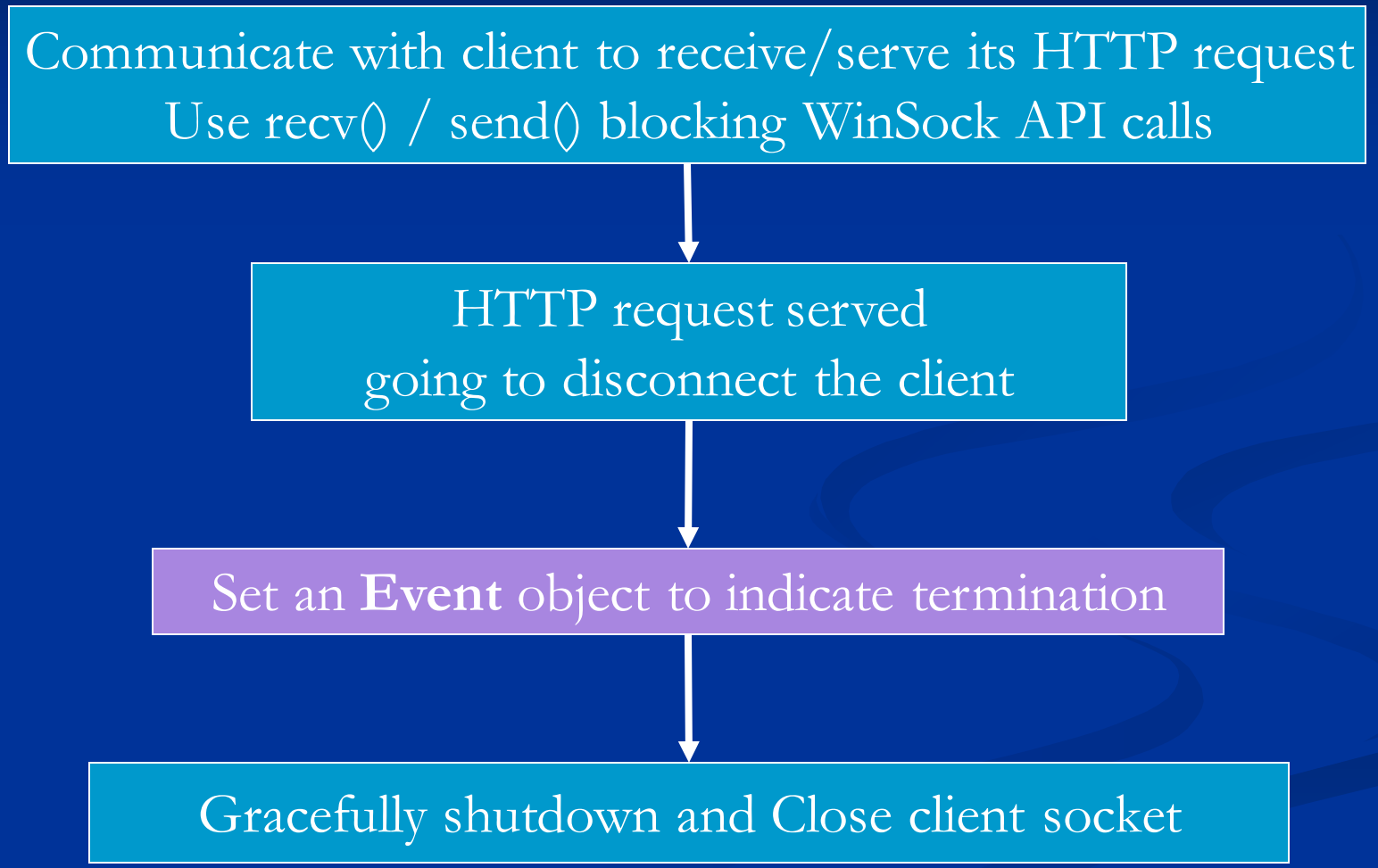


`terminateCommunicationThreads` thread routine





