BÀI 3.
CÁC CHẾ ĐỘ VÀO RA TRÊN WINSOCK

1

1

Nội dung

- Chế độ vào ra blocking và non-blocking
- Kỹ thuật đa luồng
- Kỹ thuật thăm dò
- Kỹ thuật vào ra theo thông báo
- Kỹ thuật vào ra theo sự kiện
- Kỹ thuật Overlapped

2

2

1. CÁC CHẾ ĐỘ VÀO RA

3

3

Xem lại TCP Echo Server

- Nhận xét: Chỉ làm việc được với 1 client
- Hàm recv() chỉ trả về khi nhận được dữ liệu trên socket

→ tiến trình bị chặn, không thể thực hiện lời gọi hàm *accept()* để xử lý kết nối của client khác

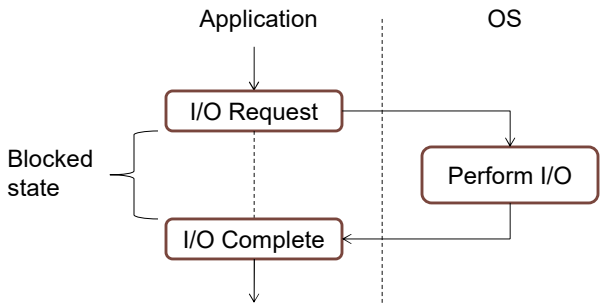
```
//Step 5: Communicate with client
sockaddr_in clientAddr;
char buff[1024];
int ret, clientAddrLen = sizeof(clientAddr);

while(1){
   SOCKET connSock;
   //accept request
   connSock = accept(listenSock, (sockaddr *) & clientAddr,
                              &clientAddrLen);
   //receive message from client
   ret = recv(connSock, buff, 1024, 0);

//...
```

4

4

Các chế độ hoạt động trên WinSock

- Chế độ chặn dừng (blocking), hoặc đồng bộ (synchronous)
  - Các hàm vào ra sẽ chặn, tạm dừng luồng thực thi đến khi thao tác vào ra hoàn tất (các hàm vào ra sẽ không trả về cho đến khi thao tác hoàn tất).
  - Là chế độ mặc định trên SOCKET: connect(), accept(), send()…
  - Hạn chế sử dụng các hàm ở chế độ chặn dừng trên giao diện



5

5

Các chế độ hoạt động trên WinSock

- Chế độ không chặn dừng(non-blocking), hoặc bất đồng bộ(asynchronous)
  - Các thao tác vào ra trên SOCKET sẽ trở về nơi gọi ngay lập tức và tiếp tục thực thi luồng. Kết quả của thao tác vào ra sẽ được thông báo cho chương trình dưới một cơ chế đồng bộ nào đó.
  - Các hàm vào ra bất đồng bộ sẽ trả về mã lỗi WSAEWOULDBLOCK nếu thao tác đó không thể hoàn tất ngay và mất thời gian đáng kể(chấp nhận kết nối, nhận dữ liệu, gửi dữ liệu...)
  - Socket cần chuyển sang chế độ này bằng hàm *ioctlsocket()*

```
int ioctlsocket (
   SOCKET s,       //[IN]socket được thiết lập chế độ
   long cmd,       //[IN]chế độ điều khiển vào ra
   u_long *argp,   //[IN/OUT]thiết lập giá trị cho cmd
);
```

6

6

Ví dụ - Chế độ non-blocking

```
SOCKET          s;
unsigned long ul = 1;
int             nRet;

s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
//Set socket into non-blocking mode
nRet = ioctlsocket(s, FIONBIO, (unsigned long *) &ul);
if (nRet == SOCKET_ERROR)
{
    //Failed to put the socket into non-blocking mode
}
```

7

7

Sử dụng chế độ non-blocking

```
//Step 2: Construct socket
listenSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
unsigned long ul = 1;
if (ioctlsocket(listenSock, FIONBIO, (unsigned long *)&ul)) {
   printf("Error! Cannot change to non-blocking mode.");
   return 0;
}
//...
//Step 5: Communicate with client
SOCKET client[MAX_CLIENT];
int numClient = 0;
while(1){
   SOCKET connSock;
   //accept request
   connSock = accept();
   if (connSock != SOCKET_ERROR){
       client[numClient] = connSock;
       numClient++
   }
```

8

8

Sử dụng chế độ non-blocking

```
   for (i = 0; i < numClient; i++){
       //receive request from client
       ret = recv(client[i], buff, BUFF_SIZE, 0);
       if (ret == SOCKET_ERROR)
          if (WSAGetLastError() != WSAEWOULDBLOCK)
              closesocket(client[i]);
       else{
          //handle request
          //...
          //send response to client
          ret = send(client[i], buff, strlen(buff), 0);
          if (ret == SOCKET_ERROR){
              errorCode = WSAGetLastError();
              if (errorCode != WSAEWOULDBLOCK)
                 closesocket(client[i]);
          }
       }
   } //end for
} //end while
```

9

9

Một số trường hợp trả về *WSAEWOULDBLOCK*

Hàm được gọi	Nguyên nhân gây lỗi
WSAAccept() hoặc accept()	Không có kết nối tới server
closesocket()	Socket được thiết lập tùy chọn SO_LINGER và bộ đếm timeout khác 0
WSAConnect() hoặc connect()	Quá trình thiết lập kết nối đã khởi tạo nhưng chưa hoàn thành
WSARecv(), recv(), WSARecvFrom(), recvfrom()	Không có dữ liệu để nhận
WSASend(), send(), WSASendTo(), sendto()	Không đủ khoảng trống trên bộ đệm để gửi dữ liệu

10

10

Các kỹ thuật vào ra trên WinSock

- Kỹ thuật đa luồng
- Kỹ thuật lựa chọn
- Kỹ thuật vào ra bất đồng bộ
- Kỹ thuật vào ra theo sự kiện
- Kỹ thuật chồng chập (overlapped)

