System and Device Programming

Windows API and Visual Studio

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Main Operating Systems

- Operating systems can be classified according to several criteria
  - Application domain
    - Mainframes, servers, workstations
    - Desktops, laptops
    - Embedded systems
    - Real-time devices
    - Handlet devices

Supercomputing, scientific calculus, services, web, etc

Everyday-life applications, e.g. cell-phones, automotive, etc.

Specific appllications, e.g., bar-code scanners, Personal Digital Assistant (PDA), smart cards, etc.

Safety critical applications, e.g., medical devices, avionics, etc.

Main Operating Systems

- Diffusion
  - Data update: September 2018

Microsoft Windows: 90.79%

All Devices	
SO	Market
Android	48.61%
iOS / OS X	11.04%
Windows	14.00%
Others	26.34%

Server	
SO	Market
Windows	49.50%
Apple	15.62%
Linux based	19.13%
Others	3.83%

Desktop, laptop, etc.	
SO	Market Share
Windows 7	47.21%
Windows 10	29.00%
Mac OS	6.35%
Windows 8.1	5.89%
Windos XP	5.69%
Linux	3.04%
FreeBSD	0.10%
Others	2.72%

Microsoft

- Microsoft
  - Founded in 1975 by Bill Gates e Paul Allen
  - First MS-DOS version in 1981
  - First Windows version in 1985
    - OS with graphical interface
    - Based on windows
    - Targeting Intel processors
  - Market share today
    - 90%, including all versions still in use
    - Windows 10 (29%), Windows 8 (6%), Windows 7 (47%), Windows XP (5%), Windows Vista (1%), etc.

Linux 3%, Mac OS 6%

Microsoft

Server	**Windows NT 3.1, 3.5, 3.51, 4.0 (from 1993), Windows 2000, Windows Server 2003, 2003 R2, 2008, 2008 R2, 2012, 2012 R2**
Device - embedded	**Windows CE, Windows Embedded, Windows Phone, Windows Mobile, Windows RT, …**
Desktop	**Windows 1.01-3.2 (from 1985 to 1993)** **Windows 95, 98, ME (Windows 9x) (from 1993)** **Windows XP (from 2001)** **Windows Vista (from 2007):** available with several flavours (home premium, business, enterprise, ultimate) **Windows 7 (from 2009):** available with several flavours (basic, premium, professional, enterprise, ultimate, thin PC) **Windows 8, 8.1 (from 2012):** available in several flavours (standard, pro, enterprise) **Windows 10 (from 2015)**

Microsoft

- 16-bit Versions
  - From Windows 1.0 (1985) to Windows 3.1 (1992)
    - Written to be portable on several architectures
    - Used mainly on Intel x86 processors
- 16/32-bit Versions
  - Windows 9x (1993-2000)
    - Derived from MS-DOS and 16-bit versions
    - New kernel
- 32/64-bit Versions
  - From Windows NT (NT = New Technology?)
    - Leave MS-DOS behind completely
    - New kernel (hybrid micro-kernel architecture derived from the UNIX system)

Operating System Standards

- Known standards
  - The C Library
  - Unix (Linux) Systems
  - Win32/Win64 or Windows
  - C++
- Different standards have different APIs
  - API = Application Programming Interface

The C language and UNIX are strictly connected as UNIX was developed in C

Linux is a free version of UNIX. The kernel identifies Linux's OS

The UNIX (Linux) Standards

ISO C	1972: UNIX is moved from assembre to C. Several versions are developed during the years: ANSI C (1989), ISO C o C90 (1990), ISO C o C99 (1999), ISO C11 (2011), ISO C18 (2018)
POSIX	POSIX = Portable Operating System Interface A family of standards born to facilitate the UNIX portability at a word-wide level. It defines the services each UNIX operating systems is suppose to deliver and to satisfy to be "POSIX compliant". It includes the ISO C standard.
SUS	SUS = Single UNIX Specification A project born in mid ’80, super-set of POSIX. It defines what an operating system has to do and how these things must be defined to be ”UNIX-like” operating system.

The Windows Standards

- Win32/Win64 (or simply Windows) API
  - Supported by Microsoft
    - Microsoft is the sole arbiter and implementor
  - Widely used by Windows operating systems
  - Has its own set of conventions and programming techniques, which are driven by Microsoft
  - Support different processors to be able to be ported under different architectures
  - Are defined in C language (**not** in C++)

The Windows Standards

- Go beyond standard C
  - Reduce code portability
  - Increase code functionality
  - For example
    - The C library is always available **but** we **cannot** fully exploit Windows with it
    - For example in C it is **not** possible to
      - Lock a file or a part of it
      - Mapping a file into main memory
      - Organize inter-process communication

In this unit, we concentrate on how to develop applications using the Windows API

Programming Principles

- Windows APIs
  - Are different from other standard (POSIX, etc.)
  - Require is own coding style and technique
  - Use threads (**not** processes) as basic unit of execution
    - A process can contain one or more threads
    - Each process has its own code and data address space
    - Threads share the process address space
    - Threads are “lightweight” and more efficient than processes

Programming Principles

- When programming in the Microsoft Visual Studio C++ environment, please remember to include
  - windows.h
  - This header includes most of the required data, such as
    - winnt.h
    - winbase.h
    - etc.

