Architettura dei sistemi Windows

Enrico Lodolo
e.lodolo@bo.nettuno.it

Inquadramento storico

- **Windows nasce nel 1985. Le prime due versioni sono poco più che prototipi con forti limitazioni. Girano in modalità reale 8086 (limite a 640k di memoria) o nella modalità protetta 80286.**
- **La prima versione "utilizzabile" - Windows 3.0 - esce nel 1991 e sfrutta le capacità dei processori di classe 386. Nascono le prime applicazioni commerciali di buon livello.**
- **Nella metà degli anni 80 erano di moda i cosiddetti pacchetti integrati (Symphony, Framework...) che racchiudevano in un'unica applicazione le funzioni più importanti: word processor, foglio elettronico, database, comunicazioni.**
- **Questi pacchetti erano però squilibrati: una delle funzioni era completa mentre le altre erano insoddisfacenti.**
- **L'obiettivo di Windows è di spostare l'integrazione fra funzioni applicative a livello di ambiente operativo: l'utente può quindi scegliere sul mercato gli applicativi specializzati migliori lasciando al sistema operativo il compito di integrarli.**

La famiglia WinXX

- **Si tratta di una famiglia di ambienti, indicati di solito come WinXX.**
- **4 sistemi: Windows 3.x (16 bit), Windows 95/98 (ibrido 16/32), Windows NT (32 bit) e Windows CE (32bit per palmari ed embedded).**
- **Windows 3.x è in fase di abbandono ma ha ancora una notevole base di installato, soprattutto nelle grandi aziende.**
- **Windows 95/98 è destinato al mercato "consumer" e a quello dei portatili. E' un ambiente ibrido in cui convivono parti a 16 e parti a 32 bit, orientato a fornire la massima compatibilità con il DOS, con problemi intrinseci di fragilità.**
- **Windows NT è stato pensato come sistema per uso professionale. E' un 32bit vero con un'architettura microkernel molto robusta (basato in parte su tecnologia Digital VMS) che si rivolge allo stesso mercato dei sistemi Unix (comprende un sottosistema Posix compliant).**
- **Supporta SMP (symmetrical multi processing).**
- **La release corrente è la 4.0 ed esiste in due versioni: server e workstation.**

Architettura: API e DLL

- **E' un ambiente GUI (Graphical User Interface),costruito intorno al concetto di finestra che, come vedremo, è qualcosa di più di un semplice elemento grafico**
- **Struttura modulare: il sistema operativo è formato da una collezione di librerie ad aggancio dinamico (DLL). Queste DLL costituiscono la cosiddetta API (Application Program Interface) ovvero l'interfaccia fra le applicazioni e il sistema operativo.**
- **Esistono 2 API: Win16 (per i sistemi a 16 bit) e Win32 (per i sistemi a 32 bit). A parte alcune piccole differenze Win95/98 e WinNT hanno la stessa API.**
- **Il cuore dell'API è formato da 3 DLL:**
  - Kernel: memory manager, scheduler, loader
  - User: sistema di windowing
  - GDI: Graphics Device Interface
- **Estensibilità: l'API si espande aggiungendo nuove DLL (per esempio WinSocket per la comunicazione su TCP/IP)**

Architettura: driver e sottosistemi

- **Windows fornisce una serie di servizi avanzati che spostano molte problematiche dall'area delle applicazioni a quella del sistema operativo**
- **L'obiettivo è quello di fornire un ambiente applicativo il più possibile astratto dall'hardware sottostante**
- **Questo risultato viene ottenuto attraverso una serie di sottosistemi basati su driver**
- **Un esempio tipico è la GDI che rende le applicazioni indipendenti dal dispositivo di output. Video e stampanti sono trattati allo stesso modo e un insieme di driver consente di mappare le chiamate sulle varie schede grafiche e sulle varie stampanti**
- **Altri sottosistemi disponibili sono: MAPI (posta elettronica), TAPI (modem e servizi di telefonia), RAS (accesso remoto), WinSocket (networking), ODBC (accesso ai database via SQL)**
- **In tutti questi casi abbiamo un'estensione dell'API, quindi una DLL, che ridirige le chiamate ai driver sottostanti. Le applicazioni accedono ai servizi indipendentemente dagli strati di basso livello.**