11

11

2. KỸ THUẬT ĐA LUỒNG

12

12

Kỹ thuật đa luồng

- Giải quyết vấn đề các hàm bị chặn dừng bằng cách tạo ra các luồng chuyên biệt

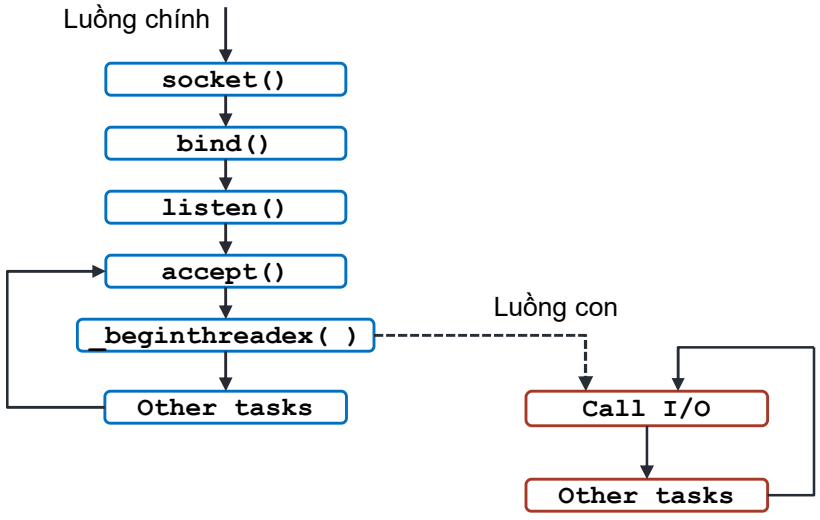
```
unsigned long _beginthreadex(
   void *security,
   unsigned stack_size,
   unsigned ( __stdcall *start_address )( void * ),
   void *arglist,
   unsigned initflag,
   unsigned *thrdaddr);
```

- Trong đó:
  - *security*: [IN] trỏ tới cấu trúc xác định quyền truy cập
  - *stack_size*: [IN] kích thước khởi tạo stack cho luồng mới
  - start_address: [IN] con trỏ tới hàm được thực thi trong luồng
  - arglist: [IN] con trỏ tới tham số truyền cho luồng mới
  - initflags: [IN] cờ điều khiển tạo luồng
  - *thrdaddr*: [OUT] trỏ tới giá trị định danh của luồng mới

13

13

Kỹ thuật đa luồng



14

14

Một số hàm xử lý luồng

- *void _endthreadex(unsigned retval)*: kết thúc luồng và đảm bảo thu hồi tài nguyên (thực tế không cần thiết)
- Đồng bộ luồng:
  - *WaitForSingleObject(), WaiForSingleObjectEx()*
  - *WaitForMultipleObjects(), WaitForMultipleObjectsEx()*
  - *MsgWaitForMultipleObjects(), MsgWaitForMultipleObjectsEx()*
- Các đối tượng thường sử dụng cho đồng bộ:
  - Sự kiện: *SetEvent()*
  - Mutex: *CreateMutex(), ReleaseMutex()*
  - Đoạn găng: *EnterCriticalSection(), LeaveCriticalSection()*
  - Semaphore: *CreateSemaphore(), ReleaseSemaphore()*
  - Bộ đếm: *CreateWaitableTimer()*

15

15

Ví dụ Viết lại Echo server (tiếp)

```
//Step 5: Communicate with clients

SOCKET connSocket;
sockaddr_in clientAddr;
int clientAddrLen = sizeof(clientAddr);

while(1){
   connSocket = accept(listenSock, (sockaddr *) & clientAddr,
                                   &clientAddrLen);
   _beginthreadex(0,0, echoThread, (void *)connSocket, 0, 0);
}
```

16

16

Viết lại Echo server (tiếp)

```
/* echoThread- Thread to receive the message from client and echo*/

unsigned __stdcall echoThread(void *param){
   char buff[1024];
   int ret;
   SOCKET connectedSocket = (SOCKET) param;

   ret = recv(connectedSocket, buff, 1024, 0);
   if(ret < 0)
        printf("Error! Cannot receive message.\n");
   else{
        ret = send(connectedSocket, buff, ret, 0);
        if(ret < 0)
           printf("Error! Cannot send message.\n");
   }
   shutdown(connectedSocket, SD_SEND);
   closesocket(connectedSocket);
   return 0;
}
```

17

17

Điều độ luồng sử dụng đoạn găng

- Khai báo đoạn găng

```
CRITICAL_SECTION tenBien;
```

- Khởi tạo đoạn găng

```
void InitializeCriticalSection(CRITICAL_SECTION*);
```

- Giải phóng đoạn găng

```
void DeleteCriticalSection(CRITICAL_SECTION*);
```

- Yêu cầu vào đoạn găng

```
EnterCriticalSection(CRITICAL_SECTION*);
```

- Rời khỏi đoạn găng

```
LeaveCriticalSection(CRITICAL_SECTION*);
```

18

18

Điều độ luồng sử dụng đoạn găng

- Luồng chính

```
// Global variable
CRITICAL_SECTION criticalSection;

int main(void)
{
   // ...
   // Initialize the critical section one time only.
   InitializeCriticalSection (&criticalSection);
   // ...
   // Release resources used by the critical section object.
   DeleteCriticalSection(&criticalSection);
}
```

19

Điều độ luồng sử dụng đoạn găng

- Hàm thực hiện trong luồng con

```
unsigned int __stdcall mythread(void*)
{
   // ...
   // Request ownership of the critical section.
   EnterCriticalSection(&criticalSection);

   // Access the shared resource.

   // Release ownership of the critical section.
   LeaveCriticalSection(&criticalSection);
   // ...
   return 0;
}
```

20

Sử dụng đoạn găng – Ví dụ

```
#define NUM_THREAD 4
CRITICAL_SECTION critical;
int sharedValue;
int main(int argc, char* argv[])
{
   int i;
   HANDLE myhandle[NUM_THREAD];
   InitializeCriticalSection(&critical);
   for(i = 0; i < NUM_THREAD, i++)
      myhandle[i] = (HANDLE)_beginthreadex(..., &mythread,
                                           (void *)i,...);
   WaitForMultipleObject(NUM_THREAD, myhandle, TRUE, INFINITE);
   DeleteCriticalSection(&critical);

   return 0;
}
```

