C,C++ Window system

**From my home**

**Thu Duc district, Sai Gon City**

**St 43**

Linh La

Every thing in this document is structure for easy selfstudy and reference. The object can be beginner or programmer. Hope this document help you much in programming.

Contents

[I. Lập trình hệ thống windows: 1](#_Toc470910731)

[1. Giới thiệu 1](#_Toc470910732)

[2. Window File và đặc tính IO: 4](#_Toc470910733)

[3. Ký tự trên WinAPI 25](#_Toc470910734)

[4. Date time trên WinAPI 32](#_Toc470910735)

[5. Quản lý file, thư mục 36](#_Toc470910736)

[6. Quản lý Registry 57](#_Toc470910737)

[7. Quản lý vùng nhớ, Memory-map file 65](#_Toc470910738)

[8. Process và thread: 65](#_Toc470910739)

[II. Library and Dictionary: 75](#_Toc470910740)

## Lập trình hệ thống windows:

### Giới thiệu

1. Trình quản lý bộ nhớ CPU - MMU

CPU hiện đại có bộ quản lý bộ nhớ (memory management unit – MMU) để ngăn bất kỳ quá trình nào vi phạm không gian bộ nhớ của quá trình khác. Chuyển từ một quá trình này sang quá trình khác – được gọi là chuyển ngữ cảnh – có nghĩa là lập trình lại MMU để chỉ đến không gian địa chỉ khác cùng với việc lưu và phục hồi thông tin của một quá trình.

1. Hệ điều hành đa nhiệm

Hệ điều hành chịu trách nhiệm quản lý chi tiết của việc chuyển ngữ cảnh và nó cũng tiêu tốn hiệu suất của CPU. Do mỗi quá trình đều được cách ly với những quá trình khác, giao tiếp giữa các quá trình đòi hỏi phải có những chức năng đặc biệt. Tương tự việc chuyển ngữ cảnh, giao tiếp giữa các quá trình cũng chiếm thời gian của bộ xử lý.

Tất cả thời gian xử lí của CPU tăng lên khi nhiều chương trình chạy cùng lúc và khi có nhiều người dùng.

Càng nhiều quá trình chạy thì càng tốn nhiều tài nguyên của CPU để thực hiện công việc chuyển ngữ cảnh.

Nếu số quá trình quá nhiều, máy chủ có thể phải dành toàn bộ thời gian để thực hiện việc chuyển đổi giữa các quá trình mà không thể thực sự xử lý được công việc nào.

Để tránh tình trạng trên, lập trình viên có thể dùng luồng (thread). Luồng cũng giống một quá trình con, ngoại trừ đặc điểm là mọi luồng kết hợp với một quá trình nào đó chia sẻ cùng không gian địa chỉ.

Ví dụ, nếu có nhiều người dùng cùng chương trình, lập trình viên có thể viết ứng dụng sao cho ứng với mỗi người dùng sẽ có một luồng mới được tạo ra.

Mỗi luồng có tiến trình kiểm soát riêng nhưng nó lại chia sẻ cùng không gian địa chỉ và hầu hết dữ liệu với tất cả luồng khác chạy trong cùng quá trình. Đối với từng người dùng có thể nói rằng chương trình dường như chỉ chạy cho một mình họ.

**Ưu điểm là gì?**

Thời gian chuyển đổi giữa các luồng cùng một quá trình ít hơn hẳn so với các quá trình khác vì không cần phải chuyển đổi không gian địa chỉ. Ngoài ra, vì chúng chia sẻ không gian địa chỉ nên các luồng trong một quá trình có thể giao tiếp với nhau dễ dàng hơn nhiều.

Trên máy tính có nhiều bộ xử lý, chương trình dạng một quá trình đơn chỉ chạy trên một CPU, còn chương trình dạng luồng có thể chia các luồng cho tất cả các bộ xử lý. Vì thế, nếu bạn chuyển chương trình dạng luồng sang máy chủ nhiều bộ xử lý thì nó sẽ chạy nhanh hơn.

**Khuyết điểm?**

Chương trình dạng luồng khó viết và kiểm lỗi hơn. Không phải mọi thư viện lập trình đều được thiết kế để dùng với luồng và không phải mọi ứng dụng cũ đều tương thích với môi trường phân luồng. Một vài công cụ lập trình cũng làm cho việc thiết kế và thử nghiệm mã luồng khó khăn hơn.

Lỗi liên quan đến luồng cũng khó phát hiện hơn. Các luồng trong một quá trình có thể bị chồng chéo dữ liệu với nhau. Hệ điều hành có thể hạn chế số luồng thực thi chẳng hạn đọc và ghi dữ liệu cùng lúc. Việc định thời cho các luồng khác nhau để tránh xung đột là rất khó khăn.

Tuy nhiên, khi các ứng dụng dùng chung phức tạp và máy chủ nhiều bộ xử lý ngày càng phổ biến thì luồng sẽ ngày càng được dùng nhiều để thực hiện đa xử lý.

1. Thành phần quan trọng của OS windows:

Phần lõi hệ điều hành Windows được phổ biến tới nhiều loại thiết bị như điện thoại, laptop, máy tính bảng và server.

Chúng ta xem xét những thành phần quan trọng màhệ điều hành windows quản lý:

• Bộ nhớ: OS quản lý 1 lượng lớn bộ nhớ hay vùng không gian địa chỉ và giúp di chuyển thong tin xuyên suốt giữa các loại bộ nhớ như RAM, ổ đĩa ….

• Hệ thống quản lý file (File systems) : HDH quản lý không gian tên và cây phân cấp tập tin, cung cấp khả năng truy xuất tuần tự và truy xuất trực tiếp trên thư mục và file.

•Bộ xử lí trung tâm: HDH phân phối hiệu quả các tác vụ tính toán cho bộ xử lí hay nhiều bộ xử lí để tang tốc, thậm chí tên cả các dòng máy tính nhỏ.

• Cung cấp khả năng đánh tên và định vị: Đánh tên file quy định độ dài, mô tả. Tiến trình đánh tên được mở rộng thành hướng đối tượng (object of naming scheme) như đối tượng các thiết bị, đối tượng đồng bộ và các đối tượng giao tiếp xử lí bên trong.

HDH giúp định vị và quản lý và truy xuất đối tượng tên (name object)

•Đa xử lí: HDH quản lí các tiến trình processe, threads, và các đơn vị khác 1 cách độc lập. Các tasks có thể bị tiếm quyền ưu tiên và sắp xếp theo sự ưu tiên được tính toán linh hoạt.

• Giao tiếp và đồng bộ hóa: HDH quản lý giao tiếp giữa các process và đồng bộ trong 1 mấy tính cũng như nhiều máy tính trên mạng.

• Bảo mật và mã hóa: HDH cung cấp cơ chế mềm dẻo để bảo vệ tài nguyên từ các nguồn truy cập không rõ rang.

Window API hỗ trợ cho bạn lập trình tương tác với tất cả thành phần trên

1. Windows NT5 and NT6

Microsoft dùng thuật ngữ NT để phân biệt giữa các phiên bản windows. Dòng Windows 2000, XP, và Server 2003 dùng phần lõi NT kernel version 5 .

Ví dụ, window XP dùng NT 5.1.2600. chỉ số 2500 là build number.

Một số thành phần API phụ thuộc vào phiên bản kernel, nên dùng thuật ngữ NT5 để định nghĩa những version window rất thuận tiện, dù Microsoft không còn dùng thuật ngữ này nữa.

NT6 kernel là nền tảng của Windows 7 (6.1), Vista (6.0), và Server 2008 (6.1 cho R2 và 6.0 cho loại khác).

Khi nhắc tới NT6, ta sẽ hiểu là những phiên bản window này.

1. Standar C hay WinAPI

Lập trình viên Window luôn phải quyết định là lựa chọn xử lí file với standard C hay Window API.

Standard C có vài thuận lợi:

* Code tương thích với hệ thống khác (portable)
* Các hàm xử lí chuỗi thuận tiện mà Window không có
* Đa số là các hàm high level, dễ sử dụng hơn WinAPI
* Hàm xử lí line và stream dễ thay đổi sang dạng tổng quát, dù mất đi khả năng tương thích hệ thống.

Những giới hạn của C là:

* Không thể quản lý file và thư mục
* Chỉ xử lí được con trỏ file 32-bits.
* Không dùng được tính năng cao cấp như bảo mật, file ánh xạ vùng nhớ, khóa file, Async I/O, giao tiếp nội bộ giữa các process.

Ngoài ra, chúng ta có thể chuyển code từ UNIX sang Window nhờ các thư viện tương thích.

Microsoft C cung thư viện gồm các hàm tương ứng với hàm trên UNIX.

Thư viện Microsoft UNIX bao gồm các hàm I/O, nhưng bỏ qua hầu hết việc quản lý process và các API khác.

Tên hàm có tiền tố “\_”, ví dụ: \_read, \_write, \_stat …

Việc quết định dùng C, thư viện tương thích, hay Win API nên xét theo đặc tính của project.

1. Quy cách tên file, path và namespace
2. Quy cách đặt tên

* Đường dẫn đầy đủ hay thủ tục đặt tên quốc tế: bắt đầu bằng “\\”, ám chỉ root toàn cục, theo sau là tên server – đường dẫn tài nguyên mạng chia sẻ
* Dấu phân cách giữa thư mục là “\”, Các hàm API window dùng dấu “/” hay “\\”
* Tên file không chứa kí tự ASCII từ 1->31 như: < > ? : “ \*\ / |
* Tên thư mục và file phân biệt hoa thường. API cho phép tên dài tới 255 kí tự, đường dẫn có giới hạn (Marco MAX\_PATH qui định 260 kí tự), chúng ta sẽ dùng kí tự thoát đặc biệt để rút gọn.
* Phiên bản Unicode mở rộng tới 32,767 kí tự, nhưng phải đặt chuỗi "\\?\" vào trước đường dẫn.
* Dấu “.”phân cách giữa tên file và phần mở rộng
* Dấu “.” Trong tên thư mục ám chỉ thư mục hiện tại
* Dấu “..” trong tên thư mục ám chỉ thư mục cha của thư mục hiện tại.

1. Tên ngắn và dài

Tên file ngắn theo chuẩn MS\_DOS 8.3 (8 kí tự tên – 3 kí tự phần mở rộng)

Tên file dài được Window lưu trên đĩa cứng như một thư mục đặc biệt, đồng thời Window cũng sẽ tạo tên bí danh 8.3 cho file, lưu trên đĩa.

Window Vista trở về trước: 8.3 aliasing không thể kích hoạt hay vô hiệu hóa trên Volume, cho tới khi có Windows 7 và Windows Server 2008 R2.

Trên 1 số file system, tiền tố (~) đặt trước tên file khi dài hơn luật 8.3 naming.

Dùng GetShortPathName để lấy tên file 8.3, và dùng GetLongPathName để lấy tên file dài

GetFullPathName thường để lấy đường dẫn đầy đủ đối với relative path.

Các fily system NTFS, exFAT, UDFS, và FAT32, Windows lưu tên file dài dạng Unicode, có thể sao chép di chuyển tên file dài qua lại giữa NTFS và FAT.