Architettura: DLL

- **In Windows abbiamo due tipi di "eseguibili": EXE e DLL**
- **DLL = Dynamic Link Library**
- **Le funzioni esportate da una DLL vengono agganciate a runtime e non durante la compilazione**
- **Abbiamo due modalità di aggancio di una DLL:**
  - Link statico: al caricamento dell'applicazione
  - Link dinamico: in qualunque momento mediante una opportuna funzione dell'API (LoadLibrary)
- **Vantaggi: modularizzazione, condivisione di codice, sostituibilità**
- **Se più applicazioni utilizzano la stessa DLL (come avviene per esempio con quelle di sistema) questa viene caricata in memoria una sola volta.**
- **Il concetto di libreria ad aggancio dinamico è comune a quasi tutti i sistemi operativi moderni: Mac, diverse versioni di Unix (cfr. formato ELF), OS/2**

DLL: caricamento statico e dinamico

- **Caricamento statico**
  - Il linker inserisce nell'eseguibile informazioni sulle DLL utilizzate e sulle funzioni che vengono chiamate.
  - Il loader, in fase di caricamento del programma, provvede a caricare in memoria anche le DLL necessarie (se non sono già presenti) e ad eseguire gli agganci con le funzioni.
- **Caricamento dinamico**
  - Non viene inserita alcuna informazione in fase di linking e il loader non esegue alcun aggancio
  - Il caricamento e l'aggancio vengono fatti esplicitamente chiamando due funzioni dell'API: LoadLibrary e GetProcAddress
  - LoadLibrary prende come parametro il nome della DLL, carica la DLL e restituisce un'handle
  - GetProcAddress prende come parametri l'handle dalle DLL e un nome di funzione e restituisce un puntatore alla funzione
  - Alla fine dell'utilizzo si chiama FreeLibrary per scaricare la DLL

Architettura: modello ad eventi

- **In un ambiente tradizionale l'applicazione ha il controllo del flusso per la maggior parte del tempo e quando ha bisogno di un servizio chiama il sistema operativo**
- **In Windows abbiamo invece un'architettura guidata da eventi : il sistema gestisce il flusso operativo e chiama l'applicazione quando necessario.**
- **Si parla di modello "Don't call me I call you" (Hollywood Model)**
- **Questa impostazione è comune a quasi tutti gli ambienti basati su GUI (per esempio Macintosh).**

Architettura: messaggi e finestre

- **Evento: azione dell'utente (mouse tastiera) , interazione con una periferica (arrivo di un carattere sulla porta seriale o su un socket)**
- **Ogni evento provoca l'invio di un messaggio ad una o più applicazioni (in Win16 una coda unica per tutte le applicazioni, in Win32 una coda per ogni applicazione)**
- **Messaggi: base della comunicazione fra applicazioni e fra finestre di un'applicazione**
- **In Windows una finestra è un'entità in grado di ricevere e elaborare messaggi.**
- **Ogni finestra è identificata da un'handle (hwnd) che viene restituita dal sistema all'atto della creazione e che serve come ID per ogni successiva interazione, compreso l'invio di messaggi.**
- **Un messaggio è una piccola struttura dati che contiene informazioni sull'evento associato (ID, 2 parametri di tipo int, valore di ritorno)**

Architettura Windows: callback

- **Il meccanismo di callback consente ad una DLL di chiamare una funzione all'interno di un'applicazione**
- **L'applicazione passa alla DLL un puntatore ad una sua funzione**
- **La DLL usa questo puntatore per chiamare la funzione**
- **Il meccanismo di callback consente l'implementazione del sistema di mesaggi/eventi**
- **Ogni finestra ha una funzione di callback (Window Procedure) che viene usata dalle DLL di sistema per inviare i messaggi**