21

21

Sử dụng đoạn găng – Ví dụ

```
unsigned int __stdcall mythread(void* )
{
      while (sharedValue < 25) {
      EnterCriticalSection(&critical);
      int number = sharedValue++;
      LeaveCriticalSection(&critical);
      if (isPrime(number));
          printf("Value = %d is prime\n",number);
   }
   return 0;
}
```

22

22

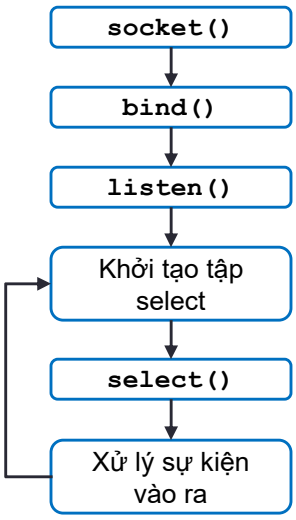
3. KỸ THUẬT THĂM DÒ

23

Kỹ thuật thăm dò

- Sử dụng hàm *select():*
  - Thăm dò các trạng thái trên socket (gửi dữ liệu, nhận dữ liệu, kết nối thành công, yêu cầu kết nối...)
  - Các socket cần thăm dò được đặt vào cấu trúc *fd_set*
- Có thể xử lý tập trung tất cả các socket trong cùng một thread (tối đa 1024).

```
typedef struct fd_set {
  u_int  fd_count;
  SOCKET fd_array[FD_SETSIZE];
} fd_set;
```



24

Hàm *select()*

```
int select(
   int nfds,           // Luôn truyền giá trị 0
   fd_set * readfds,   // [IN, OUT]tập các socket được
                       // thăm dò trạng thái có thể đọc
   fd_set * writefds,  // [IN, OUT]tập các socket được
                       // thăm dò trạng thái có thể ghi
   fd_set * exceptfds, // [IN, OUT]tập các socket được
                       // thăm dò trạng thái có lỗi
   const struct timeval * timeout // thời gian chờ trả về
);
```

- Các tập readfds, writefds, exceptfds có thể NULL, nhưng không thể cả ba cùng NULL.
- Trả về:
  - Thất bại: SOCKET_ERROR
  - Xảy ra time-out: 0
  - Thành công: tổng số socket có trạng thái sẵn sàng/có lỗi

25

25

Kỹ thuật thăm dò

- Thao tác với *fd_set* qua các macro
  - *FD_CLR(SOCKET s, fd_set *set)*: Xóa socket ra khỏi tập thăm dò
  - *FD_SET(SOCKET s, fd_set *set)*: Thêm socket vào tập thăm dò
  - *FD_ISSET(SOCKET s, fd_set *set)*: trả lại 1 nếu socket có trong tập thăm dò. Ngược lại, trả lại 0
  - *FD_ZERO(fd_set *set)*: Khởi tạo tập thăm dò bằng giá trị **null**
- Sử dụng kỹ thuật thăm dò:
  - B1: Thêm các socket cần thăm dò trạng thái vào tập *fd_set* tương ứng
  - B2: Gọi hàm *select()*. Khi hàm *select()* thực thi, các socket không mang trạng thái thăm dò sẽ bị xóa khỏi tập *fd_set* tương ứng
  - B3: Sử dụng macro *FD_ISSET()* để sự có mặt của socket trong tập *fd_set* và xử lý

26

26

Kỹ thuật thăm dò

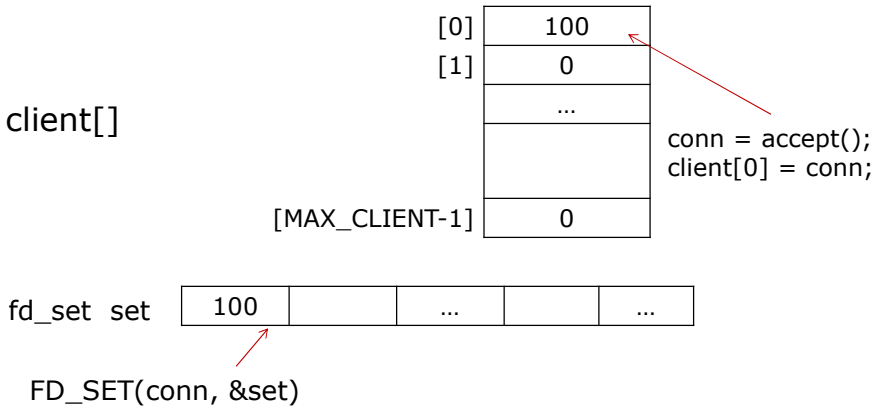
Các trạng thái trên socket được ghi nhận:

- Tập *readfds*:
  - Có yêu cầu kết nối tới socket đang ở trạng thái lắng nghe(LISTENING)
  - Dữ liệu sẵn sàng trên socket để đọc
  - Kết nối bị đóng/reset/hủy
- Tập *writefds:*
  - Kết nối thành công khi gọi hàm *connect()* ở chế độ non-blocking
  - Sẵn sàng gửi dữ liệu
- Tập *exceptfds*
  - Kết nối thất bại khi gọi hàm *connect()* ở chế độ non-blocking
  - Có dữ liệu OOB (out-of-band) để đọc