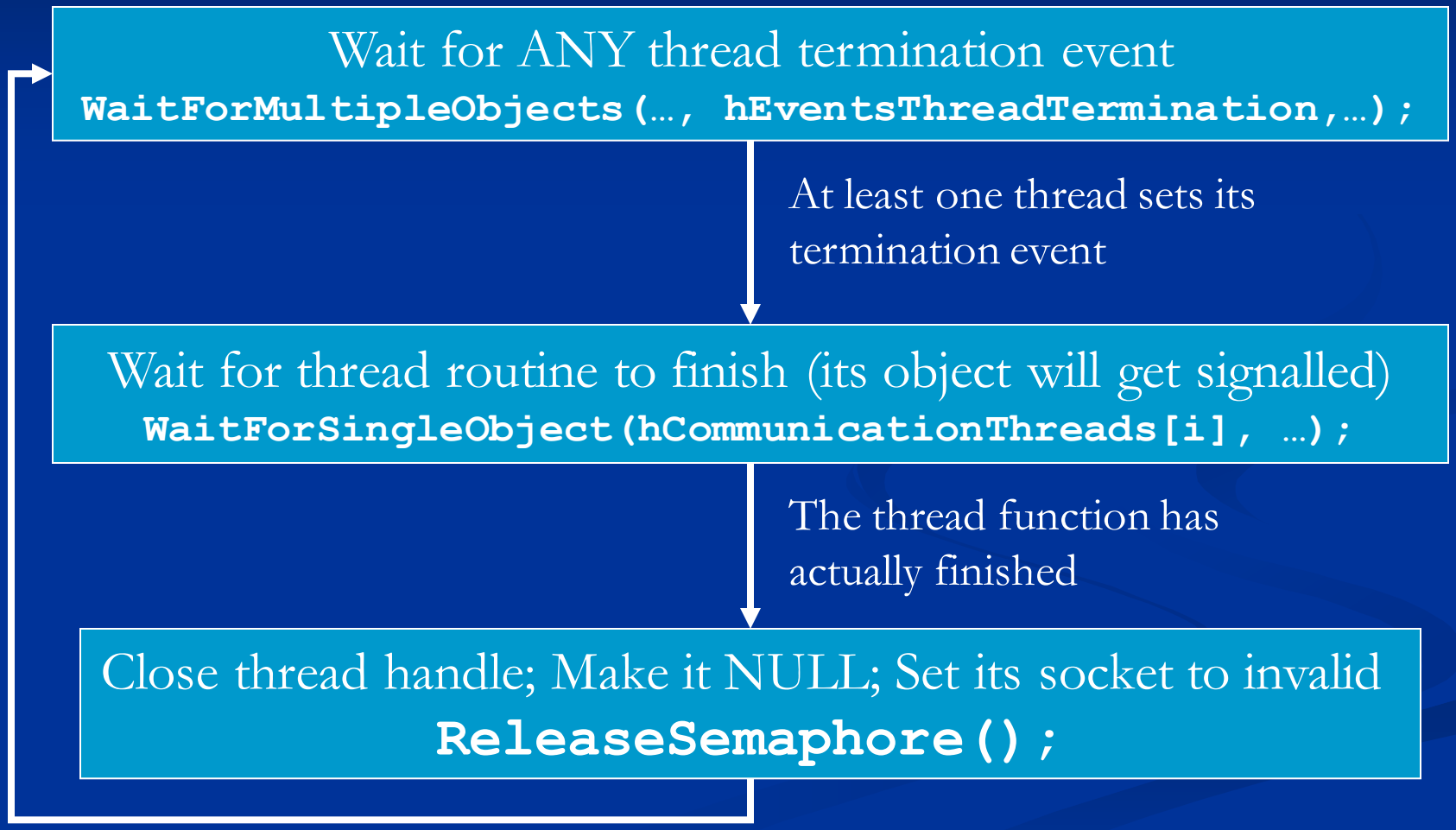
`serveClient` Communication Thread routine