Modello ad eventi e multitasking

- **Questo tipo di modello permette di realizzare una forma di pseudo-multitasking che viene chiamata “cohoperative multitasking”.**
- **In Win16 in realtà abbiamo un solo processo e il flusso operativo è sequenziale. Il meccanismo di dispatching dei messaggi (un’unica coda per tutte le applicazioni) consente simulare la concorrenza.**
- **Se un’applicazione entra in un loop all’interno di una risposta ad un messaggio (cioè in WndProc), tutto il sistema rimane bloccato**
- **In Win32 si ha invece un multitasking effettivo (prehemptive): ogni applicazione è un processo ed ha una propria coda di messaggi.**
- **Uno scheduler provvede ad assegnare una “fetta” di tempo ai vari processi attivi sulla base di un sistema di priorità**
- **Oltre a ciò ogni processo può avere più threads (processi “leggeri” con memoria condivisa).**
- **L’API Win32 mette a disposizione una serie di strumenti di sincronizzazione per i threads: mutex, semafori, sezioni critiche ...**

Risorse

- **Le applicazioni Windows fanno uso di immagini (bitmap), cursori, icone ecc.**
- **Queste “risorse” vengono create esternamente e poi inserite dal linker nell’eseguibile (EXE o DLL).**
- **In pratica un eseguibile contiene una sezione di codice e un database di risorse.**
- **Esistono funzioni di API che permettono di caricare e utilizzare queste risorse**
- **Per la creazione il metodo “classico” prevede la stesura di un file .RC con le definizioni (esiste un linguaggio specifico). Il compilatore di risorse RC.EXE crea un file .RES pronto per il linker.**
- **In pratica si usano degli editor di risorse generano direttamente file .RES**
- **E’ possibile anche editare gli EXE e le DLL e modificare quindi a posteriori gli eseguibili. E’ utile per le traduzioni in quanto anche le stringhe di testo possono essere inserite fra le risorse.**

Hello Windows: struttura

- **Vediamo il classico programma “Hello World” in versione Win32**
- **hellowin.c comprende due funzioni**
- **WinMain() - entry point del programma**
  - Definizione e registrazione della Window Class: comportamenti e attributi comuni (funzione di risposta ai messaggi, icona, cursore....)
  - Creazione della finestra principale: classe di appartenenza, titolo, stile, posizione ...
  - Attivazione della finestra principale
  - Loop di attesa
- **WndProc(..) - procedura di risposta ai messaggi della Window Class**

Hello Windows - 1

```
#include <windows.h>

long __stdcall WndProc(HWND, UINT, WPARAM, LPARAM);

int __stdcall WinMain(HANDLE hInstance, HANDLE hPrevInstance,
                  LPSTR lpszCmdParam, int nCmdShow)
{
      static char zsAppName[] = "HelloWin";
      HWND        hwnd;
      MSG         msg;
      WNDCLASS    wndclass;

      wndclass.style         = CS_REDRAW | CS_VREDRAW;
      wndclass.lpfnWndProc   = WndProc;
      wndclass.cbClsExtra    = 0;
      wndclass.cbWndExtra    = 0;
      wndclass.hInstance     = hInstance;
      wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
      wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
      wndclass.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
      wndclass.lpszMenuName  = NULL;
      wndclass.lpszClassName = szAppName;

      RegisterClass(&wndclass);
```

Hello Windows - 2

```
      hwnd = CreateWindow(szAppName,          // window class name
                  "The Hello Program",    // window caption
                  WS_OVERLAPPEDWINDOW,    // window style
                  CW_USEDEFAULT,          // initial x position
                  CW_USEDEFAULT,          // initial y position
                  CW_USEDEFAULT,          // initial x size
                  CW_USEDEFAULT,          // initial y size
                  NULL,                   // parent window handdle
                  NULL,                   // window menu handle
                  hInstance,              // program instance handle
                  NULL);

      ShowWndow(hwnd, nCmdShow);
      UpdateWindow(hwnd);

      while (GetMessage(&msg, NULL, 0, 0))
           {
           TranslateMessage(&msg)
           DispatchMessage(&msg)
           }
      return msg.wParam;
}
```

Hello Windows - 3

```
long __stdcall WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
      {
      HDC         hdc;
      PAINTSTRUCT ps;
      RECT        rect;

      switch (iMsg)
           {
           case WM_PAINT:
                hdc = BeginPaint(hwnd, &ps);
                GetClientRect(hwnd, &rect);

                DrawText(hdc, "Hello, Windows 95!", -1, &rect,
                         DT_SINGLELINE | DT_CENTER | DT_VCENTER);
                EndPaint(hwnd, &ps);
                return 0;

           case WM_DESTROY:
                PostQuitMessage(0);
                return 0;
           }
      return DefWindowProc( hwnd, iMsg, wParam, lparam);
      }
```