27

Sử dụng *select()* cho TCP server

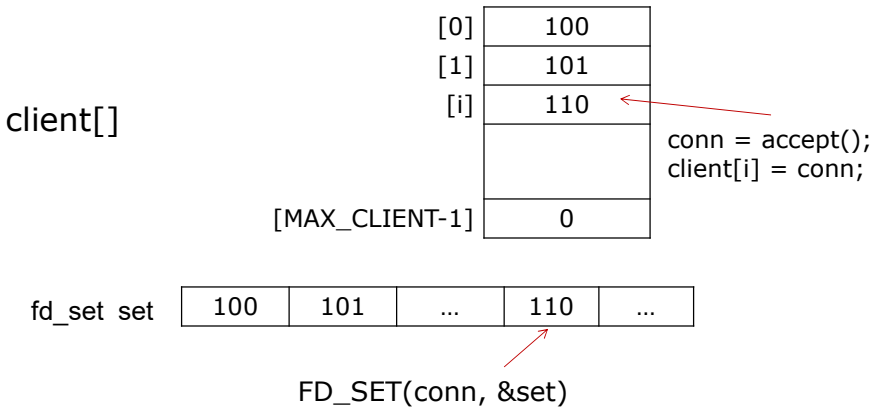
- Kết nối đầu tiên được xử lý



28

Sử dụng *select()* cho TCP server(tiếp)

- Kết nối tiếp theo được xử lý

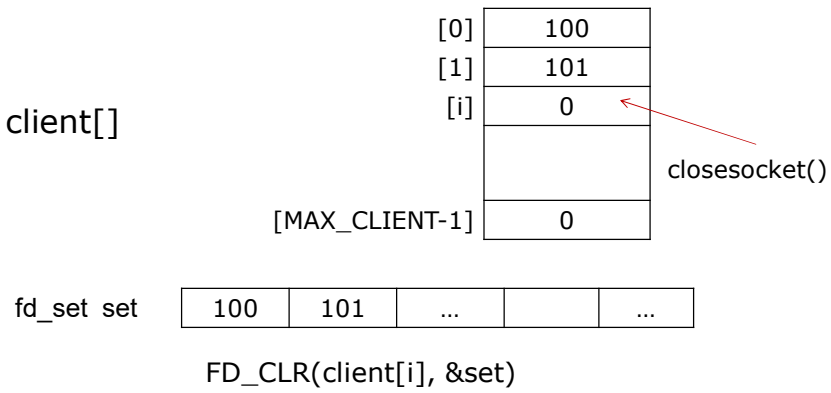


29

29

Sử dụng *select()* cho TCP server(tiếp)

- Ngắt một kết nối, giải phóng socket



30

30

Sử dụng *select()* cho TCP server(tiếp)

```
listenSock = socket(…);
listen(listenSock, …);

//Assign initial value for the array of connection socket
for(…) client[i] = 0;

//Assign initial value for the fd_set
FD_ZERO (&initfds);
FD_SET(listenSock,&initfds);
//Communicate with clients
while(1){
   readfds = initfds;

   select(0, &readfds, …);
```

31

31

Sử dụng *select()* cho TCP server(tiếp)

```
   //check the status of listenSock
   if(FD_ISSET(listenSock,&readfds)){
       connSock = accept(…);
       for(i = 0; i < FD_SETSIZE; i++)
          if (client[i] == 0)
              client[i] = connfd;
              FD_SET(client[i],&initfds);
       }
   //check the status of connfd(s)
   for(…){
       if(FD_ISSET(client[i],…)){
          doSomething();
          closesocket(client[i]);
          client[i] = 0;
          FD_CLEAR(client[i],&initfds)
       }
} //end while
```

32

32

TCP Echo Server (viết lại)

```
//Step 5: Communicate with clients
SOCKET client[FD_SETSIZE], connSock;
fd_set initfds, readfds;
sockaddr_in clientAddr;
int ret, nEvents, clientAddrLen;
char rcvBuff[1024], sendBuff[1024];

for(int i = 0; i < FD_SETSIZE; i++)
   client[i] = 0;

FD_ZERO(&initfds);
FD_SET(listenSock, &initfds);
```

33

33

TCP Echo Server (viết lại – tiếp)

```
while(1){
   readfds = initfds;
   nEvents = select(0, &readfds, 0,0,0);
   if(nEvents < 0){
       printf("\nError! Cannot check all of sockets.");
       break;
   }
   if(FD_ISSET(listenSock, &readfds)){     //new client
       clientAddrLen = sizeof(clientAddr);
       connSock = accept(listenSock, (sockaddr *)
                              &clientAddr, &clientAddrLen);
       int i;
       for(i = 0; i < FD_SETSIZE; i++)
          if(client[i] <= 0){
              client[i] = connSock;
              FD_SET(client[i], &initfds);
              break;
          }
```

34

34

TCP Echo Server (viết lại – tiếp)

```
        if(i == FD_SETSIZE){
           closesocket(connSock);
           printf("\nToo many clients.");
        }
        if(--nEvents <=0)continue; //no more event
   }
   for(int i = 0; i < FD_SETSIZE; i++){
        if(client[i] <= 0) continue;
        if(FD_ISSET(client[i], &initfds)){
           ret = Receive (client[i], rcvBuff, 1024, 0);
           if (ret <= 0){
                FD_CLR(client[i], &readfds);
                closesocket(client[i]);
                client[i] = 0;
           }
           else if(ret > 0){
                processData(rcvBuff, sendBuff);
                Send (client[i], sendBuff, strlen(sendBuff), 0);
           }
           if(--nEvents <=0)continue; //no more event
        }
    }
}//end while
```