Programming Principles

- Windows is rich and flexible
  - Many functions perform the same or similar operations
    - There are sccasional artifacts from 16-bit Windows
      - Windows 32 was created from scratch but designed to be backward-compatible (with Windows 3.1 Win 16)
    - SetFilePointer, SetFilePointerEx, GetFileSize, GetFileSizeEx, etc.
  - Function names are long and descriptive
    - WaitForSingleObject, WaitForMultipleObjects, etc.

Programming Principles

- Each function has numerous parameters and flags
  - HANDLE fileHandle, LPVOID dataBuffer, DWORD numberOfByteToRead, LPDWORD numberOfByteRead, LPOVERLAPPED overlappedDataStructure
- Parameter and variable names are long and descriptive and often use "Hungarian" notation
  - lpFileName, lpBuffer, nNumberOfBytesToRead, etc.

Long Pointer
[to a zero terminated string]

- Symbolic constants and flags explain their meaning
  - INVALID_HANDLE_VALUE, GENERIC_READ, etc.

Examples

Read a file (text or binay format)

- API C style

```
size_t fread (void *ptr, size_t size,
  size_t nObj, FILE *fp);
```

- API POSIX style

```
int read (int fd, void *buffer, size_t nbytes);
```

- API Win32/64

```
BOOL ReadFile (
  HANDLE fileHandle,
  LPVOID dataBuffer,
  DWORD numberOfByteToRead,
  LPDWORD numberOfByteRead,
  LPOVERLAPPED overlappedDataStructure
);
```

Examples

Create a thread

- API POSIX style

```
int pthread_create (
  pthread_t *tid,
  const pthread_attr_t *attr,
  void *(*startRoutine)(void *),
  void *arg
);
```