Esempi

- **A partire da hellowin creiamo una serie di esempi che ci permettono di vedere in pratica i concetti fondamentali.**
  - hellobtn.exe: creazione di un bottone (child window) all'interno della finestra e risposta all'evento "click"
  - testdll.dll: esempio di creazione di una DLL che esporta una funzione
  - hellodll.exe: la pressione del bottone provoca la chiamata alla funzione contenuta in testdll.dll collegata in modo statico
  - hellodl2.exe: la pressione del bottone prvoca la chiamata alla funzione collegata in testdll.dll ma il collegamento avviene in modo dinamico
- **Useremo un compilatore C di pubblico dominio: lcc**

hellobtn

- **I bottoni sono finestre a tutti gli effetti, di tipo “child”, e vengono create in risposta al messaggio WM_CREATE della finestra principale. Ogni child window è identificata da un ID.**

```
case WM_CREATE :
  CreateWindow("button", "Push",
     WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON, 10, 10, 60, 30,hwnd, (LPVOID)101,
                             hCurInstance, NULL);
return 0 ;
```

- **Gli eventi delle child windows generano messaggi WM_COMMAND nella finestra principale.**

```
case WM_COMMAND:
   if (wParam=101)
      MessageBox(hwnd,"Pushed!","HelloBtn Message",MB_OK | MB_ICONEXCLAMATION);
return 0;
```

- **La sequenza di compilazione è**

```
lcc -c -Ic:\lcc\include  -g2 hellobtn.c
lcclnk -subsystem windows -o hellobtn.exe hellobtn.obj
```

testdll

- **Le DLL hanno una funzione "entry point" che viene chiamata al caricamento. Il nome di questa funzione viene indicato al linker**

```
#include <windows.h>

int _stdcall MyLibMain(void *hinstDll,unsigned long dwReason,void *reserved)
{
    return(1);
}

void Popup(HWND hwnd)
{
    MessageBox(hwnd,"Pushed!","Message from DLL",MB_OK | MB_ICONEXCLAMATION);
}
```

- **Bisogna creare un file .DEF che elenca le funzioni esportate:**

```
exports Popup
```

- **La sequenza di compilazione è**

```
lcc -O -g2 testdll.c
lcclnk.exe  -dll -entry MyLibMain testdll.obj testdll.def
implib testdll.dll
```

hellodll

- **Nella sezione di risposta al messaggio WM_COMMAND chiamiamo la funzione esportata da testdll.dll**

```
case WM_COMMAND:
  if (wParam=101)
    Popup(hwnd);
return 0 ;
```

- **Il linker ha bisogno del file testdll.lib per eseguire il collegamento statico. Questo file viene creato usando l'utility IMPLIB.EXE con la sintassi:**

```
implib testdll.dll
```

- **A questo punto possiamo compilare hellodll con la sequenza:**

```
lcc -c -Ic:\lcc\include  -g2 hellodll.c
lcclnk -subsystem windows -o hellodll.exe hellodll.obj testdll.lib
```

hellodl2

- **Definiamo un puntatore a funzione e la funzione DoPopup:**

```
FARPROC lpfnPopup; /* FARPROC è definito in windows.h */

void DoPopup(HWND hwnd)
    {
      HANDLE hDLL;

      hDLL = LoadLibrary("TESTDLL.DLL");          /* carica la DLL (se non è già caricata)  */
      if (hDLL)
      {
        lpfnPopup=GetProcAddress(hDLL,"_Popup");  /* aggancia la funzione esportata */
        (* lpfnPopup)(hwnd);                      /* chiama la funzione */
        FreeLibrary(hDLL);                        /* scarica la DLL (se nessun altro la usa) */
      }
    }
```

- **Nella risposta a WM_COMMAND chiameremo DoPopup**

```
      case WM_COMMAND:
         if (wParam=101)
           DoPopup(hwnd);
         return 0;
```