35

35

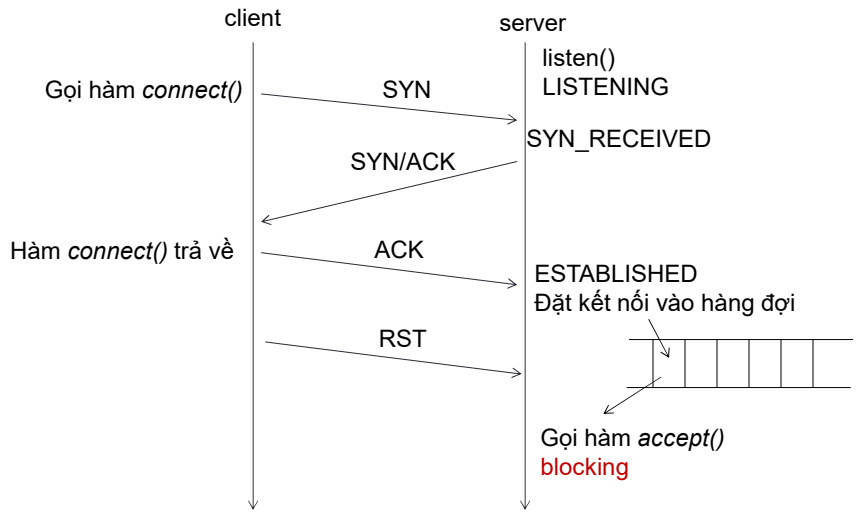
TCP Echo Server (viết lại – tiếp)

```
/* The processData function copies the input string to output*/
void processData(char *in, char *out){
        strcpy_s(out, 1024, in);
}
/* The recv() wrapper function*/
int receiveData(SOCKET s, char *buff, int size, int flags){
        int n;
        n = recv(s, buff, size, flags);
        if(n < 0)
                printf("receive error.");
        else
                buff[n] = 0;
        return n;
}
/* The send() wrapper function*/
int sendData(SOCKET s, char *buff, int size, int flags){
        int n;
        n = send(s, buff, size, flags);
        if(n < 0)
                printf("send error.");
        return n;
}
```

36

36

Chặn dừng trên hàm accept()



37

37

non-blocking *accept()*

```
//Step 2: Construct socket
SOCKET listenSock;
unsigned long ul = 1;
listenSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
ioctlsocket(listenSock, FIONBIO, (unsigned long *) &ul);
//...
while(1){
   //...
   if(FD_ISSET(listenSock, &readfds)){     //new client
        clientAddrLen = sizeof(clientAddr);
        if((connSock = accept(...))!= INVALID_SOCKET{
           int i;
           for(i = 0; i < FD_SETSIZE; i++)
              if(client[i] <= 0){
                   client[i] = connSock;
                   break;
              }
           if(i == FD_SETSIZE) printf("\nToo many clients.");
        }//...
}
```

38

38

4. KỸ THUẬT VÀO RA THEO THÔNG BÁO

39

39

CƠ BẢN VỀ LẬP TRÌNH WIN32 API

40

40

Lập trình Windows cơ bản

- Application Programming Interface: Giao diện lập trình ứng dụng
- Cung cấp thư viện liên kết động (.dll) chứa các hàm truy cập tài nguyên trong hệ thống (Windows)
- Các ứng dụng có thể truy cập đến các hàm API
- Kiểu dữ liệu HANDLE: số nguyên 32-bit được HĐH sử dụng để định danh cho một đối tượng tài nguyên nào đó (file, cửa sổ, socket,…)
  - Ví dụ: HWND, HINSTANCE, SOCKET

41

41

Một số kiểu dữ liệu

Kiểu	Ý nghĩa
BYTE	Số nguyên 8-bit không dấu
WORD	Số nguyên 16-bit không dấu
DWORD	Số nguyên 32-bit không dấu
UINT	Số nguyên 32-bit không dấu
LONG	Số nguyên 32-bit có dấu
LRESULT	Số nguyên 32-bit có dấu trả về bởi hàm xử lý thông điệp
LPSTR	Con trỏ xâu ký tự kiểu ANSI (8-bit)
LPWSTR	Con trỏ xâu ký tự kiểu Unicode(16 bit)
WPARAM	Số nguyên không dấu (32-bit với x86 và 64-bit với x64)
LPARAM	Số nguyên có dấu (32-bit với x86 và 64-bit với x64)
LPVOID	Con trỏ kiểu void
ATOM	Số nguyên 16-bit không dấu
LPTSTR	Con trỏ xâu ký tự kiểu tự thích nghi(8-bit hoặc 16-bit)

42

42

Quy ước lời gọi hàm của C/C++

- Mô tả cách thức hệ thống xử lý lời gọi hàm
  - Thứ tự truyền tham số
  - Cách thức truyền tham số
  - Thanh ghi dành riêng
  - Vào ra trên stack
- __stdcall
  - Truyền từ phải qua trái
  - Kiểu truyền mặc định: tham trị
  - Hàm được gọi (callee) lấy giá trị tham số từ đỉnh stack
  - Hàm được gọi xóa stack trước khi trả về kết quả
  - Tên hàm: bắt đầu bởi ký tự '_'
- __cdecl: tương tự __stdcall
  - Hàm gọi xóa stack sau khi hàm được gọi trả về kết quả

43

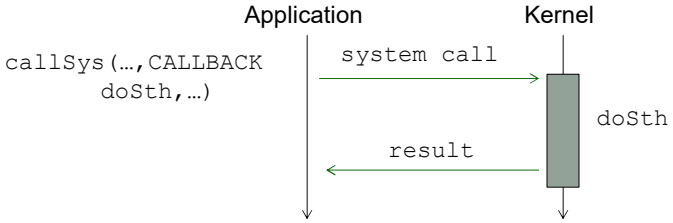
43

Một số quy ước gọi hàm của Windows

- Hàm kiểu CALLBACK

```
#define CALLBACK __stdcall
```

  - Hàm được gọi lại bởi một hàm hệ thống
  - Ứng dụng "nói" cho hệ thống biết cách thức xử lý một đối tượng nào đó khi lời gọi hệ thống được thực hiện



- Hàm kiểu WINAPI: sử dụng trong ứng dụng Windows

```
#define WINAPI __stdcall
```