Nhưng với MS-DOS FAT và vài dạng CDFS (CD-ROM), chỉ hỗ trợ 8.3 file

1. Đường dẫn

Ngoài tên file, thư mục, dấu cách (/), đường dẫn có thể chứa các tiền tố, biến môi trường, kí tự đặc biệt mà hệ thống có thể phiên dịch.

Các tiền tố giúp xác định namespace trong path.

Mỗi yếu tố của path có độ dài tối đa qui định bởi file system, luật này gồm 2 mệnh đề là dài và ngắn

Thư mục cũng chỉ là dạng file đặc biệt, cũng chịu luật tên file.

1. Fully Qualified vs. Relative Paths

Các WinAPI xử lí file, tên file thường định vị từ thư mục gốc, một số API yêu cầu đường dẫn đầy đủ từ thư mục gốc

* Tên UNC thì luôn bắt đầu với (“\\”), như “\\server\_name\share\_folder\file”
* Tên là ký tự ổ đĩa và dấu (:) như (“C:tmp.txt”) thì phiên dịch thành “tmp.txt” trên thư mục hiện hành trên ổ C  
  "C:tempdir\tmp.txt" ám chỉ tới file trong thư mục con tempdir của thư mục hiện hành trên ổ C.
* Kí hiệu “..” tới thự mục cha như "..\tmp.txt" hay "..\..\tmp.txt", "..\tempdir\tmp.txt"

Thỉnh thoảng có sự kết hợp như "C:..\tmp.txt": hữu dụng bởi vì hệ thống đều giám sát thư mục hiện hành của ứng dụng trên mỗi ổ đĩa.

1. Giới hạn độ dài Path

Qui định bởi MAX\_PATH, 260 kí tự. Ví dụ "D:\some 256-character path string<NUL>"

File I/O trong Windows API tự chuyển "/" thành "\.

Các API Unicode cho phép path dài tới 32,767 kí tự.

Tham số lpMaximumComponentLength từ GetVolumeInformation cho biết path dài nhất trên Volume là bao nhiêu.

Khi tham số path nhận chuỗi có tiền tố “\\?\”, hàm sẽ hiểu là không áp dụng luật kí tự tối đa, ví dụ "\\?\D:\very long path". Tối đa cho phép là 32,767 kí tự.

Tiền tố “\\?\” có thể dùng cho UNC path, như "\\?\UNC\server\share".

Tiền tố “\\?\” không dùng cho relative path (biến môi trường), relative path vẫn giới hạn bởi MAX\_PATH

Khi tạo thư mục, tên không được vượt qua MAX\_PATH – 12.

Kể từ Windows 10, version 1607, MAX\_PATH bị bỏ, tuy nhiên chúng ta vẫn phải định nghĩa và áp dụng luật này.

1. Namespace

NT namespace được thiết kế là namespace cấp thấp trên các hệ thống nhỏ, gồm Win32 subsystem. POSIX là một ví dụ về subsystem trong Window

Win32 namespaces là phần mở rộng thêm cho NT namspace

1. Không gian tên File

Với file I/O, tiền tố "\\?\" báo cho Win API không parsing chuỗi và gửi chuỗi này tới file system xử lí.

Ví dụ nếu file system hỗ trợ tên file và path dài, có thể bỏ qua giới hạn MAX\_PATH, bằng cách chèn tiền tố này.

Đồng thời tiền tố “\\?\” còn tắt tính năng tự mở rộng path, như khi dùng “..” và “.” Trong path.

Nếu không dùng tiền tố “\\?\”, có thể dùng relative path trong đường dẫn đầy đủ.

Lưu ý: một số file I/O API không hỗ trợ tiền tố này. Chỉ dùng cho Windows API , không áp dụng cho Windows shell applications như Windows Explorer.

1. Không gian tên thiết bị

Tiền tố “\\.\” , cho phép truy xuất tới nhiều thiết bị trực tiếp không qua file system, như khi truy xuất đĩa cứng vật lý và volumes trực tiếp, sử dụng CreateFile và DefineDosDevice.

Ví dụ: Nếu muốn truy xuất cổng nối tiếp port 1, dùng “COM1” cho tham số CreateFile mà không cần tiền tố “\\,\”, vì COM1–COM9 là tên dành riêng trong không gian tên NT.

Nếu bạn có 100 cổng nối tiếp và muốn mở COM56, bạn phải dùng “\\.\COM56”, vì COM56 không định nghĩa trong không gian tên NT

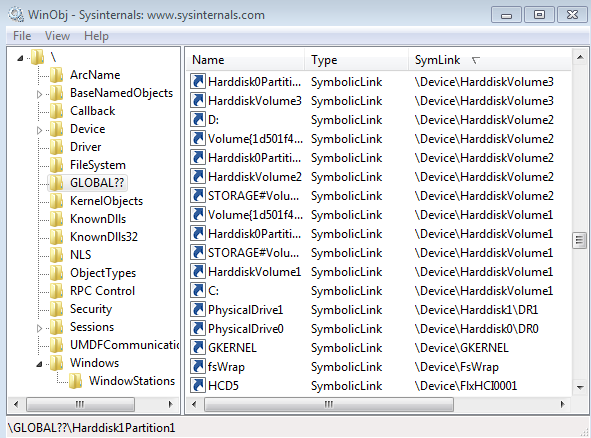
Với ổ cứng: dùng "\\.\PhysicalDiskX", với “X” là số thứ tự hợp lệ giá trị từ “0” trở lên,

Với ổ CD dùng "\\.\CdRomX", với “X” là số thứ tự hợp lệ giá trị từ “0” trở lên.

Hầu hết WinAPI không áp dụng “\\.\”, vì chúng không làm việc trực tiếp với thiết bị.

1. NT Namespaces

Một số API hỗ trợ NT namespace. Chúng ta dùng Windows Sysinternals WinObj tool để hình dung, NT namespace bắt đầu ở root “\”



Thư mục con "Global??" chứa không gian tên Win32. Chúng ta có thể tìm thấy Serial0 và Serial1 tương ứng với 2 cổng COM.

Đối tượng Volume “C:” có tên là “HarddiskVolume1”.

Đối tượng "Physicaldrive0" có tên dưới thư mục con DR là “Harddisk0\DR0”

Nếu không có symbolic link, các thiết bị không thể truy xuất bởi ứng dụng dùng Win32 namespace.

Tuy nhiên, handle có thể tạo ra dùng NT namespace như “\Device\Xxx”.

Để hỗ trợ nhiều user qua Terminal Services và virtual machines, Window ảo hóa thiết bị root vào Win32 namespace, thông qua tên “GLOBALROOT”, truy xuất qua path “\\.\GLOBALROOT”.

Chúng ta dùng tiền tố “\\.\GLOBALROOT” đảm bảo path truy xuất từ root của System object manager, chứ không phải path session.

### Window File và đặc tính IO:

1. Hệ thống quản lý file

Window phát triển 5 hệ thống quản lý file:

* CD-ROM file system (CDFS): quản lý file trên CD theo chuẩn ISO 9660
* FAT và F32: 2 hệ thống quản lý file cũ. FAT không hỗ trợ nhiều về bảo mật và chỉ quản lý dung lượng đĩa thấp hơn so với FAT32
* NT file system (NTFS): hệ thống quản lý mới hơn, hỗ trợ tên file dài, bảo mật tốt hơn, hỗ trợ dung lượng cực lớn.
* Universal Disk Format (UDF): chuẩn công nhiệp cho DVD.
* Networked File System (NFS) và Common Internet File System (CIFS): hệ thống quản lý phân tán của window server

1. Handle và Object

Object là dữ liệu trừu tượng đại diện cho tài nguyên hệ thống (file, thread, hình ảnh…)

Ứng dụng phải thông qua Object Handle để truy xuất Object ( dữ liệu đối tượng hay tài nguyên hệ thống.)

Window phân giải, ánh xạ Object Handle như một con trỏ tới vùng nhớ, giải phóng các Handle đồng nghĩa với việc giải phóng tài nguyên tương ứng.

Window duy trì một bảng quản lý các entry Object Handle, entry dùng địa chỉ tài nguyên để nhận diện kiểu tài nguyên

Mục đích của việc dùng Object và Handles để truy xuất vì 2 lý do chính:

* Tính đồng bộ và ổn định: khi phiên bản Window mới phát hành, chúng ta không phải bận tâm về thay đổi bên trong của Object.
* Tính bảo mật của Window: Window kiểm soát sự truy xuất tới Object của ứng dụng thông qua danh sách kiểm soát truy xuất (ACL- access control list).

Hệ thống kiểm tra ALC mỗi lần ứng dụng tạo Object Handle.

Chúng ta nên phân biệt Object Handle của tài nguyên với đối tượng handle HWND của cửa sổ window.

1. Tạo Object Handle

Cú pháp:

HANDLE WINAPI CreateFile(

\_In\_ LPCTSTR lpFileName,

\_In\_ DWORD dwDesiredAccess,

\_In\_ DWORD dwShareMode,

\_In\_opt\_ LPSECURITY\_ATTRIBUTES lpSecurityAttributes,

\_In\_ DWORD dwCreationDisposition,

\_In\_ DWORD dwFlagsAndAttributes,

\_In\_opt\_ HANDLE hTemplateFile

);

* pFileName [in]: Đường dẫn file, có thể dùng dấu phân cách “/” hay “\”. Phiên bản ANSI giới hạn số kí tự tối đa qui định bởi Macro MAX\_PATH.

Phiên bản UNICODE cho phép mở rộng tới 32.767 kí tự

* dwDesiredAccess [in]: qui định chế độ tạo file

|  |  |
| --- | --- |
| **Mode** | **Macro** |
| Đọc | **GENERIC\_READ** |
| Ghi | **GENERIC\_WRITE** |
| Vừa ghi vừa đọc | **GENERIC\_READ | GENERIC\_WRITE** |

Tham khảo thêm tại: [Generic Access Rights](https://msdn.microsoft.com/en-us/library/windows/desktop/aa446632(v=vs.85).aspx), [File Security and Access Rights](https://msdn.microsoft.com/en-us/library/windows/desktop/aa364399(v=vs.85).aspx), [**File Access Rights Constants**](https://msdn.microsoft.com/en-us/library/windows/desktop/gg258116(v=vs.85).aspx), và [**ACCESS\_MASK**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa374892(v=vs.85).aspx)

Tham số “zero”, hệ thống sẽ chỉ tham chiếu thuộc tính Object mà không đụng tới đối tượng.

* dwShareMode [in]: chế độ chia sẻ file hay thiết bị

Chế độ truy xuất không ảnh hưởng tới chế độ chia sẻ.

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| **0**  0x00000000 | Không chia sẻ. Ngăn tiến trình mở file khác xóa, đọc hay ghi. |
| **FILE\_SHARE\_DELETE**  0x00000004 | Cho phép tiến trình khác được xóa. Sau đó, không thể mở file được nữa.  Lưu ý: Cho phép xóa cũng cho phép đổi tên |
| **FILE\_SHARE\_READ**  0x00000001 | Cho phép tiến trình khác đọc |
| **FILE\_SHARE\_WRITE**  0x00000002 | Cho phép tiến trình khác được ghi |

Nếu để tham số “zero”, tiến trình khác chỉ truy xuất được khi Handle Object hiện tại đã đóng.