- **La sequenza di compilazione non richiede più TESTDLL.LIB:**

```
lcc -c -Ic:\lcc\include  -g2 hellodl2.c
lcclnk -subsystem windows -o hellodl2.exe hellodl2.obj
```

Da Win16 a Win32: standard e common controls

- Come abbiamo visto in HelloBtn, i controlli non sono altro che finestre di tipo child appartenenti a classi predefinite nel sistema
- In Win16 era presente un certo numero di controlli denominati “Standard controls”: Bottoni, CheckBox, Radio buttons, Edit controls, ListBox, ComboBox , Static ecc.
- E’ possibile creare controlli “custom” inserendoli all’interno di DLL e registrandoli nel sistema. Si possono acquistare sul mercato.
- In Win32 viene reso disponibile un altro gruppo di controlli denominati “Common Controls”, implementati nella DLL di sistema COMMCTRL.DLL: TreeView, ListView, TrackBar, ProgressBar, TabControl, Property Pages, RTF edit ecc.
- I controlli rappresentano una forma molto semplice di “componenti software”
- Le applicazioni creano i controlli con CreateWindow() indicando la classe opportuna e comunicano con essi mediante messaggi
- I controlli inviano messaggi di notifica (WM_COMMAND o WM_NOTIFY) alla finestra che li contiene

Da Win16 a Win32: UI objects e kernel objects

- **Gli elementi fondamentali dell'architettura di Win 16 sono i cosiddetti "User Interface Objects", cioè le finestre e gli strumenti grafici (penne, pennelli, display context, bitmap ecc.).**
- **Gli UI Objects sono strutture dati contenute all'interno del sistema operativo (in USER.DLL e GDI.DLL) , manipolabili dalle applicazioni mediante "handles" che li identificano e funzioni di API.**
- **Win32 introduce un nuovo tipo di entità chiamato "kernel object" che rappresenta il fondamento delle capacità di multitasking.**
- **Anche i kernel objects sono strutture dati residenti nel sistema operativo (in particolare nel kernel), accessibili mediante handles e funzioni di API.**
- **Sono classificabili in diverse categorie:**
  - Processi e thread
  - Strumenti di sincronizzazione: eventi, mutex, semafori
  - Oggetti "stream": file, filemapping, pipe, mailslot

Caratteristiche dei kernel objects

- **Per ogni kernel object esiste una funzione di creazione (CreateXXXX) che restituisce l'handle dell oggetto**
- **Tutti i kernel objects vengono distrutti con la stessa funzione CloseHandle(Handle)**
- **Spesso i kernel objects devono essere condivisi fra diversi processi ma un handle ha senso solo all'interno di un processo**
- **Si fa uso di diverse tecniche per condividere i K.O.: per esempio l'identificazione mediante nome.**
- **E' possibile proteggere un K.O. mediante un descrittore di protezione.**
- **Quasi tutte le chiamate di creazione hanno come parametro opzionale un puntatore ad una struttura dati di tipo SECURITY_ATTRIBUTES mediante la quale è possibile definre i diritti di accesso all'oggetto.**
- **Il meccanismo di protezione non è implementato in Windows 95/98 ma solo in NT**

Processi e thread

- **In genere si definisce come processo un'istanza di un programma in esecuzione. In Win32 un processo è uno spazio di indirizzamento di 4 GB e, a differenza di altri sistemi, è un entità inerte.**
- **Un processo Win32 non esegue nulla: dispone semplicemente di uno spazio di indirizzamento che contiene il codice e i dati di un eseguibile (il file EXE più tutte le DLL caricate da questo)**
- **Sono i thread le entità attive che eseguono il codice sorgente: quando viene creato un processo viene creato anche un thread primario che esegue il codice**
- **Tutti i thread del processo condividono lo stesso spazio di indirizzamento ma ogni thread possiede un proprio stack e un proprio insieme di registri di CPU**
- **Il thread primario può creare altri thread mediante la chiamata CreateThread(...) che restituisce un handle**
- **Ogni thread ha una priorità (compresa fra IDLE e REALTIME) e dd ogni thread attivo viene assegnata una porzione di tempo di CPU in base ad essa.**