44

44

Lập trình hướng sự kiện

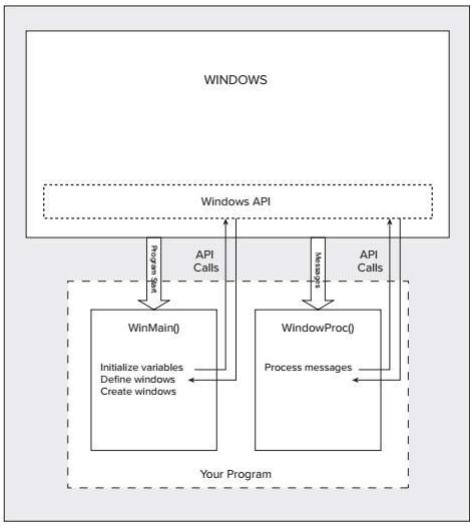
- Luồng thực thi chương trình quyết định bởi các sự kiện xảy ra
- Sự kiện trên cửa sổ:
  - Hệ thống (Windows) ghi lại mỗi sự kiện xảy ra (event) trong một thông điệp (message) và đặt trong hàng đợi thông điệp (messages queue)
  - HĐH Windows đảm nhiệm việc truyền thông điệp vào cửa sổ của ứng dụng đó
  - Hàm kiểu CALLBACK cần được định nghĩa để xử lý các thông điệp mà cửa sổ nhận được
    - Với những thông điệp không xử lý, có thể truyền lại cho hệ thống qua hàm DefWindowProc()

45

45

Cấu trúc chương trình

- Hàm WINAPI WinMain():
  - Khởi tạo giá trị biến
  - Định nghĩa cửa sổ
  - Khởi tạo cửa sổ
- Hàm CALLBACK WindowProc():
  - Xử lý thông điệp



46

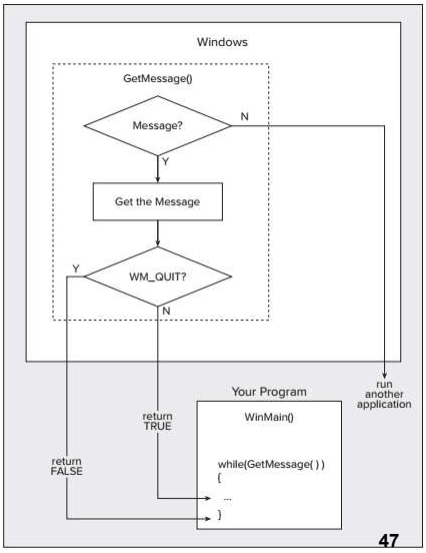
46

Hàm WinMain()

- Xử lý hàng đợi thông điệp

```
MSG msg;
while (GetMessage(&msg,NULL,0,0)){
   //Dịch sự kiện thành thông điệp
   TranslateMessage (&msg);
   //Lấy thông điệp
   DispatchMessage(&msg);
}
```

- Hàm GetMessage() trả về 0 khi thông điệp chứa định danh WM_QUIT báo chương trình kết thúc



47

47

Cấu trúc MSG

```
typedef struct tagMSG {
   HWND   hwnd;      // Cửa sổ nhận thông điệp
   UINT   message;   // Định danh thông điệp
   WPARAM wParam;    // Nội dung thông điệp
   LPARAM lParam;    // Nội dung thông điệp
   DWORD  time;      // Thời điểm thông điệp được đưa lên
   POINT  pt;        // Vị trí con trỏ khi thông điệp được
                     // đưa lên
} MSG, *PMSG, *LPMSG;
```

48

48

Lớp cửa sổ WNDCLASS

```
typedef struct tagWNDCLASS {
    UINT        style;
    WNDPROC     lpfnWndProc;
    int         cbClsExtra;
    int         cbWndExtra;
    HINSTANCE   hInstance;
    HICON       hIcon;
    HCURSOR     hCursor;
    HBRUSH      hbrBackground;
    LPCTSTR     lpszMenuName;
    LPCTSTR     lpszClassName;
} WNDCLASS, *PWNDCLASS;
```

- Lớp WNDCLASSEX bổ sung thêm trường **cbSize** chứa kích thước cấu trúc và **hIconSm** chứa icon nhỏ của cửa sổ

49

49

Cấu trúc cửa sổ WNDCLASS

Tên trường	Ý nghĩa	Gán giá trị
`style`	Kiểu hiển thị	
`lpfnWndProc`	Con trỏ hàm xử lý cửa sổ	
`cbClsExtra`	Kích thước (byte) cấp phát thêm sau biến cấu trúc WNDCLASS	0: gán mặc định bởi hệ thống
`cbWndExtra`	Kích thước (byte) cấp phát thêm cho tiến trình	0: gán mặc định bởi hệ thống
`hInstance`	Định danh của tiến trình	
`hIcon`	Định danh của icon	Gọi hàm `LoadIcon()`
`hCursor`	Định danh con trỏ chuột	Gọi hàm `Cursor()`
`hbrBackground`	Định danh chổi vẽ	
`lpszMenuName`	Tên menu	
`lpszClassName`	Tên cửa sổ	*Gán bằng 1 xâu*

50

50

Đăng ký và tạo cửa sổ

- Đăng ký cửa sổ

```
ATOM RegisterClass(CONST WNDCLASS * lpWndClass);
ATOM RegisterClassEx(CONST WNDCLASS * lpWndClass);
```

- Tạo cửa sổ

```
HWND WINAPI CreateWindow(
   LPCTSTR   lpClassName,     // Tên cửa sổ
   LPCTSTR   lpWindowName,    // Tên hiển thị
   DWORD     dwStyle,         // Kiểu cửa sổ
   int       x,               // Hoành độ vị trí hiển thị
   int       y,               // Tung độ vị trí hiển thị
   int       nWidth,          // Kích thước ngang
   int       nHeight,         // Kích thước cao
   HWND      hWndParent,      // Cửa sổ cha
   HMENU     hMenu,           // Định danh menu
   HINSTANCE hInstance,       // Định danh tiến trình
   LPVOID    lpParam          // Con trỏ tham số truyền
                              // cho cửa sổ
);
```