Application Variables

```
#define MAX_CLIENTS   5

HANDLE hEventsThreadTermination[MAX_CLIENTS];
```

`terminateCommunicationThreads` thread routine



Thread Procedures Summary

`acceptClientConnections`

- to accept client connection

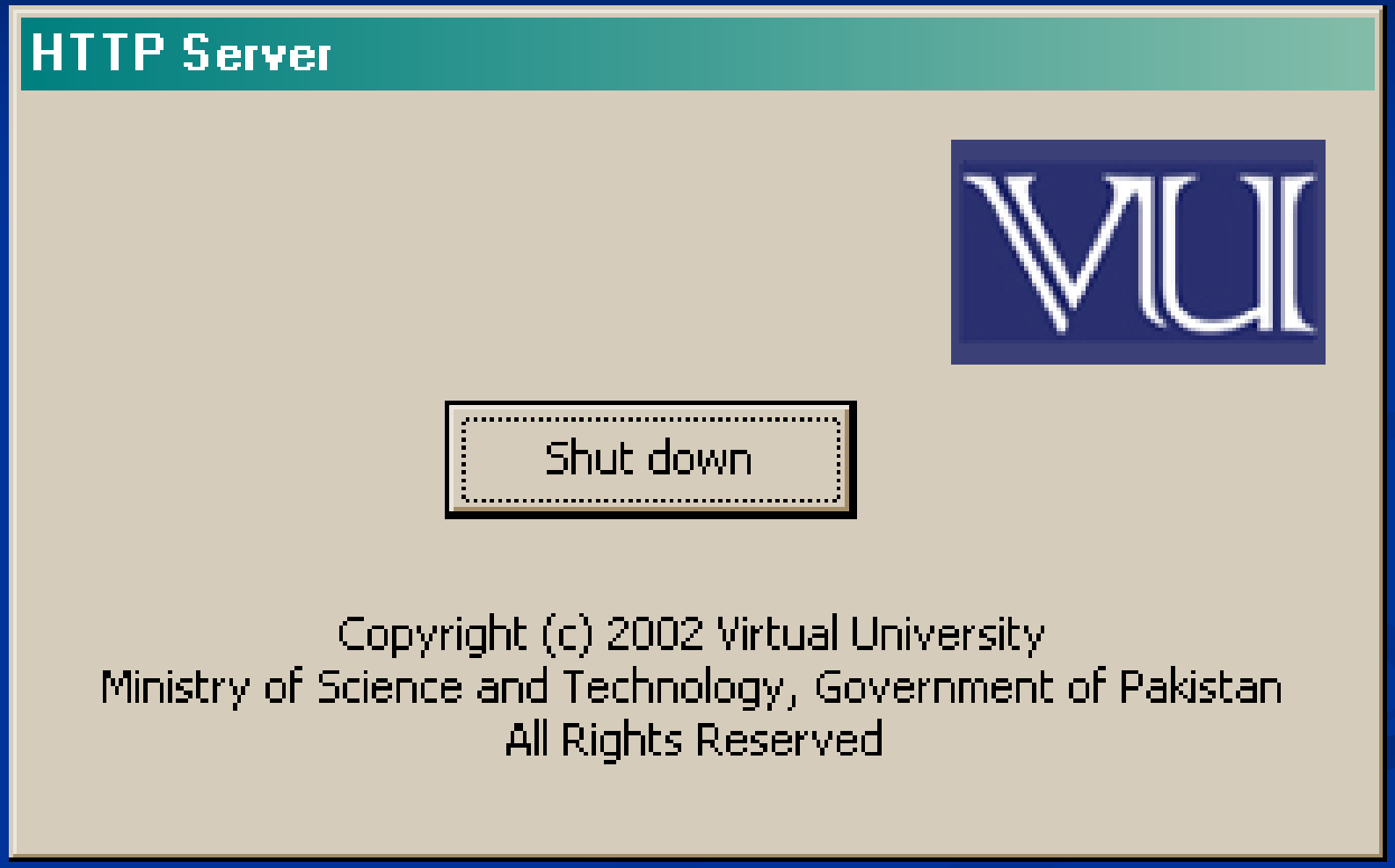
`terminateCommunicationThreads`

- to do housekeeping when communication threads terminate

`serveClient`

- to do actual communication to receive and serve an HTTP request

User Interface



Variable Initialisation

```
for(i=0; i<MAX_CLIENTS; ++i)
{
    clientSockets[i] = INVALID_SOCKET;

    hCommunicationThreads[i] = NULL;
    dwCommunicationThreads[i] = 0;

    hEventsThreadTermination[i] = NULL;
}
```

Initialise WinSock Library

```
if(WSAStartup(MAKEWORD(1,1), &wsaData))
{
  MessageBox(NULL,
      "Error initialising sockets library.",
      "WinSock Error",
      MB_OK | MB_ICONSTOP);

      return 1;
}
```

Win32 Error Codes

```
int  WSAGetLastError(void);
```

- get error code for the last unsuccessful Windows Sockets operation

```
DWORD GetLastError(VOID);
```

- retrieve calling threads last-error code

HTTP Web Server Application

Get machine's hostname and IP address

```
gethostname(hostName, sizeof(hostName));
ptrHostEnt = gethostbyname(hostName);
```

Fill the socket address with appropriate values

```
serverSocketAddress.sin_family = AF_INET;
serverSocketAddress.sin_port = htons(SERVER_PORT);
… … …
memcpy(&serverSocketAddress.sin_addr.S_un.S_addr,
 ptrHostEnt->h_addr_list[0], sizeof(unsigned long));
```

HTTP Web Server Application

Create the server socket

```
serverSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if(serverSocket == INVALID_SOCKET)
{
   … … …
   WSACleanup();
   return 1;
}
```

Bind the socket

```
if(bind(serverSocket,
        (struct sockaddr *)&serverSocketAddress,