Chế độ truy xuất và chế độ chia sẻ không được xung đột với nhau, xem thêm [Creating and Opening Files](https://msdn.microsoft.com/en-us/library/windows/desktop/aa363874(v=vs.85).aspx). Nếu xung đột, **GetLastError**() trả về **ERROR\_SHARING\_VIOLATION**

* lpSecurityAttributes [in, optional]: Nên để NULL.

Đây là con trỏ cấu trúc **SECURITY\_ATTRIBUTES**, gồm mã số nhận dạng bảo mật **SECURITY\_DESCRIPTOR** và logic flag **bInheritHandle** qui định Handle trả về có được truy xuất bởi tiến trình con hay không.

Nếu để **NULL**,Handle không được dùng bởi tiến trình con của ứng dụng, file được set mã số bảo mật mặc định

Nếu trong cấu trúc bảo mật **SECURITY\_DESCRIPTOR** = **NULL**, Object được set mã số bảo mật mặc định

Nếu Object có tồn tại : CreateFile bỏ qua tham số **SECURITY\_DESCRIPTOR**.

* dwCreationDisposition [in]: Đối với thiết bị không phải file, mặc định là **OPEN\_EXISTING**

Flag này không được phép kết hợp toán tử OR, gồm :

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| **CREATE\_ALWAYS**  2 | Luôn tạo file mới. Nếu file tồn tại, ghi dè |
| **CREATE\_NEW**  1 | Chỉ tạo file mới khi không có |
| **OPEN\_ALWAYS**  4 | Luôn mở file. Nếu file không tồn tại sẽ báo lỗi |
| **OPEN\_EXISTING**  3 | Mở nếu file có tồn tại. Nếu file không tồn tại báo lỗi **ERROR\_FILE\_NOT\_FOUND** (2). |
| **TRUNCATE\_EXISTING**  5 | Mở và cắt file xuống còn 0 byte dung lượng nếu file tồn tại. Nếu file không tồn tại, báo lỗi **ERROR\_FILE\_NOT\_FOUND** (2). |

* dwFlagsAndAttributes [in]: qui định thuộc tính và flag.

Thuộc tính mặc định là **FILE\_ATTRIBUTE\_NORMAL** cho file, có thể kết hợp nhiều thuộc tính (FILE\_ATTRIBUTE\_\*).

|  |  |
| --- | --- |
| **Attribute** | **Meaning** |
| **FILE\_ATTRIBUTE\_ARCHIVE**  32 (0x20) | Tập tin này là file nén |
| **FILE\_ATTRIBUTE\_ENCRYPTED**  16384 (0x4000) | File mã hóa. Đối với thư mục, các file con được mã hóa khi tạo.  Flag không hỗ trợ trên bản Home, Home Premium, Starter, hay ARM của Window |
| **FILE\_ATTRIBUTE\_HIDDEN**  2 (0x2) | File ẩn |
| **FILE\_ATTRIBUTE\_NORMAL**  128 (0x80) | Mặc định của File |
| **FILE\_ATTRIBUTE\_OFFLINE**  4096 (0x1000) | The data of a file is not immediately available. This attribute indicates that file data is physically moved to offline storage. This attribute is used by Remote Storage, the hierarchical storage management software. Applications should not arbitrarily change this attribute. |
| **FILE\_ATTRIBUTE\_READONLY**  1 (0x1) | File chỉ đọc. Không thể ghi hay xóa. |
| **FILE\_ATTRIBUTE\_SYSTEM**  4 (0x4) | File thuộc về hệ thống |
| **FILE\_ATTRIBUTE\_TEMPORARY**  256 (0x100) | File tạm tạo bởi hệ thống  Xem thêm [Caching Behavior](https://msdn.microsoft.com/en-us/library/windows/desktop/aa363858(v=vs.85).aspx#caching_behavior) . |

 Flag có thể kết hợp nhiều Macro (FILE\_FLAG\_\*), qui định cách caching của Object, chế độ truy xuất…

|  |  |
| --- | --- |
| **Flag** | **Meaning** |
| **FILE\_FLAG\_BACKUP\_SEMANTICS**  0x02000000 | File tạo ra cho việc sao lưu, phục hồi hệ thống. Hệ thống cho phép tiến trình vượt qua hàng rào bảo mật với quyền ưu tiên **SE\_BACKUP\_NAME**  và **SE\_RESTORE\_NAME**.  Tham khảo [Changing Privileges in a Token](https://msdn.microsoft.com/en-us/library/windows/desktop/ms717797(v=vs.85).aspx).  Flag này cần thiết cho Handle thư mục. |
| **FILE\_FLAG\_DELETE\_ON\_CLOSE**  0x04000000 | File sẽ bị xóa ngay khi Handle đóng lại, bao gồm luôn các Handle bản sao.  Nếu có tiến trình khác đang mở, việc xóa sẽ thất bại. Trừ khi chế độ chia sẻ **FILE\_SHARE\_DELETE** được bật cho tất cả. |
| **FILE\_FLAG\_NO\_BUFFERING**  0x20000000 | Object mở mà không caching cho dữ liệu đọc và ghi.  Flag này không ảnh hưởng tới caching đĩa cứng hay file mapped-memory.  Tham khảo thêm [File Buffering](https://msdn.microsoft.com/en-us/library/windows/desktop/cc644950(v=vs.85).aspx). |
| **FILE\_FLAG\_OPEN\_NO\_RECALL**  0x00100000 | Dùng cho việc truy xuất từ xa. Object không nên lưu lại trên local. |
| **FILE\_FLAG\_OPEN\_REPARSE\_POINT**  0x00200000 | Dùng để mở object có reparse point, bất chấp có tìm thấy bộ lọc hay không.  Không dùng chung với flag **CREATE\_ALWAYS**  Nếu object không có reparse point, flag này bị bỏ qua. |
| **FILE\_FLAG\_OVERLAPPED**  0x40000000 | Object dùng cho Async I/O. Nếu flag không set, I/O diễn ra tuần tự  Tham khảo [Synchronous and Asynchronous I/O Handles](https://msdn.microsoft.com/en-us/library/windows/desktop/aa363858(v=vs.85).aspx#synchronous_and_asynchronous_i_o_handles) |
| **FILE\_FLAG\_POSIX\_SEMANTICS**  0x0100000 | Access will occur according to POSIX rules. This includes allowing multiple files with names, differing only in case, for file systems that support that naming. Use care when using this option, because files created with this flag may not be accessible by applications that are written for MS-DOS or 16-bit Windows. |
| **FILE\_FLAG\_RANDOM\_ACCESS**  0x10000000 | Access is intended to be random. The system can use this as a hint to optimize file caching.  This flag has no effect if the file system does not support cached I/O and **FILE\_FLAG\_NO\_BUFFERING**.  For more information, see the [Caching Behavior](https://msdn.microsoft.com/en-us/library/windows/desktop/aa363858(v=vs.85).aspx#caching_behavior) section of this topic. |
| **FILE\_FLAG\_SESSION\_AWARE**  0x00800000 | The file or device is being opened with session awareness. If this flag is not specified, then per-session devices (such as a device using RemoteFX USB Redirection) cannot be opened by processes running in session 0. This flag has no effect for callers not in session 0. This flag is supported only on server editions of Windows.  **Windows Server 2008 R2 and Windows Server 2008:**This flag is not supported before Windows Server 2012. |
| **FILE\_FLAG\_SEQUENTIAL\_SCAN**  0x08000000 | Access is intended to be sequential from beginning to end. The system can use this as a hint to optimize file caching.  This flag should not be used if read-behind (that is, reverse scans) will be used.  This flag has no effect if the file system does not support cached I/O and **FILE\_FLAG\_NO\_BUFFERING**.  For more information, see the [Caching Behavior](https://msdn.microsoft.com/en-us/library/windows/desktop/aa363858(v=vs.85).aspx#caching_behavior) section of this topic. |
| **FILE\_FLAG\_WRITE\_THROUGH**  0x80000000 | Ghi trực tiếp vào đĩa, không qua cache.  Bạn tham khảo cơ chế Caching của window [Caching Behavior](https://msdn.microsoft.com/en-us/library/windows/desktop/aa363858(v=vs.85).aspx#caching_behavior). |

Tham số có thể chứa thông tin bảo mật QoS bởi flag: **SECURITY\_SQOS\_PRESENT**, các Flag về bảo mật:

|  |  |
| --- | --- |
| **Security flag** | **Meaning** |
| **SECURITY\_ANONYMOUS** | Impersonates a client at the Anonymous impersonation level. |
| **SECURITY\_CONTEXT\_TRACKING** | The security tracking mode is dynamic. If this flag is not specified, the security tracking mode is static. |
| **SECURITY\_DELEGATION** | Impersonates a client at the Delegation impersonation level. |
| **SECURITY\_EFFECTIVE\_ONLY** | Only the enabled aspects of the client's security context are available to the server. If you do not specify this flag, all aspects of the client's security context are available.  This allows the client to limit the groups and privileges that a server can use while impersonating the client. |
| **SECURITY\_IDENTIFICATION** | Impersonates a client at the Identification impersonation level. |
| **SECURITY\_IMPERSONATION** | Impersonate a client at the impersonation level. This is the default behavior if no other flags are specified along with the **SECURITY\_SQOS\_PRESENT** flag. |

Lưu ý: nếu file tồn tại, hàm sẽ kết hợp flag, thuộc tính của file sẵn có với tham số bạn truyền vào. 1 số Macro chỉ áp dụng cho file, không áp dụng thiết bị khác.

* hTemplateFile [in, optional]: Handle của file tạm với chế độ **GENERIC\_READ**. File tạm chỉ bổ sung thông tin thuộc tính cho file sắp tạo.

Tham số này có thể **NULL**.

Nếu file tồn tại: hàm bỏ qua tham số này.

Nếu tạo file mã hóa: sẽ áp dụng ACL từ thư mục cha.

Hàm trả về Handle Object đã mở nếu thành công. Ngược lại, trả về **INVALID\_HANDLE\_VALUE**

1. Đóng file

Để đóng Handle Object và giải phóng tài nguyên, chúng ta dùng hàm CloseHandle:

BOOL WINAPI CloseHandle(

\_In\_ HANDLE hObject

);

* hObject [in]: Handle Object đã mở

Hàm trả về true nếu thành công. Ngược lại trả về false. Dùng GetLastError() để lấy thông tin về lỗi.

Trường hợp đóng Handle Object hai lần là không hợp lệ, chế độ debug sẽ throw exception.

Chúng ta không cần phải đóng các Handle thiết bị chuẩn.