- API Win32/64

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpsa,
  DWORD cbStack,
  LPTHREAD_START_ROUTINE lpStartAddr,
  LPVOID lpvThreadParm,
  DWORD dwCreate,
  LPDWORD lpIDThread
);
```

Programming Principles

- Nearly every resource is a **kernel object**
- **Objects** are identified and referenced by handle
- Handle objects are of type HANDLE

Similar to UNIX file descriptor or process id

  - HANDLE objects include
    - Files, pipes, processes, memory mapping, threads, events, mutexes, semaphores
  - HANDLEs are gray boxes
    - Kernel objects **must** be manipulated by Windows APIs

Programming Principles

- Specific names are reserved for Microsoft Visual C and the Microsoft compiler
  - _ keywordName
  - _functionName
- Functions
  - CloseHandle applies to (nearly) all objects
  - ReadFile, WriteFile, and many other return Boolean values
  - GetLastError returns system error codes

Programming Principles

- Predefined descriptive data types
  - Are expressed in upper case
    - BOOL, DWORD, LPDWORD, etc.
  - Avoid the ”*” operator and make (name) distinctions
    - LPTSTR
      - Long Pointer To STRing defined as TCHAR *
    - LPCTSTR
      - Long Pointer Constant To STRing defined as const TCHAR *

”LP” is obsolete and inconsistently used

”WIN32” appears in macro names even when the macro is for 64 bits

Examples

- Windows Data Types

See WEB sources (Windows Data Types) for a complete list

INT8	An 8-bit signed integer. This type is declared in BaseTsd.h as follows: typedef signed char INT8;
INT16	A 16-bit signed integer. This type is declared in BaseTsd.h as follows: typedef signed short INT16;
INT32	A 32-bit signed integer. The range is -2147483648 through 2147483647 decimal. This type is declared in BaseTsd.h as follows: typedef signed int INT32;
INT64	A 64-bit signed integer. The range is -9223372036854775808 through 9223372036854775807 decimal. This type is declared in BaseTsd.h as follows: typedef signed __int64 INT64;
...	...

UINT8	An unsigned **INT8**. This type is declared in BaseTsd.h as follows: typedef unsigned char UINT8;
UINT16	An unsigned **INT16**. This type is declared in BaseTsd.h as follows: typedef unsigned short UINT16;
UINT32	An unsigned **INT32**. The range is 0 through 4294967295 decimal. This type is declared in BaseTsd.h as follows: typedef unsigned int UINT32;
UINT64	An unsigned **INT64**. The range is 0 through 18446744073709551615 decimal. typedef usigned __int 64 UINT64;
ULONG	An unsigned **LONG**. The range is 0 through 4294967295 decimal. This type is declared in WinDef.h as follows: typedef unsigned long ULONG;
ULONGLONG	A 64-bit unsigned integer. The range is 0 through 18446744073709551615 decimal. This type is declared in WinNT.h as follows:

Coding Systems

- Windows supports executable code build in
  - 16 (Win16), 32 (Win32), 64 (Win64) bits
    - 16-bit versions are maintained only for backward compatibility
    - 32-bit versions run on 64-bit architecture but cannot exploit the larger address space
  - It is usually fairly simple to build applications able to run uder either Win32 and Win64
- Most of the difference concern the pointer size
  - Avoid any assumption about pointer length
  - Win64 uses 64-bit pointers
    - DWORD32, DWORD64, POINTER_32, POINTER_64, LONG32, LONG64, etc.

Character Coding Systems

- For characters and strings, four different coding strategies are possible
  - 8-bit only
  - Unicode only
  - 8-bit and Unicode with **generic** code
  - 8-bit and Unicode with **run-time** selection

This is generally the wiser and safer solution

Character Coding Systems

- 8-bit only
  - Use char (or CHAR) and C library such as printf, scanf, strcmp, etc.
- Unicode only
  - Use only 16-bit chars by defining proper variable (UNICODE and _UNICODE)
- 8-bit and Unicode with generic code
  - Use generic functions
  - These functions are automatically mapped on 8-bit or on the corresponding unicode functions
- 8-bit and Unicode with run-time selection
  - Use 8-bit or unicode functions on purpose
  - The selection is made by the programmer

Character Coding Systems

- To assure maximum flexibility and source portability
  - Define all characters and strings using generic type **TCHAR**
  - Calculate lengths using **sizeof(TCHAR)**
  - TCHAR is mapped on
    - ANSI ASCII coding when it is on 8-bits
      - char (or CHAR)
    - Unicode UTF-16 coding when it is mapped on 16-bits
      - WCHAR or (wchar_t)

String Coding Systems

- Constant strings are expressed in one of three forms
  - "This string uses **8-bit** characters"

    ANSI C
  - L"This string uses **16-bit** characters"

    ANSI C
  - _T("This string uses **generic** characters")

    A macro
    - Expands to "T..." if UNICODE is not defined
    - Expenad to L"T..." if UNICODE is defined
    - The **TEXT** macro is the same as **_T**

      .. coming ...
  - LPTSTR expands to either char * or wchar_t *

Selecting the Coding System

- To select the coding system
  - Include
    - #define UNICODE
      - To get WCHAR in all source modules
    - #undef UNICODE
      - To get CHAR
    - Be consistent
      - Define UNICODE before #include <windows.h>

Different Visual Studio versions may have different requirements

```
#ifdef UNICODE
  #define TCHAR WCHAR
#else
  #define TCHAR CHAR
#endif
```

Symbol definitions

The generic C library

- To make available a wide class of string processing and I/O functions
  - Include
    - #define _UNICODE
  - Consistently with UNICODE
  - This enables functions such as
    - _fgettc, _itot, _ttoi, _totupper, _totlower
    - And many more, nearly the complete library
  - Also, locale-specific functions (seven in all)
    - lstrlen, lstrcmp, lstrcpy, lstrcat, …

The generic C library

- To get generic C library text macros and functions
  - After windows.h, include tchar.h, i.e.,
    - #include <tchar.h>
  - Use the **generic** C library for all string functions

_tprintf	in place of printf
_stprintf	in place of sprintf
_tcslen	in place of strlen
_itot	in place of itoa
… and MANY more	

  - Generic versions of some functions are not provided (e.g., memchr)

Examples

```
#ifndef _TCHAR_H_
#define _TCHAR_H_
...
#ifdef _UNICODE
/* Unicode functions */
#define _tprintf wprintf
#define _ftprintf fwprintf
#define _stprintf swprintf
...
#else
/* Non-unicode (standard) functions */
#define _tprintf printf
#define _ftprintf fprintf
#define _stprintf sprintf
...
#endif
#endif
```

tchar.h

Windows standard library functions

ANSI C standard library functions

Main Program Definition

- Pay attention on how the main header is defined
  - int main (int argc, char * argv[])
    - Is for 8-bit characters only
  - int main (int argc, w_char * argv[])
    - 8-bit definition header but with wide-characters
    - ASCII is no entirely accurate but it is used sometimes
  - int wmain (int argc, w_char *argv[])
    - Is for Unicode characters only
  - int _tmain (int argc, LPTSTR argv[]) {
    - Expands to **main** or **wmain** depending on definition of _UNICODE
    - This assures correct operations in all combinations

Examples

Generic code (mapped on 16-bits)

Remove Warnings from projects in Visual Studio

C library and macros available

Windows and standard C libraries

Generic header (mapped on 8 or 16 bits)