        sizeof(serverSocketAddress)))
{
   … … …
   WSACleanup();
   return 1;
}
```

HTTP Web Server Application

Put the socket in listening mode

```
if(listen(serverSocket, MAX_PENDING_CONNECTIONS))
{
   … … …
   WSACleanup();
   return 1;
}
```

Here is the time to accept client connections

Limiting Maximum Concurrent Connections

Create an unnamed semaphore object with MAX_CLIENTS as initial/maximum count

```
hSemaphoreMaxClients =
        CreateSemaphore(NULL,
                        MAX_CLIENTS,
                        MAX_CLIENTS,
                        NULL);
```

"I am dying…", the thread said

create an array of non-signalled event objects

```
for(i=0; i<MAX_CLIENTS; i++)

  hEventsThreadTermination[i] =
    CreateEvent(NULL, FALSE, FALSE, NULL);
```

HTTP Web Server Application

Create the connection-accepting thread

```
hAcceptingThread = CreateThread(
       NULL,
       0,
       (LPTHREAD_START_ROUTINE)
                    acceptClientConnections,
       NULL,
       CREATE_SUSPENDED,
       &dwAcceptingThread);
```

Create the termination house-keeping thread

```
hTerminatingThread = CreateThread(… … …);
```

Display the dialog

```
DialogBox(…, …, …, mainDialogProc);
```

mainDialogProc dialog procedure

Resume the threads

```
case WM_INITDIALOG:
      ResumeThread(hAcceptingThread);
      ResumeThread(hTerminatingThread);
      return TRUE;
      break;
```

Handling the server shut-down button

```
case IDC_BUTTON_SHUTDOWN:
```

Perform any shut-down tasks that my be necessary

```
      EndDialog(hDlg, 0);
      break;
```

acceptClientConnections thread routine

Start of the loop to accept client connections

Wait for semaphore count to go non-zero

```
dwWaitResult = WaitForSingleObject(
    hSemaphoreMaxClients,
    INFINITE);

switch(dwWaitResult)
{
case WAIT_OBJECT_0:
```