51

51

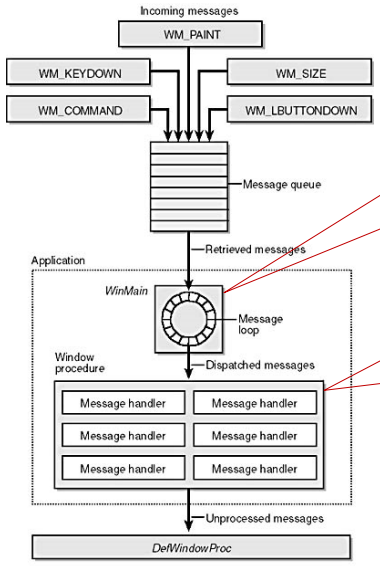
Khai báo hàm CALLBACK

```
LRESULT CALLBACK windowProc(
   HWND hWnd,          // Định danh cửa sổ
   UINT message,       // Định danh thông điệp
   WPARAM wParam,      // Nội dung thông điệp
   LPARAM lParam){     // Nội dung thông điệp

   switch(message){
        case WM_XXX:
        …
        case WM_DESTROY:       // Cửa sổ bị hủy
           PostQuitMessage(0);
           …
        case WM_CLOSE:         // Cửa sổ bị đóng
           DestroyWindow(hWnd);
   }
   DefWindowProc(hWnd, message, wParam, lParam);
}
```

52

52

Hoạt động của chương trình




```
int WINAPI WinMain (
    HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow)
{
}
```

```
LRESULT CALLBACK WinProc (
        HWND hWnd,
        UINT message,
        WPARAM wParam,
        LPARAM lParam)
{
}
```

53

53

Kỹ thuật vào ra socket theo thông báo

- Ứng dụng GUI có thể nhận được các thông điệp từ WinSock qua cửa sổ của ứng dụng.
- Khi có sự kiện xảy ra trên socket, một thông điệp được thông báo tới cửa sổ ứng dụng
- Hàm *WSAAsyncSelect()* được sử dụng để chuyển socket sang chế độ không chặn dừng và thiết lập tham số cho việc xử lý sự kiện
- Trả về:
  - Thành công: 0
  - Lỗi: SOCKET_ERROR

```
int WSAAsyncSelect(
    SOCKET s,           // [IN] Socket sẽ xử lý sự kiện
    HWND hWnd,          // [IN] Handle cửa sổ nhận sự kiện
    unsigned int wMsg, // [IN] Mã thông điệp, tùy chọn,
                        // thường lớn hơn WM_USER
    long lEvent         // [IN] Mặt nạ xác định các sự
                        // kiện ứng dụng muốn nhận
);
```

54

54

Một số giá trị mặt nạ

FD_READ	• Còn dữ liệu nhận được mà chưa đọc
FD_WRITE	• Có thể gửi dữ liệu(send(), sendto()) • Có liên kết được thiết lập (accept(), connect())
FD_OOB	Có dữ liệu out-of-band sẵn sàng để nhận
FD_ACCEPT	Còn kết nối chưa được gắn socket
FD_CONNECT	Kết nối được thiết lập
FD_CLOSE	Ngắt kết nối hoặc giải phóng socket
FD_ADDRESS_LIST_CHANGE	Địa chỉ của giao tiếp mạng thay đổi
FD_ROUTING_INTERFACE_CHANGE	Thông tin default gateway của giao tiếp mạng thay đổi

- Sử dụng toán tử nhị phân OR để nhận thông báo từ nhiều sự kiện

Ví dụ: FD_READ | FD_WRITE | FD_CLOSE

55

55

Lưu ý

- Để thiết lập lại chế độ chặn dừng cho socket:
  1. Gọi lại hàm *WSAAsyncSelect( )* với lEvent = 0
  2. Gọi hàm *ioctlsocket( )* thiết lập lại chế độ chặn dừng
- Socket trả về từ hàm *accept()* sử dụng cùng mã thông điệp và mặt nạ sự kiện với listenning socket
  - Gọi hàm *WSAAsyncSelect()* để thiết lập các giá trị khác (nếu cần)
- Hai cách gọi sau là không tương đương

```
WSAAsyncSelect(s, hWnd, WM_SOCKET, FD_READ | FD_WRITE);
```

và

```
WSAAsyncSelect(s, hWnd, WM_SOCKET, FD_READ);
WSAAsyncSelect(s, hWnd, WM_SOCKET, FD_WRITE);
```

56

56

Xử lý thông báo

- Khi cửa sổ nhận được thông điệp, HĐH gọi hàm *windowProc()* tương ứng với cửa sổ đó

```
LRESULT CALLBACK windowProc(HWND hWnd, UINT uMsg,
                            WPARAM wParam, LPARAM lParam);
```

- Khi cửa sổ nhận được các sự kiện liên quan đến WinSock:
  - *uMsg* sẽ chứa mã thông điệp mà ứng dụng đã đăng ký bằng WSAAsyncSelect(ví dụ WM_SOCKET)
  - *wParam* chứa bản thân socket xảy ra sự kiện
  - Nửa cao của *lParam* chứa mã lỗi nếu có, nửa thấp chứa giá trị mặt nạ của sự kiện. Sử dụng hai MACRO là WSAGETSELECTERROR và WSAGETSELECTEVENT để kiểm tra lỗi và sự kiện xảy ra trên socket

57

57

Sử dụng *WSAAsyncSelect()*

```
LRESULT CALLBACK windowProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
                   hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{

   //Registering the Window Class
   WNDCLASSEX wcex;
   wcex.lpfnWndProc = windowProc;
   wcex.hInstance = hInstance;
   //...
   RegisterClassEx(&wcex);

   //Create the window
   HWND hWnd;
   hWnd = CreateWindow(..., hInstance, ...);
   ShowWindow(hWnd, SW_SHOWNORMAL);
   UpdateWindow(hWnd);
}
```

58

58

Sử dụng *WSAAsyncSelect() – Tiếp (1)*

```
   //Construct listenning socket
   SOCKET listenSock;
   listenSock = socket(...);

   //requests Windows message-based notification of network
   //events for listenSock
   WSAAsyncSelect(listenSock, hWnd,
                           WM_SOCKET,FD_ACCEPT|FD_CLOSE);

   // Call bind(), listen()

   //Translate and dispatch window messages for the
   //application thread
   while( GetMessage(&Msg, NULL, 0, 0) )
       {
             TranslateMessage(&Msg);
             DispatchMessage(&Msg);
       }
   return 0;
}
```