1. Đọc file

Hàm thiết kế cho cả tiến trình synchronous và asynchronous I/O

Cú pháp:

BOOL WINAPI ReadFile(

\_In\_ HANDLE hFile,

\_Out\_ LPVOID lpBuffer,

\_In\_ DWORD nNumberOfBytesToRead,

\_Out\_opt\_ LPDWORD lpNumberOfBytesRead,

\_Inout\_opt\_ LPOVERLAPPED lpOverlapped

);

* hFile [in]: Handle Object (file, stream, disk, console buffer, socket, pipe …), phải có chế độ đọc GENERIC\_READ.   
  Asynchronous I/O yêu cầu Handle Object bật flag **FILE\_FLAG\_OVERLAPPED**, trong CreateFile() hay socket(), accept()
* lpBuffer [out] con trỏ tới bộ đệm cho việc đọc dữ liệu từ Object.
* nNumberOfBytesToRead [in]: Số byte đọc tối đa.
* lpNumberOfBytesRead [out, optional]: Con trỏ cho biết số byte đã đọc được cho tiến trình Sync I/O. Phải để **NULL** cho Async I/O.
* lpOverlapped [in, out, optional]: Sync I/O để là **NULL**. Async I/O phải truyền cấu trúc **OVERLAPPED** hợp lệ.

Hàm trả về true nếu thành công, ngược lại là false. GetLastError() để lấy thông tin

Nếu không phải Async Handle, tiến trình bắt đầu từ vị trí hiện tại của file. Khi đọc hết file, hàm không bị lỗi, mà số byte đọc được lpNumberOfBytesRead = zero.

1. Ghi file

Hàm thiết kế cho cả tiến trình synchronous và asynchronous I/O

BOOL WINAPI WriteFile(

\_In\_ HANDLE hFile,

\_In\_ LPCVOID lpBuffer,

\_In\_ DWORD nNumberOfBytesToWrite,

\_Out\_opt\_ LPDWORD lpNumberOfBytesWritten,

\_Inout\_opt\_ LPOVERLAPPED lpOverlapped

);

* hFile [in]: Handle Object (file, stream, disk, console buffer, socket, pipe …), phải có chế độ ghi **GENERIC\_WRITE.**Để thực sự ghi vào đĩa cứng, phải có flag: **FILE\_FLAG\_WRITE\_THROUGH**.   
  Async I/O yêu cầu Handle Object bật flag **FILE\_FLAG\_OVERLAPPED**, trong CreateFile() hay socket(), accept()
* lpBuffer [in]: Con trỏ tới bộ đệm dữ liệu ghi vào
* nNumberOfBytesToWrite [in]: Số byte cần ghi
* lpNumberOfBytesWritten [out, optional]: Con trỏ cho biết số byte đã ghi trong tiến trình Sync I/O. Phải để **NULL** cho Async I/O.
* lpOverlapped [in, out, optional]: Sync I/O để là **NULL**. Async I/O phải truyền cấu trúc **OVERLAPPED** hợp lệ.

Nếu Handle Object có hỗ trợ byte offset, có thể thiết lập vị trí ghi file trong cấu trúc **OVERLAPPED,**  thành viên Offset và OffsetHigh.

Để ghi tiếp vào cuối file, thiết lập Offset và OffsetHigh là 0xFFFFFFFF, tương ứng với chế độ **FILE\_APPEND\_DATA**

Hàm trả về true nếu thành công, ngược lại là false. GetLastError() để lấy thông tin

Lưu ý: **GetLastError**() = **ERROR\_IO\_PENDING** ám chỉ tiến trình đang pending bởi thủ tục completion Async I/O.

1. Quản lý file
2. Cơ chế paging File

Cơ chế Paging File (bộ đệm file) tạo ra nhằm giúp tiết kiệm vùng nhớ RAM, đối với các chương trình tiêu hao nhiều vùng nhớ như Firefox chẳng hạn.

Window sẽ không dùng RAM mà dùng bộ đệm trên file ẩn tên là “papefile.sys” tại thư mục gốc ổ đĩa.

Trường hợp cửa sổ chương trình thu nhỏ xuống taskbar, Window cũng move vùng nhớ tới page file. Điều này khiến chương trình sẽ mất một lúc khi kích hoạt lại và ổ đĩa cứng sẽ hoạt động nhiều hơn một chút.

Window cho phép chúng ta quản lý bằng tay lượng page file, nhưng tốt nhất là để OS tự động làm việc này.

Lưu ý: việc vô hiệu tính năng page file có thể gây ra lỗi, khuyến cáo không nên làm vậy.

1. Xóa File

Phương thức DeleteFile dùng để xóa file, cũng như xóa link

BOOL WINAPI DeleteFile(

\_In\_ LPCTSTR lpFileName

);

* lpFileName [in]: Đường dẫn file xóa

Hàm trả về true nếu thành công, ngược lại trả về false. GetLastError() có thể trả về:

ERROR\_FILE\_NOT\_FOUND : file không tồn tại

ERROR\_ACCESS\_DENIED : thuộc tính file xóa là read-only.

1. Sao chép File

Hàm CopyFile() sao chép, và chỉ thay thế file mới đã tồn tại khi tham số bFailIfExists = FALSE

BOOL WINAPI CopyFile(

\_In\_ LPCTSTR lpExistingFileName,

\_In\_ LPCTSTR lpNewFileName,

\_In\_ BOOL bFailIfExists

);

* lpExistingFileName [in]: Đường dẫn file để sao chép
* lpNewFileName [in]: Đường dẫn tới file mới
* bFailIfExists [in]: TRUE hay FALSE

Hàm trả về true nếu thành công, ngược lại trả về false. GetLastError() có thể trả về:

ERROR\_ACCESS\_DENIED : nếu file sao chép có thuộc tính FILE\_ATTRIBUTE\_HIDDEN hay FILE\_ATTRIBUTE\_READONLY

ERROR\_ENCRYPTION\_FAILED : thất bại khi sao chép file mã hóa.

Khi sao chép, các thuộc tính bảo mật file (ATTRIBUTE\_SECURITY\_INFORMATION) và thuộc tính cũng được sao chép

1. Đổi tên và chuyển File

Hàm MoveFile() có thể áp dụng cho cả thư mục và tất cả file con.

BOOL WINAPI MoveFile(

\_In\_ LPCTSTR lpExistingFileName,

\_In\_ LPCTSTR lpNewFileName

);

* lpExistingFileName [in]: file hay thư mục trên máy nội bộ
* lpNewFileName [in]: tên mới cho file hay thư mục. Tên này không được tồn tại

File mới có thể nằm khác ổ đĩa và khác File system. Nhưng thư mục phải cùng ổ đĩa.

Hàm MoveFile() trả về true nếu thành công, ngược lại trả về false nếu tên đã tồn tại. GetLastError() để bug lỗi.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**MoveFileEx**() có cùng mục đích và được mở rộng, sử dụng phổ biến hơn, cho phép ghi đè file đã tồn tại.

BOOL WINAPI MoveFileEx(

\_In\_ LPCTSTR lpExistingFileName,

\_In\_opt\_ LPCTSTR lpNewFileName,

\_In\_ DWORD dwFlags

);

* dwFlags [in]: Tham số qui định cách move file

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| **MOVEFILE\_COPY\_ALLOWED**  2 (0x2) | Chuyển file khác ổ đĩa, hàm dùng CopyFile() và DeleteFile().  Nếu không thể xóa file gốc, hàm vẫn trả về TRUE.  Flag không được dùng chung với **MOVEFILE\_DELAY\_UNTIL\_REBOOT**. |
| **MOVEFILE\_CREATE\_HARDLINK**  16 (0x10) | Không sử dụng. |
| **MOVEFILE\_DELAY\_UNTIL\_REBOOT**  4 (0x4) | OS không thực sự move file, mà đợi cho tới khi OS khởi động lại, và chương trình AUTOCHK thực thi trước khi bộ đệm paging file được tạo.  Flag này khiến hàm xóa paging file từ lần khởi động mới đây.  Flag yêu cầu User phải là ADMIN group hay LocalSystem User.  Flag không dùng chung với **MOVEFILE\_COPY\_ALLOWED**. |
| **MOVEFILE\_FAIL\_IF\_NOT\_TRACKABLE**  32 (0x20) | Áp dụng khi file move tới FAT file system.  Không áp dụng cho file gốc là liên kết hard / symbolic.  File sẽ không bị giám sát sau khi move. |
| **MOVEFILE\_REPLACE\_EXISTING**  1 (0x1) | Flag không áp dụng cho đối tượng thư mục.  Nếu file mới tồn tại, hàm sẽ ghi đè file. Tiến trình phải thỏa điều kiện bảo mật của ACL. |
| **MOVEFILE\_WRITE\_THROUGH**  8 (0x8) | Hàm chỉ return tới khi file được move.  Flag này đảm bảo tiến trình Copy() và Delete() phải hoàn tất (flush) trên đĩa cứng trước khi trả về.  Nếu **MOVEFILE\_DELAY\_UNTIL\_REBOOT** được bật, flag này vô hiệu. |

**MOVEFILE\_DELAY\_UNTIL\_REBOOT** yêu cầu User phải có quyền hạn ADMIN vì nó cần truy xuất tới registry:

*HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations*

Các thông tin thao tác file chứa trong REG\_MULTI\_SZ, các entry có dạng chuỗi, mỗi thao tác phân cách bới kí tự “\0”

szDstFile\0\0 : đây là thao tác delete.

szSrcFile\0szDstFile\0 : thao tác rename.

Khi hệ thống khởi động (reboot), các entry này được thực thi và xóa sạch.

1. Console I/O
2. STD Handle trên Win API

Giống như Unix, Window cũng có 3 thiết bị chuẩn cho Input, Output và Error

Unix dùng 3 giá trị cho file descriptor : 0, 1 và 2

C dùng 3 std stream: stdin, stdout và stderror

C++ là : std::cout, std::cin và std::error

Window lại dùng Handle và dùng hàm GetStdHandle() để lấy std Handle:

HANDLE WINAPI GetStdHandle(

\_In\_ DWORD nStdHandle

);

* nStdHandle [in]: Macro tương ứng thiết bị chuẩn

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| **STD\_INPUT\_HANDLE** (DWORD)-10 | Std in. console input buffer, CONIN$. |
| **STD\_OUTPUT\_HANDLE** (DWORD)-11 | Std out. console screen buffer, CONOUT$. |
| **STD\_ERROR\_HANDLE** (DWORD)-12 | Std error. console screen buffer, CONOUT$. |

Hàm trả về Handle tương ứng với thiết bị chuẩn nếu thành công, ngược lại trả về INVALID\_HANDLE\_VALUE. Giá trị std Handle trả về là giống nhau, hàm không tạo ra bản sao nào cả.

Nếu ứng dụng là service chạy ngầm, hàm trả về NULL.

Khi bạn đóng std Handle, sẽ khiến thiết bị không thể dùng được nữa cho process. Vì vậy chúng ta không bao giờ đóng std Handle.

Handle trả về sẽ có quyền truy xuất GENERIC\_READ và GENERIC\_WRITE, Chúng ta có thể dùng SetStdHandle() để thay đổi quyền truy xuất std Handle

BOOL WINAPI SetStdHandle(

\_In\_ DWORD nStdHandle,

\_In\_ HANDLE hHandle

);

* nStdHandle [in]: Macro tương ứng thiết bị chuẩn

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| **STD\_INPUT\_HANDLE** (DWORD)-10 | Std in |
| **STD\_OUTPUT\_HANDLE** (DWORD)-11 | Std out |
| **STD\_ERROR\_HANDLE** (DWORD)-12 | Std error |

* hHandle [in]: là std Hanlde

Hàm trả về true nếu thành công, ngược lại trả về false

Chúng ta có thể dùng SetStdHandle() để chuyển hướng (redirect) Standard I/O, khi đó không thể lấy std Handle bằng GetStdHandle() được.