We can accept more connections here because semaphore object is signalled

```
    clientSocket = accept(… … …);
```

acceptClientConnections thread routine

```
clientSocket = accept(… … …);
```

Connection accepted! Look for the first empty slot to save the new socket descriptor

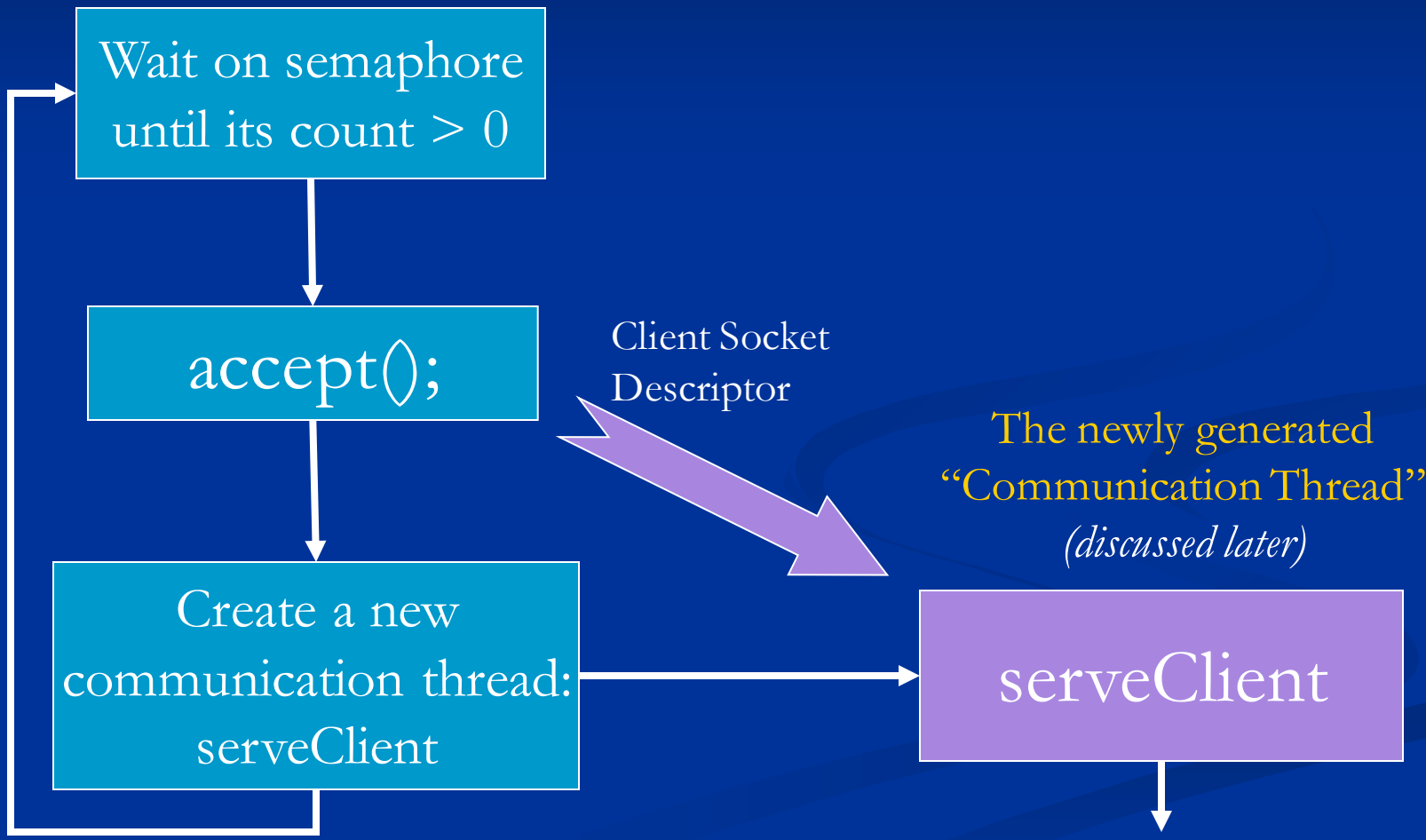
```
for(i=0; i<MAX_CLIENTS; i++)
{
  if(clientSockets[i] == INVALID_SOCKET)
     break;
}

nextClientIndex = i;

clientSockets[nextClientIndex]=clientSocket;
```