59

59

Sử dụng *WSAAsyncSelect() – Tiếp (2)*

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam){
   SOCKET serverSock = (SOCKET) wParam;
   switch(message){
       case WM_DESTROY:
          PostQuitMessage(0);
          break;
       case WM_CLOSE:
          DestroyWindow(hWnd);
          break;
       case WM_SOCKET:
          if (WSAGETSELECTERROR(lParam)){ // check socket error
              closesocket(...); // close socket
          }
```

60

60

Sử dụng *WSAAsyncSelect() – Tiếp (3)*

```
            switch(WSAGETSELECTEVENT(lParam)){
                case FD_ACCEPT:
                   connSock = accept(serverSock,...);
                   WSAAsyncSelect(connSock, hWnd, WM_SOCKET,... );
                   //...
                case FD_READ://...
                case FD_WRITE: //...
                case FD_CLOSE: //...
            }
            break;
        }

    return DefWindowProc(hWnd, message, wParam, lParam);
}
```

61

61

TCP Echo server – Viết lại

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
   MSG msg;
   HWND serverWindow;

   //Registering the Window Class
   MyRegisterClass(hInstance);

   //Create the window
   if ((serverWindow = InitInstance (hInstance, nCmdShow))==NULL)
        return FALSE;
   //Initiate WinSock
   WSADATA wsaData;
   WORD wVersion = MAKEWORD(2,2);
   if(WSAStartup(wVersion, &wsaData)){
        MessageBox(serverWindow, L"Cannot listen!",
                                              L"Error!", MB_OK);
        return 0;
   }
```

62

62

TCP Echo server – Viết lại(tiếp)

```
   //Construct socket
   listenSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

   /*requests Windows message-based notification of network events
     for listenSock*/
   WSAAsyncSelect(listenSock, serverWindow,
                  WM_SOCKET,FD_ACCEPT|FD_CLOSE);

   //Bind address to socket...
   //Listen request from client...
   // Main message loop:
   while (GetMessage(&msg, NULL, 0, 0)){
        TranslateMessage(&msg);
        DispatchMessage(&msg);
   }

   return 0;
}
```

63

63

TCP Echo server – Viết lại(tiếp)

```
ATOM MyRegisterClass(HINSTANCE hInstance)
{
   WNDCLASSEX wcex;

   wcex.cbSize         = sizeof(WNDCLASSEX);
   wcex.style                  = CS_HREDRAW | CS_VREDRAW;
   wcex.lpfnWndProc    = windowProc;
   wcex.cbClsExtra     = 0;
   wcex.cbWndExtra     = 0;
   wcex.hInstance      = hInstance;
   wcex.hIcon          = LoadIcon(hInstance,
                       MAKEINTRESOURCE(IDI_WSAASYNCSELECTSERVER));
   wcex.hCursor        = LoadCursor(NULL, IDC_ARROW);
   wcex.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
   wcex.lpszMenuName   = NULL;
   wcex.lpszClassName  = L"WindowClass";
   wcex.hIconSm        = LoadIcon(wcex.hInstance,
                                  MAKEINTRESOURCE(IDI_SMALL));

   return RegisterClassEx(&wcex);
}
```

64

64

TCP Echo server – Viết lại(tiếp)

```
LRESULT CALLBACK windowProc(HWND hWnd, UINT message,
                                    WPARAM wParam, LPARAM lParam)
{
   SOCKET connSock;
   sockaddr_in clientAddr;
   int ret, clientAddrLen = sizeof(clientAddr), i;
   char rcvBuff[BUFF_SIZE], sendBuff[BUFF_SIZE];

   switch(message){
        case WM_SOCKET:
        {
           if (WSAGETSELECTERROR(lParam)){
                for(i = 0; i < MAX_CLIENT; i++)
                   if(client[i] == (SOCKET) wParam){
                        closesocket(client[i]);
                        client[i] = 0;
                        continue;
                   }
                }
```

65

65

TCP Echo server – Viết lại(tiếp)

```
        switch(WSAGETSELECTEVENT(lParam)){
           case FD_ACCEPT:
           {
                connSock = accept((SOCKET)wParam,
                        (sockaddr *) &clientAddr, &clientAddrLen);
                if(connSock == INVALID_SOCKET)
                   break;
                for(i = 0; i < MAX_CLIENT; i++)
                   if(client[i] == 0){
                        client[i] = connSock;
                        break;
                        /*requests Windows message-based notification
                          of network events for listenSock*/

                        WSAAsyncSelect(client[i], hWnd,
                                        WM_SOCKET,FD_READ|FD_CLOSE);
                   }
                if(i == MAX_CLIENT)
                   MessageBox(hWnd, L"Too many clients!",
                                                L"Notice", MB_OK);
           }
           break;
```

66

66

TCP Echo server – Viết lại(tiếp)

```
        case FD_READ:{
           for(i = 0; i < MAX_CLIENT; i++)
                if(client[i] == (SOCKET) wParam)
                   break;
           ret = recv(client[i], rcvBuff, BUFF_SIZE, 0);
           if(ret > 0){
                rcvBuff[ret] = 0;
                processData(rcvBuff, sendBuff);
                send(client[i], sendBuff, strlen(sendBuff), 0);
           }
        }
        break;
        case FD_CLOSE: {
           for(i = 0; i < MAX_CLIENT; i++)
                if(client[i] == (SOCKET) wParam){
                   closesocket(client[i]);
                   client[i] = 0;
                   break;
                }
        }
        break;
```

67

67

TCP Echo server – Viết lại(tiếp)

```
        case WM_DESTROY:
        {
           PostQuitMessage(0);
           shutdown(listenSock, SD_BOTH);
           closesocket(listenSock);
           WSACleanup();
           return 0;
        }
        break;

        case WM_CLOSE:
        {
           DestroyWindow(hWnd);
           shutdown(listenSock, SD_BOTH);
           closesocket(listenSock);
           WSACleanup();
           return 0;
        }
        break;
        }
   return DefWindowProc(hWnd, message, wParam, lParam);
}
```

68

68

Còn tiếp…

69