Bạn có thể dùng CONIN$, CONOUT$ và CONOUT$ truyền vào tham số lpFileName của CreateFile() để lấy đúng Console I/O Handle.

HANDLE hin=CreateFile("CONIN$",GENERIC\_READ|GENERIC\_WRITE,FILE\_SHARE\_READ,0,OPEN\_EXISTING,0,0);

CreateFile() trả về INVALID\_HANDLE\_VALUE nếu không lấy đúng fdwShareMode. Đảm bảo FILE\_SHARE\_READ cho "CONIN$" và FILE\_SHARE\_WRITE cho "CONOUT$".

1. Chọn chế độ Console

Việc chọn chế độ console sẽ ảnh hưởng tới cách hệ thống thực thi các tiến trình I/O. Window định nghĩa 2 bộ đệm I/O là bộ đệm Input và bộ đệm screen.

Chế độ Output chỉ phân ở mức high-level.

Chế độ Input chia làm 2 mức tương ứng với 2 bộ API high-level va low-level.

Ứng dụng có thể thay đổi chế độ console bất kì lúc nào

BOOL WINAPI SetConsoleMode(

\_In\_ HANDLE hConsoleHandle,

\_In\_ DWORD dwMode

);

* hConsoleHandle [in]: là Handle input hay screen buffer. Phải có quyền GENERIC\_READ.
* dwMode [in]: Chế độ I/O cho hConsoleHandle, qui định cách kí tự được xử lí.

Hàm trả về TRUE nếu thành công, ngược lại trả về FALSE. GetLastError để bug lỗi.

[Tham khảo thêm về tham số dwMode](https://msdn.microsoft.com/en-us/library/windows/desktop/ms686033(v=vs.85).aspx)

Có 5 tham số hay sử dụng nhất:

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| **ENABLE\_ECHO\_INPUT**  0x0004 | Kí tự đọc bởi [**ReadFile**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365467(v=vs.85).aspx) hay [**ReadConsole**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms684958(v=vs.85).aspx) xuất ra màn hình (screen buffer)   Yêu cầu **ENABLE\_LINE\_INPUT** phải kích hoạt. |
| **ENABLE\_LINE\_INPUT**  0x0002 | [**ReadFile**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365467(v=vs.85).aspx) và [**ReadConsole**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms684958(v=vs.85).aspx) trả về khi có kí tự ngắt dòng carriage return ‘/r’ được nhập. |
| **ENABLE\_PROCESSED\_INPUT**  0x0001 | Mặc định CTRL+C được hệ thống tự xử lí, và không lưu vào trong bộ đệm.  Khi kích hoạt flag này, [**ReadFile**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365467(v=vs.85).aspx) và [**ReadConsole**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms684958(v=vs.85).aspx) sẽ khiến hệ thống xử lí backspace ‘/t, carriage return ‘/r’, và line feed ‘/r/n’ |
| **ENABLE\_PROCESSED\_OUTPUT**  0x0001 | [**WriteFile**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365747(v=vs.85).aspx) và [**WriteConsole**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms687401(v=vs.85).aspx) sẽ khiến hệ thống xử lí backspace ‘/t, carriage return ‘/r’, và line feed ‘/r/n’ |
| **ENABLE\_WRAP\_AT\_EOL\_OUTPUT**  0x0002 | [**WriteFile**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365747(v=vs.85).aspx) và [**WriteConsole**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms687401(v=vs.85).aspx) hay echo bởi [**ReadFile**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365467(v=vs.85).aspx) và [**ReadConsole**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms684958(v=vs.85).aspx), cho phép line wrap. |

line wrap : Là tính năng cho phép xuống dòng mới khi dòng hiện tại đã bị đầy, giúp văn bản có thể được xem mà không cần phải kéo màn hình.

1. Low-level I/O

Cho phép ứng dụng truy xuất trực tiếp tới bộ đệm console input và screen. API low-level cho phép:

* Tiếp nhận các sự kiện Input về chuột, bàn phím, thay đổi bộ đệm.
* Lưu bản ghi Input vào bộ đệm Input.
* Đọc bộ đệm Input mà không phải xóa
* Xác định số lượng sự kiện đang bị trì hoãn trong bộ đệm Input
* Flush bộ đệm Input
* Đọc và ghi chuỗi Unicode hay ANSI vào bộ đệm screen.
* Đọc và ghi chuỗi và thuộc tính màu nền tại vị trí screen bất kì
* Đọc và ghi dữ liệu màu và kí tự theo một khối hình chữ nhật trên màn hình
* Ghi 1 kí tự hay chuỗi kí tự Unicode/ANSI và màu nền.

1. High-level I/O

Cung cấp phương thức đơn giản để đọc và ghi các stream dữ liệu, kí tự Unicode hay ANSI. (ReadFile, WriteFile, ReadConsole, WriteConsole)

Hàm File I/O (không hỗ trợ Unicode)có thể dùng để truy xuất file, pipe và thiết bị nối tiếp.

Hàm console chỉ có thể dùng với console Handle.

Cho phép ứng dụng kiểm soát kí tự và màu nền hiển thị kí tự trong bộ đệm screen.

Cho phép kích hoạt hay vô hiệu các tính năng:

* Phản hồi input bàn phím lên màn hình (screen buffer)
* Nhập line kí tự, enter để kết thúc
* Tự động xử lí input bàn phím khi carriage return, CTRL + C…
* Tự động xư lí output khi line wrapping, carriage returns, backspaces, …

1. ReadConsole và WriteConsole

ReadConsole đọc và xóa luôn kí tự khỏi bộ đệm Input. Các tham số tương tự như ReadFile()

BOOL WINAPI ReadConsole(

\_In\_ HANDLE hConsoleInput,

\_Out\_ LPVOID lpBuffer,

\_In\_ DWORD nNumberOfCharsToRead,

\_Out\_ LPDWORD lpNumberOfCharsRead,

\_In\_opt\_ LPVOID pInputControl

);

* hConsoleInput [in]: Handle console Input có quyền GENERIC\_READ
* pInputControl [in, optional]: Con trỏ kiểu cấu trúc CONSOLE\_READCONSOLE\_CONTROL , dùng để kiểm soát kí tự nào đánh dấu kết thúc việc đọc.  
  Tham số này phải để NULL, nghĩa là hàm chỉ xử lí ANSI.

Hàm trả về TRUE nếu thành công, ngược lại trả về FALSE. GetLastError() để bug lỗi.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

WriteConsole ghi tiếp kí tự từ bộ đệm vào screen buffer. Tham số cũng tương tự như WriteFile()

BOOL WINAPI WriteConsole(

\_In\_ HANDLE hConsoleOutput,

\_In\_ const VOID \*lpBuffer,

\_In\_ DWORD nNumberOfCharsToWrite,

\_Out\_ LPDWORD lpNumberOfCharsWritten,

\_Reserved\_ LPVOID lpReserved

);

* hConsoleOutput [in]: Handle console Output, có quyền GENERIC\_WRITE
* lpReserved: phải để NULL.

Hàm trả về TRUE nếu thành công, ngược lại trả về FALSE. GetLastError() để bug lỗi.

1. FreeConsole và AllocConsole

Một tiến trình chỉ có thể có duy nhất một console 1 thời điểm. Thông thường ứng dụng như server hay ứng dụng đồ họa sẽ cần cửa sổ console, để hiển thị trạng thái hệ thống, debug.

Lý do việc khởi tạo console là vì ứng dụng Window GUI không có console mặc định, kể cả process khi tạo ra.

Có hai phương thức đơn giản để làm việc này:

BOOL WINAPI AllocConsole(void);

BOOL WINAPI FreeConsole(void);

Hàm trả về TRUE nếu thành công, ngược lại trả về FALSE. GetLastError() để bug lỗi.

AllocConsole sẽ FALSE nếu process đã có console rồi. AllocConsole khởi tạo các std Handle (Input, Ouput, Error) cho console mới.

Để tạo console mới, bạn phải gọi FreeConsole để tách console của process, sau đó dùng AllocConsole khởi tạo console mới cho process.

Nếu có process con tạo ra, nó sẽ được gắn console mới.

Ứng dụng đồ họa được tạo ra sẽ không có console, nó có thể dùng AllocConsole hoặc tạo ra một process riêng lẻ (CreateProcess()) để dùng cửa sổ console.

FreeConsole sẽ FALSE nếu process chưa có console nào cả, lỗi ERROR\_INVALID\_PARAMETER.

Trường hợp process đang dùng console chia sẻ với process khác, nó sẽ bị tách khỏi console này.

Console chia sẻ sẽ bị đóng chỉ khi không còn process nào tham chiếu tới nó (do kết thúc hoặc do gọi FreeConsole)

1. Ứng dụng minh họa
2. Xuất thông tin lỗi

Phương thức ReportError có chức năng tương đương với C API perror (giải nghĩa mã lỗi err\_no).

Hàm có hỗ trợ in thêm tin nhắn từ tham số đầu vào.

Thay vì xuất ra std Handle Error, chúng ta dùng std error từ C library cho tiện với \_ftprintf.

Nếu tham số exitCode > 0 ta sẽ ngắt chương trình với ExitProcess.

Tham số isPrint qui định hàm có in thông tin hay không.

FormatMessage(): phân giải error code từ GetLastError sang tin nhắn ý nghĩa vào bộ đệm cấp phát.

Cuối cùng phải dùng LocalFree để giải phóng tài nguyên cho bộ đệm từ FormatMessage.

#include <Windows.h>

#include <tchar.h>

VOID ReportError(TCHAR \*Msg, DWORD exitCode, BOOL isPrint){

DWORD MsgLen;

DWORD errCode = GetLastError();

TCHAR \*SystemMsg;

\_ftprintf(stderr, \_T("%s/n"), Msg);

if (isPrint){

MsgLen = FormatMessage(FORMAT\_MESSAGE\_ALLOCATE\_BUFFER | FORMAT\_MESSAGE\_FROM\_SYSTEM,

NULL, errCode, NULL, SystemMsg, 0, NULL);

if (MsgLen >0)

\_ftprintf(stderr, \_T("%s/n"), SystemMsg);

else

\_ftprintf(stderr, \_T("Last error number: %d/n"), errCode);

//Giải phóng tài nguyên

if (SystemMsg!=NULL)

LocalFree(SystemMsg);

}

//Giải phóng tài nguyên

if (exitCode > 0)

ExitProcess(exitCode);

}

Chúng ta sẽ tìm hiểu FormatMessage và cơ chế trong chương sau.