acceptClientConnections thread routine

nextClientIndex is used as an index in ALL arrays to store information relevant to this new client connection

```
clientSockets[nextClientIndex]=clientSocket;

hCommunicationThreads[nextClientIndex] =
  CreateThread(…, …,
    serveClient,                    thread procedure
    (LPVOID)nextClientIndex,        thread parameter
    CREATE_SUSPENDED,
    …);
```

`acceptClientConnections` thread routine



serveClient thread routine

Index for this client in all arrays is passed to this thread routine

```
DWORD WINAPI serveClient(LPVOID clientNumber)
{
  char msg[2046] = "";
```

Receiving an HTTP request from browser

```
  recv(
      clientSockets[(UINT)clientNumber],
      msg,
      2046,
      0 );
```

Available HTTP request data will be received in this buffer

acceptClientConnections thread routine

nextClientIndex is used as an index in ALL arrays to store information relevant to this new client connection

```
clientSockets[nextClientIndex]=clientSocket;

hCommunicationThreads[nextClientIndex] =
  CreateThread(…, …,
    serveClient,
    (LPVOID)nextClientIndex,
    CREATE_SUSPENDED,
    …);
```

thread parameter

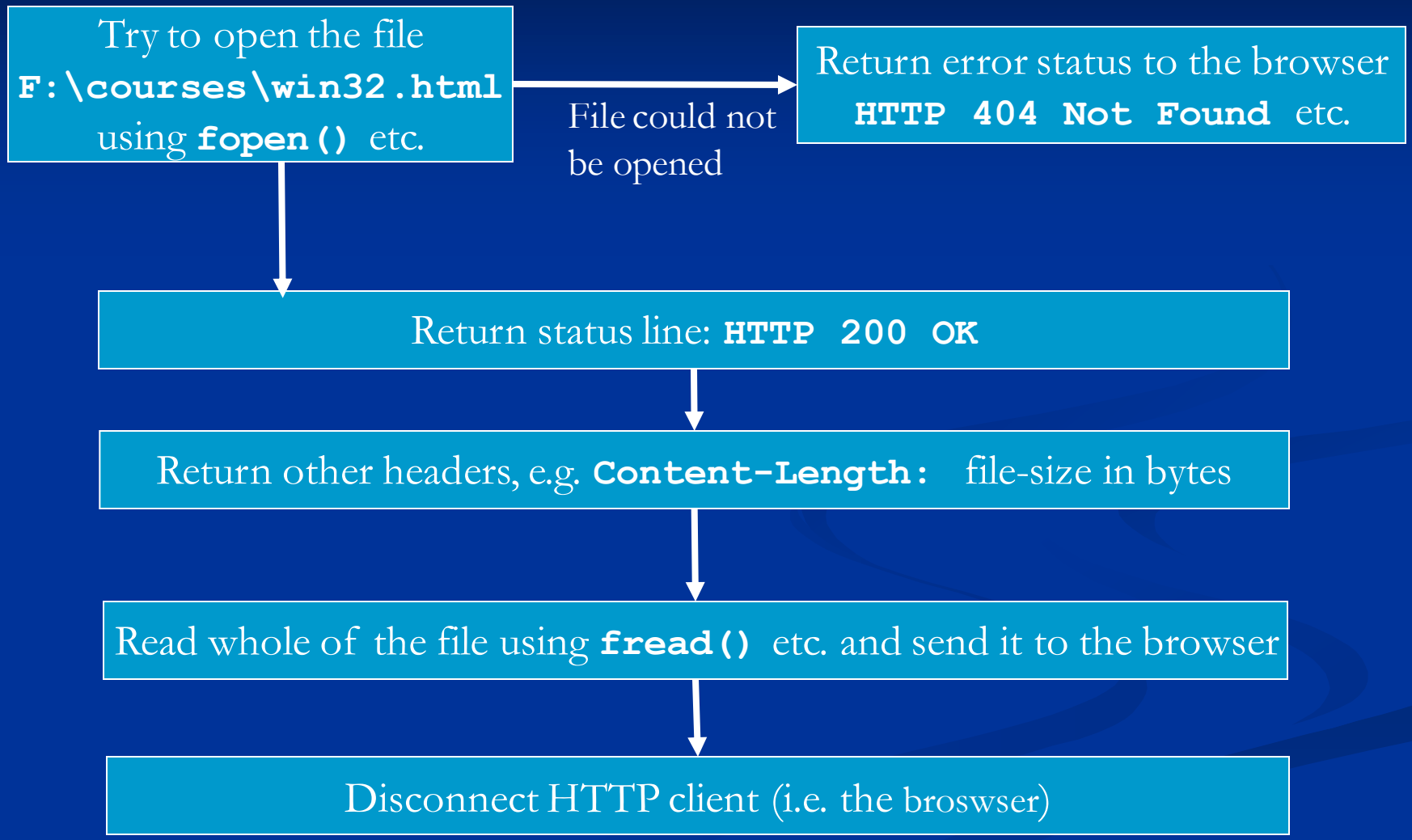
Sample Request

- Request parsing: understanding what the client has demanded

```
GET /courses/win32.html HTTP/1.0
```

- Assume `F:\` is your server's home directory, and \courses\is not a virtual directory, server should return the file `F:\courses\win32.html`

Server returns the requested file



HTTP Redirection

- Redirecting the client irrespective of the HTTP request!

The string in the #define directive is assumed to be on a single line

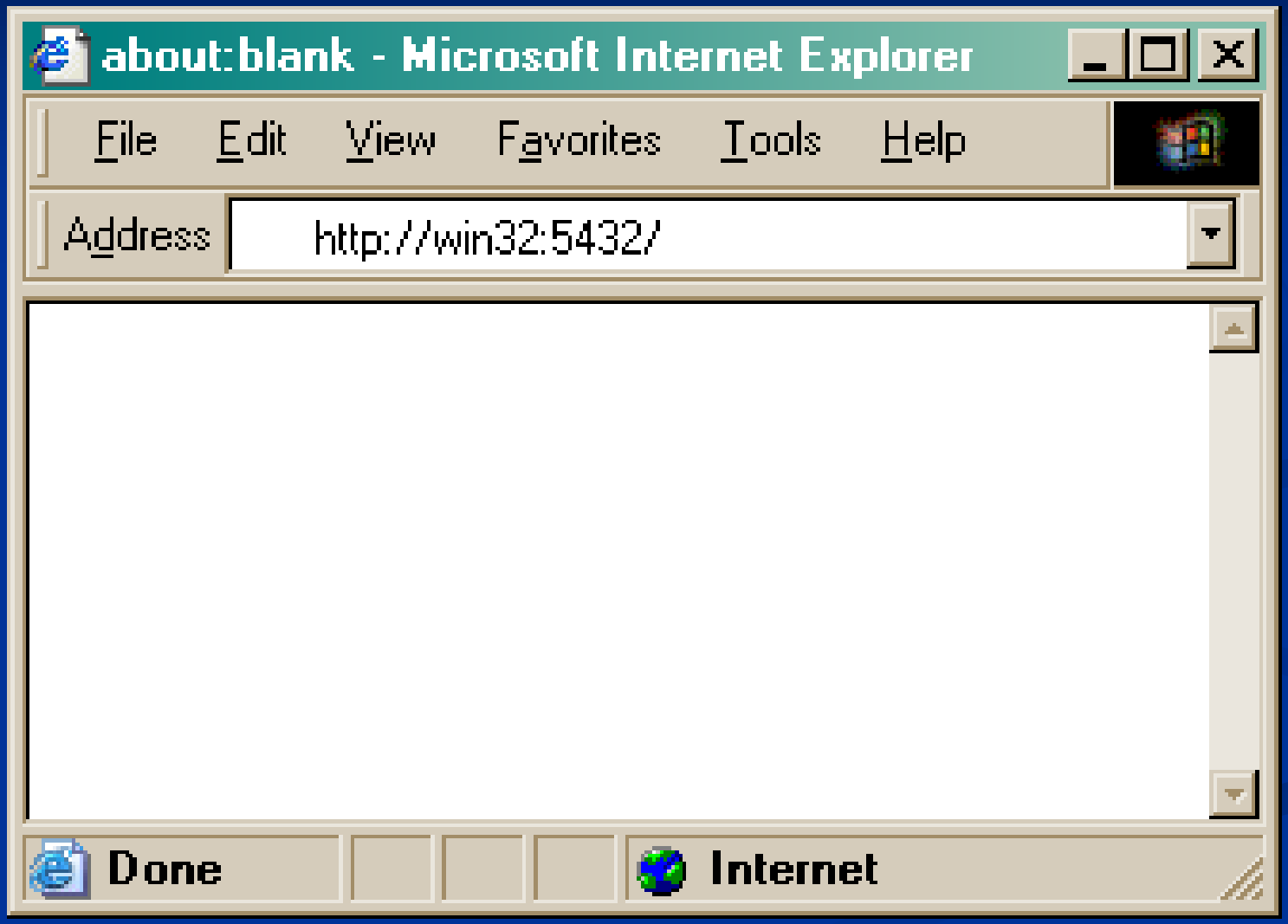
```
#define RESPONSE
   "HTTP/1.1 302 Object Moved\r\n
   Location: http://www.vu.edu.pk\r\n\r\n"
```