Strumenti di sincronizzazione

- **Ad ogni handle di K.O. è associato un meccanismo di segnalazione (flag) che consente di realizzare meccanismi di sincronizzazione.**
- **Ogni handle possiede quindi due stati: *segnalato* e non *segnalato***
- **Esistono due funzioni di API: WaitForSingleObject e WaitForMultipleObjects che prendono come parametro un handle di K.O. e sospendono il thread chiamante fino a quando questo non diventa *segnalato***
- **Questo passaggio di stato avviene con modalità diverse per ogni K.O.**
- **Per esempio lo stato di un thread è *non segnalato* finchè il thread è in esecuzione e diventa *segnalato* quando questo termina: quindi se il Thread T1 chiama WaitForSingleObject passando come handle quello del Thread T2, T1 rimane sospeso fino al termine dell'esecuzione di T2**
- **Gli oggetti *evento* sono la forma più generica di sincronizzazione: segnalano semplicemente che un'operazione è terminata.**
- **Vengono creati con CreateEvent, portati allo stato *segnalato* con SetEvent o *non segnalato* con ResetEvent.**

Mutex e semafori

- **I mutex sono uno strumento per garantire l'accesso esclusivo ad una risorsa, sia all'interno del processo che fra processi diversi.**
- **I mutex vengono creati con CreateMutex assegnandogli un nome: se esiste già un mutex con lo stesso nome viene semplicemente incrementato un reference counter.**
- **Un thread che vuole l'accesso esclusivo chiama WaitForSingleObject sul mutex: se è segnalato prosegue, altrimenti viene sospeso.**
- **Quando il thread che ha ottenuto il controllo del mutex ha finito il suo lavoro chiama ReleaseMutex per metterlo in stato segnalato, riattivando quindi il primo thread in attesa**
- **I semafori consentono di gestire situazioni in cui esiste un numero limitato di risorse di un certo tipo. Ogni semaforo ha un contatore di risorse. Quando questo contatore è a zero il semaforo è *non segnalato*, quando è > 0 il semaforo è *segnalato.***
- **Il valore iniziale viene stabilito alla creazione (CreateSemaphore), viene decrementato con WaitForSingleObject e incrementato con ReleaseSemaphore.**

Oggetti stream

- **Come in UNIX anche in Win32 c'è una sostanziale uniformità fra file e dispositivi: entrambi vengono visti come oggetti kernel di tipo *stream***
- **Quindi sia un file che, per esempio, una porta seriale vengono aperti con CreateFile. ed è possibile leggere e scrivere su di essi utilizzando ReadFile e WriteFile**
- **Oltre a file e dispositivi è possibile operare in questo modo anche su strumenti di comunicazione come Sockets, Named Pipes e Mailslots.**
- **Un particolare oggetto stream è il *filemapping:* si tratta di uno spazio di memoria (dimensione max 2GB) condiviso fra due processi. E' uno strumento di comunicazione interprocesso all'interno di una singola macchina**
- **Sockets, Pipes e Mailslots consentono invece la comunicazione interprocesso sia all'interno di una macchina che fra macchine diverse su una rete.**
- **Le pipes sono uno strumento simile ai sockets ma più astratti (non sono collegate ad un particolare protocollo)**

Oggetti stream - I/O sincrono e asincrono

- **Su tutti i K.O di tipo stream è possibile agire sia in modalità sincrona che asincrona.**
- **In modalità sincrona le chiamate a WriteFile e ReadFile ritornano solo quando l'operazione è completata**
- **La modalità asincrona (chiamata anche overlapped) si basa sull'utilizzo di eventi che vengono segnalati quando un'operazione è stata terminata. In CreateFile è possibile specificare che l'apertura è in modalità asincrona ed indicare un evento associato.**
- **Quindi una chiamata a WriteFile ritorna immediatamente anche se l'operazione di scrittura richiede un certo tempo. Quando l'operazione termina viene segnalato l'evento associato**
- **Esistono poi eventi speciali, legati ai diversi tipi di oggetto: per esempio per una porta seriale abbiamo la funzione WaitCommEvent che permette di associare eventi all'arrivo di un carattere o al cambiamento di stato di un segnale di controllo (CTS, DSR ecc.)**
- **La modalità asincrona è implementata in modo completo solo su NT, mentre per 95/98 è limitata ai dispositivi di I/O e ai protocolli**