1. In nội dung file

Chương trình dùng đối số dòng lệnh với cú pháp: -s inputFilePath

Xuất nội dung của file inputFilePath lên màn hình

#include <Windows.h>

#include <tchar.h>

#define BUFSIZ 512

const TCHAR\* opt = \_T("-s");

//

VOID CatFile(const TCHAR\* in){

HANDLE inFile = CreateFile(in, GENERIC\_READ, 0, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

HANDLE stdOut = GetStdHandle(STD\_OUTPUT\_HANDLE);

if (inFile == INVALID\_HANDLE\_VALUE)

exit(1);

DWORD ByteRead, ByteWrite;

BYTE buffer[BUFSIZ];

while(ReadFile(inFile, buffer, BUFSIZ, &ByteRead,NULL)&&

ByteRead!=0 &&

WriteFile(stdOut, buffer, ByteRead, &ByteWrite,NULL)){

}

CloseHandle(inFile);

return;

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

//Kiểm tra tham số

if (argc ==3 && lstrcmp(argv[1], opt) ==0)

CatFile(argv[2]);

return 0;

}

1. Mã hóa Caesar Cipher

Caesar là thuật mã hóa đơn giản nhất. Mỗi kí tự được thay thế bởi kí tự phía trước hay sau trong bảng chữ cái alphabet.

Chuỗi kí tự sau khi mã hóa sẽ không thể hiểu được. Ví dụ:

THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG

Mã hóa thành:

QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD

Gọi x là kí tự gốc, n là giá trị thay đổi

Chúng ta có giá trị kí tự mã hóa là:

Kí tự giải mã:

1. Trình mã hóa, giải mã file

Chúng ta dùng thuật toán Caesar để mã hóa từng byte một và lưu file.

Chọn n = 5 và mod 256. Để giải mã chỉ cần giảm n thôi.

Chương trình này khá đơn giản, và dùng cho nhiều ví dụ về sau cho thread, Async I/O…

#include <Windows.h>

#include <tchar.h>

#define shift 5 //số step nhảy

const TCHAR\* encrypt = \_T("-e");

const TCHAR\* decrypt = \_T("-d");

VOID encryptF(const TCHAR\* in, const TCHAR\* out){

HANDLE inFile, outFile;

//Tạo File handle

inFile = CreateFile(in, GENERIC\_READ, 0, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

if (inFile == INVALID\_HANDLE\_VALUE)

exit(1);

outFile = CreateFile(out, GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, NULL);

if (inFile == INVALID\_HANDLE\_VALUE)

exit(1);

CHAR buffer[BUFSIZ], caesarBuffer[BUFSIZ];

DWORD ByteRead, ByteWrite;

while(ReadFile(inFile, buffer, BUFSIZ, &ByteRead, NULL) && ByteRead!=0){

for(DWORD i = 0; i < ByteRead; ++i)

caesarBuffer[i] = (buffer[i] + shift) % 256;

if (!WriteFile(outFile, caesarBuffer, ByteRead, &ByteWrite, NULL))

break;

}

//Đóng File handle

CloseHandle(inFile);

CloseHandle(outFile);

}

VOID decryptF(const TCHAR\* in, const TCHAR\* out){

HANDLE inFile, outFile;

//Tạo File handle

inFile = CreateFile(in, GENERIC\_READ, 0, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

if (inFile == INVALID\_HANDLE\_VALUE)

exit(1);

outFile = CreateFile(out, GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, NULL);

if (inFile == INVALID\_HANDLE\_VALUE)

exit(1);

CHAR buffer[BUFSIZ], caesarBuffer[BUFSIZ];

DWORD ByteRead, ByteWrite;

while(ReadFile(inFile, buffer, BUFSIZ, &ByteRead, NULL) && ByteRead!=0){

for(DWORD i = 0; i < ByteRead; ++i)

caesarBuffer[i] = (buffer[i] - shift) % 256;

if (!WriteFile(outFile, caesarBuffer, ByteRead, &ByteWrite, NULL))

break;

}

//Đóng File handle

CloseHandle(inFile);

CloseHandle(outFile);

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

//Kiểm tra đối số

if (argc==4){

//Trương hợp mã hóa

if (lstrcmp(argv[1], encrypt)==0)

encryptF(argv[2], argv[3]);

//Trường hợp giải mã

if (lstrcmp(argv[1], decrypt)==0)

decryptF(argv[2], argv[3]);

}

return 0;

}

1. Output tin nhắn lên Console

Chúng ta xây dựng chương trình ConsolePromt phản hồi tin nhắn từ người dùng, có tùy chọn không phản hồi.

Phương thức dùng Console I/O và kí tự Generic, có thể xử lí mọi Handle Output và Error.

PrintStrings dùng tham số tùy ý với va\_start, va\_arg và va\_end .

PrintMsg dùng cho xuất một chuỗi đơn giản.

#include <Windows.h>

#include <stdarg.h>

#include <tchar.h>

#include "ErrorReport.h"

const TCHAR\* consolePromtOpt = \_T("-p");

//

BOOL PrintStrings(HANDLE out, ...){

DWORD MsgLen, count;

LPCTSTR Msg;

va\_list MsgList;

va\_start(MsgList, out);

while((Msg = va\_arg(MsgList, LPCTSTR)) !=NULL){

MsgLen= lstrlen(Msg);

//

if (!WriteConsole(out, Msg, MsgLen, &count, NULL) && !WriteFile(out, Msg, MsgLen\*sizeof(TCHAR), &count, NULL)){

va\_end(MsgList);

return FALSE;

}

}

va\_end(MsgList);

return TRUE;

}

//In chuỗi đơn giản

BOOL PrintMsg(HANDLE out, const TCHAR\* Msg){

return PrintStrings(out, Msg, NULL);

}

//

BOOL ConsolePromt(LPCTSTR PromtMsg, TCHAR\* ResponseMsg, DWORD MaxChar, BOOL echo){

HANDLE in,out;

DWORD charRead, flagInput;

BOOL ret;

//Lấy console Handle I/O

in = CreateFile(\_T("CONIN$"), GENERIC\_READ | GENERIC\_WRITE, 0, NULL, OPEN\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, NULL);

out = CreateFile(\_T("CONOUT$"), GENERIC\_WRITE, 0, NULL, OPEN\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, NULL);

flagInput = echo?ENABLE\_ECHO\_INPUT:0;

flagInput = flagInput | ENABLE\_LINE\_INPUT | ENABLE\_PROCESSED\_INPUT;

ret = SetConsoleMode(in, flagInput)

&& SetConsoleMode(out, ENABLE\_PROCESSED\_OUTPUT | ENABLE\_WRAP\_AT\_EOL\_OUTPUT)

&& PrintStrings(out, PromtMsg)

&& ReadConsole(in, (VOID\*)ResponseMsg, MaxChar - 2, &charRead, NULL);

//

if (ret)

ResponseMsg[charRead - 2] = '\0';

else

ReportError(\_T("Console Promt failure."), 0, TRUE);

CloseHandle(in);

CloseHandle(out);

return ret;

}

Tham số MaxChar đảm bảo không bị tràn bộ đệm.

1. Liệt kê thư mục hiện hành

Unix sử dụng lệnh “pwd” để liệt kê thư mục hiện hành.

Nhắc lại rằng MAX\_PATH qui định kích thước tối đa file name.

#include "consolePromt.h"

#include "ErrorReport.h"

#define DIR\_NAME\_LEN (MAX\_PATH + 2)

VOID PrintCurrentDirectory(){

DWORD len;

TCHAR buffer[DIR\_NAME\_LEN];

len = GetCurrentDirectory(DIR\_NAME\_LEN, buffer);

if (len ==0)

ReportError(\_T("failure getting path name"), 1, TRUE);

if (len > DIR\_NAME\_LEN)

ReportError(\_T("Path name too long"), 2, FALSE);

PrintMsg(GetStdHandle(STD\_OUTPUT\_HANDLE), buffer);

}

1. Trên Unix
2. Đọc file

The comparable UNIX functions are different in a number of ways. The UNIX

function returns an integer file descriptor rather than a handle, and it

specifies access, sharing, create options, attributes, and flags in the single integer

parameter. The options overlap, with Windows providing a richer set.

There is no UNIX equivalent to . UNIX files are always shareable.

Both systems use security information when creating a new file. In UNIX, the

argument specifies the familiar user, group, and other file permissions.

is comparable to but it is not general purpose.

The C library functions use objects, which are comparable to han

dles (for disk files, terminals, tapes, and other devices) connected to streams. The

mode parameter specifies whether the file data is to be treated as binary or

text. There is a set of options for read-only, update, append at the end, and so on.

allows reuse without closing it first. The Standard C library can

not set security permissions.

closes a . Most -related functions have the f prefix.

1. Ghi file

UNIX and are the comparable functions, and the programmer supplies a file descriptor, buffer, and byte count. The functions return the number of bytes actually transferred. A value of on indicates the end of file; – indicates an error. Windows, by contrast, requires a separate transfer count and returns Boolean values to indicate success or failure.

The functions in both systems are general purpose and can read from files, terminals, tapes, pipes, and so on.

The Standard C library and binary I/O functions use object size and object count rather than a single byte count as in UNIX and Windows. A short transfer could be caused by either an end of file or an error; test explicitly

with or . The library provides a full set of text-oriented functions, such as and , that do not exist outside the C library in either OS.

1. Standard I/O

UNIX standard I/O redirection is considerably different (see Stevens and Rago [pp. 61–64]).

The first method is indirect and relies on the fact that the function returns the lowest numbered available file descriptor. Suppose you wish to reassign standard input (file descriptor ) to an open file description, . The first method is:

close(STDIN\_FILENO);

dup(fd\_redirect);

The second method uses dup2, and the third uses F\_DUPFD on the cryptic and overloaded fcntl function.

1. Move file

UNIX pathnames do not include a drive or server name; the slash indicates the system root. The Microsoft C library file functions also support drive names as required by the underlying Windows file naming.

UNIX does not have a function to copy files directly. Instead, you must write a small program or call to execute the command.

unlink() is the UNIX equivalent DeleteFile() of except that unlink() can also delete directories.

Rename() and remove() are in the C library, and will fail when attempting to move a file to an existing file name or a directory to a non-empty directory.

1. Console I/O

For historical reasons, Windows does not support character-oriented terminals in the way that UNIX does, and not all the UNIX terminal functionality is replicated by Windows. For example, UNIX provides functions for setting baud rates and line control functions. Stevens and Rago dedicate a chapter to UNIX terminal I/O

(Chapter 11) and one to pseudo terminals (Chapter 19). Serious Windows user interfaces are, of course, graphical, with mouse as well as keyboard input. The GUI is outside the scope of this book, but everything we discuss works within a GUI application.

### Ký tự trên WinAPI

1. Lưu ý về bộ kí tự
2. Unicode và kí tự phổ quát

Window hỗ trợ kí tự 8-bit và kí tự mở rộng 16-bits (định nghĩa bởi C). Microsoft xem kí tự 8-bit là chuẩn ANSI, thực tế là họ nhầm lẫn, chúng ta xem đó là chuẩn ASCII.

Bộ kí tự mở rộng hỗ trợ dùng chuẩn Unicode UTF-16 mã hóa, hiển thị cho nhiều ngôn ngữ.