Status line

Sending the hard-coded HTTP response back to browser

```
send(clientSockets[(UINT)clientNumber],
   RESPONSE,
   sizeof(RESPONSE),
   0);
```

Sending a request to our Web Server



Using Port Numbers

- There is no compulsion to build all HTTP Web Servers to run on port 80
- These are 'suggested' port numbers for a Win32 developer
- Standard servers do run on port 80
- Our HTTP Web Server may also need to run on port 80 if put it to public use

Returning an HTML document

The string in the #define directive is assumed to be on a single line

```
#define RESPONSE
   "HTTP/1.0 200 OK\r\n
   Content-type: text/html\r\n
   Content-length: 1325\r\n\r\n"
```

File **a.htm** is **1325** bytes long

Send the hard-coded HTTP status and headers

```
send(clientSockets[(UINT)clientNumber], RESPONSE,
   sizeof(RESPONSE), 0);
```

Now send the whole file using character I/O of standard C runtime

```
ch = fgetc(fptr);
while(!feof(fptr))  {
 send(clientSockets[(UINT)clientNumber], &ch, 1, 0);
 ch = fgetc(fptr);
}
```

A Flawed Web Server

- Fixed sized arrays waste memory and lack run-time flexibility
- One event per thread to signify termination: WaitForMultipleObjects() can not wait on more than a certain number of objects e.g. 64 on x86 under NT.

Dynamic Web Content

- Server blindly dumps HTML files to the clients. This is 'static content'.
- If server reads the file and modifies its output e.g.
  `%%time%%` replaced with current system time
  Every 2 clients connected at different instants of time will receive **different** content.
  This is 'dynamic content'.
- `%%time%%` may be called a **tag**

Dynamic Web Content

- Microsoft Active Server Pages
- Macromedia ColdFusion
- Tags are not sent to the client. These are processed by the server and the resulting output is sent to the browser.

CGI

- Common Gateway Interface
- Win32 EXE executable is executed at the server
- All browser request data is available at `stdin` (read using `scanf()` etc.)
- All output sent to `stdout` (output using `printf` etc.) is sent to the browser instead of the server screen.