```
#define UNICODE
#define _UNICODE

#define _CRT_SECURE_NO_WARNINGS

#include <windows.h>
#include <tchar.h>
...

int _tmain (int argc, LPTSTR argv[]) {
...
}
```

One Example: Copy a File

- Write a simple C program copying an input file into an output file
  - Input and output file names are passed to the program on the command line
  - The file can be in **binary** or in text form

Functions such as **fscanf** cannot be used

    - If in text form can be in ASCII or UNICODE format
  - Check and debug the program on the Visual Studio environment

One Example: Copy a File

- Write 4 versions of the program, using
  - The standard C library
  - The UNIX library
  - The Windows API
  - The Windows API with **convenience** functions

Windows API is reach and many operations can be done in several different ways

C Library Implementation

```
#include <stdio.h>

#define N 256

int main (int argc, char *argv []) {
  FILE *fpIn, *fpOut;
  char str[N];
  size_t nIn, nOut;

  fpIn  = fopen (argv[1], "rb");
  fpOut = fopen (argv[2], "wb");

  if (fpIn==NULL || fpOut==NULL) {
     printf (″Error opening file.\n″);
     return 1;
  }
```

Can also read single bytes
(avoided for the sake of efficiency)

'b': Binary mode
(no meaning for UNIX)

C Library Implementation

Number of objects read or written

Parameters:
Data structure pointer, Size of the structure, Number of elements, File pointer

```
  while ((nIn = fread (str, 1, N, fpIn)) > 0) {
    nOut = fwrite (str, 1, nIn, fpOut);
    if (nOut!=nIn) {
      printf ("I/O Error.\n");
      return 2;
    }
  }

  fclose (fpIn);
  fclose (fpOut);

  return 0;
}
```

Synchronous I/O (wait to terminate)
No file security control
No file locking

UNIX Implementation

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>

#define N 256

int main (int argc, char *argv []) {
  int fdIn, fdOut;
  ssize_t nIn, nOut;
  char str[N];

  fdIn = open(argv[1], O_RDONLY);
  fdOut = open(argv[2], O_WRONLY|O_CREAT, 0666);
  if (fdIn==-1 || fdOut==-1) {
    printf ("Error opening file.\n");
    return 1;
  }
```

Return file descriptor or -1 on error

Parameters: Pathname, OR-ing constant from fnctl.h, Access mode

UNIX Implementation

Number of objects read or written

```
  while ((nIn = read (fdIn, str, N)) > 0) {
    nOut = write (fdOut, str, (size_t) nIn);
    if (nOut!=nIn) {
      printf (”I/O Error.\n”);
      return 2;
    }
  }

  close (fdIn);
  close (fdOut);

  return 0;
}
```

Parameters: File descriptor, Data structure pointer, Number of elements

Windows Implementation 1

For now:
NO #define UNICODE, #define _UNICODE,
#include <tchar.h>, etc.

```
#include <windows.h>
#include <stdio.h>

#define N 256


INT main (INT argc, LPTSTR argv []) {
  HANDLE hIn, hOut;
  DWORD nIn, nOut;
  CHAR str[N];
```

Standard main header

Windows API standard types

Windows Implementation 1

Parameters: File name, Access type, Share mode, Security attribute, Creation mode Flags, Template

```
hIn = CreateFile (argv[1], GENERIC_READ,
  0, NULL, OPEN_EXISTING,
  FILE_ATTRIBUTE_NORMAL, NULL);
hOut = CreateFile (argv[2], GENERIC_WRITE,
  0, NULL, CREATE_ALWAYS,
  FILE_ATTRIBUTE_NORMAL, NULL);
if (hIn==INVALID_HANDLE_VALUE ||
  hOut==INVALID_HANDLE_VALUE) {
  printf ("Error opening file.\n");
  return 1;
}
```

Using standard C library

Windows Implementation 1

```
  while (
    ReadFile (hIn, str, N, &nIn, NULL) && nIn > 0) {
    WriteFile (hOut, str, nIn, &nOut, NULL);
  }

  CloseHandle (hIn);
  CloseHandle (hOut);

  return 0;
}
```

Parameters: similar to C & UNIX
Error code is returned, number of bytes is a parameter

Windows Implementation 2

```
#include <windows.h>
#include <stdio.h>

int main (int argc, LPTSTR argv []) {
  if (!CopyFile (argv[1], argv[2], FALSE)) {
    printf (″Error opening file.\n″);
    return 1;
  }
  return 0;
}
```

Convenience function
Parameters: Input file name, Output file name, Overwrite or not (yes iff FALSE)