Đây là các bước mà ứng dụng Window được built để dùng Unicode hay 8-bits ASCII:

1. Định nghĩa mọi kí tự và chuỗi dùng kiểu chung TCHAR, LPTSTR và LPCTSTR
2. Chèn chỉ thị tiền xử lí #define UNICODE và #define \_UNICODE vào mọi mã nguồn để lấy kí tự UNICODE (TCHAR sẽ tương ứng với wchar\_t trong ANSI C).

Ngược lại, nếu không định nghĩa, TCHAR sẽ tương ứng với char trong C.

Bạn phải #include <window.h> trong <stdafx.h> của Visual studio project.

* #define UNICODE: bổ sung định nghĩa UNICODE của window API.
* #define \_UNICODE: bổ sung phần điều khiển thư viện C.

1. Khi tính toán kích thước theo bytes của bộ đệm phải tính theo sizeof(TCHAR)
2. Chỉ nên dùng các hàm chuẩn C trong thư viện <tchar.h> như:   
   \_fgettc, \_itoa (với itoa)  
   \_stprintf với sprintf  
   \_tcscpy với strcpy  
   \_ttoi, \_totupper, \_totlower và \_ftprintf  
   Có thể tham khảo thêm trong msdn, các hàm trên đều lệ thuộc vào \_UNICODE
3. Các chuỗi kí tự hằng sẽ là 1 trong 3 dạng sau:  
   “example string” : chuỗi kí tự 8-bit ASCII  
   L“example string” : chuỗi kí tự 16-bit  
   \_T“example string” : chuỗi kí tự chung với \_T macro, trên trình biên dịch Microsoft C
4. Chèn #include <tchar.h> sau <window.h>, để có định nghĩa về text Macro và các API chuẩn của C

Chỉ vài Window API hỗ trợ Unicode, tương lai sẽ tiếp tục hoàn thiện.

Chúng ta phải kiểm tra xem các API hệ thống lệ thuộc thế nào về định nghĩa kiểu kí tự, đơn giản là:

#ifdef UNICODE

#define TCHAR WCHAR

#else

#define TCHAR CHAR

#endif

1. WinAPi xử lí chuỗi

So sánh chuỗi có thể dùng lstrcmp và lstrcmpi nhiều hơn so với hàm chung như \_tcscmp và \_tcscmpi, cho các loại ngôn ngữ vùng miền.

Lstrcmp() và lstrcmpi() có thể so sánh từng từ, từng kí tự một.

Cả 2 phương thức cho ra kết quả trái ngược nhau khi so sánh hai cặp chuỗi: coop/co-op và were/we’re.

Window API có nhiều hàm xử lí Unicode và tính vùng miền của chuỗi, như hàm CompareString()

Hàm CharUpper(),IsCharAlphaNumeric() có thể áp dụng cho kí tự và chuỗi

1. Hàm main tổng quát

Win API thay thế hàm main của C, dùng macro **\_tmain** với đối số dòng lệnh argv[]

**\_tmain:** Macro này lệ thuộc vào macro \_UNICODE, được định nghĩa trong <tchar.h>. Cú pháp chuẩn sẽ như sau:

#include <Windows.h>

#include <tchar.h>

int \_tmain(int argc, TCHAR \*argv[])

Ngoài ra \_tmain còn hỗ trợ tham số thứ 3, giống như trên phiển bản non-standard main trên Unix vậy.

1. Định nghĩa hàm

Những hàm như CreateFile() được định nghĩa qua macro CreateFileA nếu không có khai báo UNICODE.

Ngược lại nếu có khai báo UNICODE, CreateFile() định nghĩa bởi CreateFileW

Phiên bản UNICODE sẽ xử lí được kí tự mở rộng, bản không UNICODE chỉ xử lí kí tự 8-bits.

Nếu tham số chuỗi kí tự không phù hợp, trình biên dịch sẽ báo lỗi cho các macro CreateFileA và CreateFileW.

1. Cách dùng Unicode

Lập trình viên trên Window có thể dựa vào 4 phương pháp sau:

1. Chỉ dùng 8-bits: bỏ qua Unicode và tiếp tục dùng kiểu char và hàm cơ bản của C chuẩn
2. 8-bits hay Unicode với code tổng quát: tuân theo chỉ dẫn về lập trình generic, không định nghĩa Macro Unicode chi phiên bản 8-bits
3. Chỉ dùng Unicode: tuân theo chỉ dẫn lập trình generic, nhưng không định nghĩa cả hai Macro. Thay vào đó, dùng kí tự mở rộng và hàm kí tự mở rộng mà thôi.
4. Unicode và 8-bit: chương trình dùng đồng thời 2 bộ kí tự và phải tự chọn phương thức phù hợp trong runtime

Việc viết code generic sẽ tốn nhiều chi phí hơn giúp chương trình tùy biến giữa 2 bộ API.

Trường hợp thứ 3 là phổ biến, đặc biệt trong ứng dụng giao diện.

1. Thao tác với chuỗi

Các hàm xử lí chuỗi của Win API dưới đây đều là dạng phổ quát (hỗ trợ Unicode và ANSI).

Lưu ý khi chúng ta khai báo tham số độ dài của bộ đệm:

* Dùng lstrlen() \* sizeof(TCHAR) cho các hàm File I/O như ReadFile, WriteFile, vì chúng không hỗ trợ Unicode nên không biết kích thước chính xác của ký tự Unicode
* Dùng lstrlen() với các hàm Console như ReadConsole, WriteConsole, vì chúng hỗ trợ Unicode.

1. Khởi tạo chuỗi TCHAR\*

Có thể dùng các Macro sau: TEXT(), \_T() để khởi tạo chuỗi TCHAR\*

TCHAR myVariable[260] = TEXT("initial value");

const TCHAR\* opt = \_T("-s");

1. Độ dài chuỗi

int WINAPI lstrlen(

\_In\_ LPCTSTR lpString

);

* pString [in]: Con trỏ tới chuỗi kí tự TCHAR\* có chứa kí tự kết chuỗi.

1. So sánh chuỗi

Tham số chuỗi LPCTSTR là kiểu TCHAR\*

Các hàm có thể dùng cho kí tự 8-bit hay Unicode

1. lstrcmp

Hàm so sánh từng kí tự 2 chuỗi, có phân biệt hoa thường.

Hai chuỗi so sánh phải chứa kí tự kết chuỗi

int WINAPI lstrcmp(

\_In\_ LPCTSTR lpString1,

\_In\_ LPCTSTR lpString2

);

* lpString1 [in]: con trỏ tới chuỗi 1
* lpString2 [in]: con trỏ tới chuỗi 2

Giá trị trả về:

< 0 : chuỗi 1 ngắn hơn chuỗi 2

>0 : chuỗi 1 dài hơn chuỗi 2

= 0 : hai chuỗi giống nhau

1. lstrcmpi

Hàm so sánh từng kí tự 2 chuỗi, không phân biệt hoa thường.

Hai chuỗi so sánh phải chứa kí tự kết chuỗi

int WINAPI lstrcmpi(

\_In\_ LPCTSTR lpString1,

\_In\_ LPCTSTR lpString2

);

* lpString1 [in]: con trỏ tới chuỗi 1
* lpString2 [in]: con trỏ tới chuỗi 2

Giá trị trả về:

< 0 : chuỗi 1 ngắn hơn chuỗi 2

>0 : chuỗi 1 dài hơn chuỗi 2

= 0 : hai chuỗi giống nhau

1. CompareString

Là hàm so sánh hai chuỗi, có qui định về cách thức so sánh, tính vùng miền ngôn ngữ.

int CompareString(

\_In\_ LCID Locale,

\_In\_ DWORD dwCmpFlags,

\_In\_ LPCTSTR lpString1,

\_In\_ int cchCount1,

\_In\_ LPCTSTR lpString2,

\_In\_ int cchCount2

);

* Locale [in]: qui định đặc tính vùng miền. Có thể dùng hàm MAKELCID() để tạo Locale mới, hay có thể dùng các Marco sau:

**LOCALE\_CUSTOM\_DEFAULT**

**LOCALE\_CUSTOM\_UI\_DEFAULT**

**LOCALE\_CUSTOM\_UNSPECIFIED**

**LOCALE\_INVARIANT**

**LOCALE\_SYSTEM\_DEFAULT**

**LOCALE\_USER\_DEFAULT**

* dwCmpFlags [in]: Flag định nghĩa cách so sánh của hàm

[Tham khảo tại đây](https://msdn.microsoft.com/en-us/library/windows/desktop/dd317761(v=vs.85).aspx)

* lpString1 [in]: Con trỏ tới chuỗi 1
* cchCount1 [in]: Độ dài chuỗi 1
* lpString2 [in]: Con trỏ tới chuỗi thứ 2
* cchCount2 [in]: Độ dài chuỗi 2

Hàm trả về các giá trị sau:

CSTR\_LESS\_THAN : chuỗi 1 ngắn hơn chuỗi 2.

CSTR\_EQUAL: chuỗi 1 giống chuỗi 2 theo định nghĩa của flag dwCmpFlags

CSTR\_GREATER\_THAN.: chuỗi 1 dài hơn chuỗi 2

Trả về 0: không thành công. GetLastError() trả về:

ERROR\_INVALID\_FLAGS: giá trị flag truyền không hợp lệ.

ERROR\_INVALID\_PARAMETER: giá trị tham số không hợp lệ.

1. Nối chuỗi

StringCchCat nằm trong thư viện <Strsafe.h>, dùng thay cho các phương thức cùng chức năng khác như : strcat, wcscat, \_tcsat, lstrcat, StrCat, StrCatBuff

phương thức có tham số độ dài bộ đệm để không vượt qua phạm vi vùng nhớ.

HRESULT StringCchCat(

\_Inout\_ LPTSTR pszDest,

\_In\_ size\_t cchDest,

\_In\_ LPCTSTR pszSrc

);

* pszDest [in, out]: chuỗi hay bộ đệm cần nối
* cchDest [in]: độ dài chuỗi hay bộ đệm, có tính cả ký tự ‘/0’. Tối đa là STRSAFE\_MAX\_CCH
* pszSrc [in]: chuỗi được nối vào, có ‘/0’

Hàm trả về SUCCEEDED nếu thành công, ngược lại là FAILED

1. Sao chép chuỗi

StringCchCopy nằm trong thư viện <Strsafe.h>, dùng thay cho các phương thức cùng chức năng khác như : strcpy, wcscpy, \_tcscpy, lstrcpy, StrCpy

Vì phương thức có tham số độ dài bộ đệm để không vượt qua phạm vi vùng nhớ.

HRESULT StringCchCopy(

\_Out\_ LPTSTR pszDest,

\_In\_ size\_t cchDest,

\_In\_ LPCTSTR pszSrc

);

* pszDest [out]: chuỗi hay bộ đệm
* cchDest [in]: độ dài chuỗi hay bộ đệm tính theo số kí tự, có tính cả ký tự ‘/0’. Tối đa là STRSAFE\_MAX\_CCH
* pszSrc [in]: Chuỗi được sao chép qua, có chứa ‘/0’

Hàm trả về SUCCEEDED nếu thành công, ngược lại là FAILED

1. Ghi chuỗi vào bộ đệm

StringCchPrintf nằm trong thư viện <Strsafe.h>, thay thế cho các hàm sau: sprintf, swprintf, \_stprintf, wsprintf, wnsprintf, \_snprintf, \_snwprintf, \_sntprintf

Phương thức có tham số kích thước bộ đệm để tránh tràn.

HRESULT StringCchPrintf(

\_Out\_ LPTSTR pszDest,

\_In\_ size\_t cchDest,

\_In\_ LPCTSTR pszFormat,

\_In\_ ...

);

* pszDest [out]: Bộ đệm
* cchDest [in]: kích thước bộ đệm, tính theo số kí tự. Tối đa là STRSAFE\_MAX\_CCH
* pszFormat [in]: chuỗi định dạng tương tự như trên C, phải có ‘\0’.
* ... [in]: các đối số bổ sung tương ứng cho chuỗi định dạng

Hàm trả về SUCCEEDED nếu thành công, ngược lại là FAILED

1. Tìm kí tự từ cuối chuỗi
2. Tìm kí tự từ đầu chuỗi

### Date time trên WinAPI

1. Giới thiệu

WinAPI cung cấp nhiều phương thức date time, làm việc với kiểu cấu trúc riêng biệt tùy theo ngữ cảnh:

|  |  |  |
| --- | --- | --- |
| **Format** | **Kiểu cấu trúc** | **Description** |
| System | [**SYSTEMTIME**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms724950(v=vs.85).aspx) | Thông tin ngày tháng năm giờ phút giây lấy từ đồng hồ phần cứng. |
| Local | [**SYSTEMTIME**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms724950(v=vs.85).aspx) hay [**FILETIME**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms724284(v=vs.85).aspx) | Thông tin thời gian đã chuyển đổi sang mũi giờ địa phương |
| File | [**FILETIME**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms724284(v=vs.85).aspx) | Bội số của 100-nano giây tính từ January 1, 1601. |
| MS-DOS | **WORD** | A packed word for the date, another for the time. |
| Windows | **DWORD** hay **ULONGLONG** | Số mili giây tính từ lúc hệ thống khởi động. When retrieved as a DWORD value, Windows time cycles every 49.7 days. |
| Interrupt Count | **ULONGLONG** | Bội số của 100-nano giây tính từ lúc hệ thống khởi động mới đây. |

1. FILETIME

FILETIME là một giá trị 64-bits, bội số của 100-nano giây tính từ 12:00 A.M January 1, 1601 (UTC).

Hệ thống sẽ cập nhật file time mỗi khi truy xuất file.

NTFS dùng định dạng UTC, trong khi FAT dùng định dạng giờ địa phương của máy tính.

Ví dụ một file lưu lúc 3:00 PM ở Washington. Tại New York, nó hiển thị theo múi giờ New York là được lưu lúc 6:00 PM trên NTFS, nhưng trên FAT nó lại hiển thị là 3:00 PM theo giờ địa phương của Washington.

Cấu trúc FILETIME:

typedef struct \_FILETIME {

DWORD dwLowDateTime;

DWORD dwHighDateTime;

} FILETIME, \*PFILETIME;

* dwLowDateTime: 4-bytes thấp
* dwHighDateTime: 4-bytes cao

1. FileTimeToSystemTime

FILETIME là một giá trị số vô nghĩa. Để chuyển sang định dạng đọc được, ta dùng FileTimeToSystemTime để chuyển sang định dạng SYSTEMTIME (UTC).

BOOL WINAPI FileTimeToSystemTime(

\_In\_ const FILETIME \*lpFileTime,

\_Out\_ LPSYSTEMTIME lpSystemTime

);

* lpFileTime [in]: con trỏ file time.
* lpSystemTime [out]: con trỏ SYSTEMTIME, nhận thông tin trả về

Hàm trả về TRUE nếu thành công, ngược lại FALSE. GetLastError để bug lỗi.

Giá trị file time phải nhỏ hơn 0x8000000000000000, nếu không hàm sẽ FALSE.

1. SetFileTime

Dùng để set Create time, Last Access time và Last Write Time cho file Handle

BOOL WINAPI SetFileTime(

\_In\_ HANDLE hFile,

\_In\_opt\_ const FILETIME \*lpCreationTime,

\_In\_opt\_ const FILETIME \*lpLastAccessTime,

\_In\_opt\_ const FILETIME \*lpLastWriteTime

);

* hFile [in]: file hay thư mục Handle, tạo với quyền FILE\_WRITE\_ATTRIBUTES
* lpCreationTime [in, optional]: con trỏ FILETIME. Để NULL nếu bỏ qua
* lpLastAccessTime [in, optional]: con trỏ FILETIME. Để NULL nếu bỏ qua.   
  Last Access Time bao gồm thời gian cuối cùng file và thư mục ghi, đọc hay thực thi.
* lpLastWriteTime [in, optional]: con trỏ FILETIME. Để NULL nếu bỏ qua

Để ngăn thao tác I/O tự cập nhật last access time, last write time của file handle, chúng ta gọi SetFileTime ngay lập tức sau khi mở file Handle. Set giá trị dwLowDateTime, dwHighDateTime bằng 0xFFFFFFFF.

Hàm trả về TRUE nếu thành công, ngược lại FALSE. GetLastError để bug lỗi.

1. GetFileTime

Dùng để lấy file time từ file Handle

BOOL WINAPI GetFileTime(

\_In\_ HANDLE hFile,

\_Out\_opt\_ LPFILETIME lpCreationTime,

\_Out\_opt\_ LPFILETIME lpLastAccessTime,

\_Out\_opt\_ LPFILETIME lpLastWriteTime

);

* hFile [in]: file hay thư mục Handle, phải có quyền GENERIC\_READ.
* lpCreationTime [out, optional]: con trỏ FILETIME. Để NULL nếu bỏ qua
* lpLastAccessTime [out, optional]: con trỏ FILETIME. Để NULL nếu bỏ qua

Last Access Time bao gồm thời gian cuối cùng file và thư mục ghi, đọc hay thực thi.

* lpLastWriteTime [out, optional]: con trỏ FILETIME. Để NULL nếu bỏ qua

Last Write Time bao gồm thời gian cuối cùng file và thư mục ghi, mở rộng hay bị ghi đè (WriteFile, SetEndOfFile). File Handle chỉ thay đổi thuộc tính sẽ không ảnh hưởng tới tham số này.

Hàm trả về TRUE nếu thành công, ngược lại FALSE. GetLastError để bug lỗi.

1. SYSTEMTIME

Giờ hệ thống là giờ hiện tại lấy từ phần cứng, dùng định dạng UTC (coordianted universal time)

Khi hệ thống khởi động lần đầu tiên, nó set giờ hệ thống theo đồng hồ thực phần cứng.

Cấu trúc SYSTEMTIME:

typedef struct \_SYSTEMTIME {

WORD wYear;

WORD wMonth;

WORD wDayOfWeek;

WORD wDay;

WORD wHour;

WORD wMinute;

WORD wSecond;

WORD wMilliseconds;

} SYSTEMTIME, \*PSYSTEMTIME;

* wYear: Năm, giá trị từ 1601 tới 30827
* wMonth: Tháng, từ 1 tới 12
* wDayOfWeek: Ngày trong tuần, từ 0 tới 6, tương ứng từ Sunday tới Saturday.
* wDay: Ngày trong tháng, từ 1 tới 31
* wHour: Giờ, từ 0 tới 23
* wMinute: Phút, từ 0 tới 59
* wSecond: Giây, từ 0 tới 59
* wMilliseconds: Mili giây từ 0 tới 999

1. GetSystemTime

Dùng để lấy giờ hệ thống theo định dạng system time. Việc hiển thị theo định dạng nào là tùy vào người dùng.

void WINAPI GetSystemTime(

\_Out\_ LPSYSTEMTIME lpSystemTime

);

* lpSystemTime [out]: Con trỏ SYSTEMTIME nhận thông tin trả về, phải hợp lệ, không được để NULL.

Hàm không có giá trị trả về và bug lỗi.

1. GetSystemTimeAsFileTime

Dùng để lấy giờ hệ thống theo định dạng file time

void WINAPI GetSystemTimeAsFileTime(

\_Out\_ LPFILETIME lpSystemTimeAsFileTime

);

lpSystemTimeAsFileTime [out]: Con trỏ FILETIME nhận thông tin trả về theo UTC

1. Window time

Là số mili giây tính từ lúc hệ thống khởi động. Để lấy Window time, chúng ta dùng GetTickCount hay GetTickCount64

Giá trị Window time trả về tính luôn cả thời gian máy tính sleep hay hibernation.

1. GetTickCount

DWORD WINAPI GetTickCount(void);

Hàm trả về số mili giây từ lúc khởi động.

1. Ứng dụng minh họa
2. Thiết lập file time

Mô phỏng command line touch trên Unix: thay đổi thời gian tạo, access và modify của file.

### Quản lý file

Nội dung chương:

* Khả năng truy xuất file ngẫu nhiên: dùng bởi các ứng dụng như cơ sở dữ liệu, trình quản lý file…
* Cách dùng con trỏ file 64-bit để truy xuất file lớn hơn 4GB
* Cách quét toàn bộ nội dung thư mục
* Quản lý, truy xuất thuộc tính file (ngày tạo, lần truy xuất cuối cùng, kích thước…)
* Khóa file khi đang truy xuất để bảo vệ khỏi process khác.
* Truy xuất Registry

1. Truy xuất file ngẫu nhiên
2. Hệ thống 64-bit

File system NTFS đánh địa chỉ file 64-bit, tức 1 file có thể lớn tới 264 –bytes. File system 32-bit (FAT) chỉ hỗ trợ tới 232-byte.

Sự giới hạn này ảnh hưởng tới độ lớn cơ sở dữ liệu, hệ thống media.

File lớn hơn 4GB có thể xem là rất lớn, hiếm khi nào ứng dụng cần một file như vậy.

Win32 sẽ dùng API 32-bit. Sự giới hạn của Win32 sẽ đề cập ở phần khác.

1. Cấu trúc LARGE\_INTEGER

Để tính toán, biểu diễn số 64-bit như con trỏ file, kích thước file, chúng ta dùng kiểu union LARGE\_INTEGER:

typedef union \_LARGE\_INTEGER {

struct {

DWORD LowPart;//2-bytes thấp

LONG HighPart;//2-bytes cao

};

struct {

DWORD LowPart;//2-bytes thấp

LONG HighPart;//2-bytes cao

} u;

LONGLONG QuadPart;//4-byte có dấu

} LARGE\_INTEGER, \*PLARGE\_INTEGER;

Nếu trình biên dịch hỗ trợ 64-bit, chúng ta dùng luôn QuadPart. Nếu không hỗ trợ, có thể dùng thành viên LowPart và HighPart.

1. Con trỏ File

Window, Unix và C library duy trì con trỏ file cho mỗi thao tác mở file, nó cho biết vị trí byte đang đọc.

Vị trí byte thay đổi cho các thao tác đọc, ghi kế tiếp nếu thành công.

Để thay đổi vị trí con trỏ file Handle, chúng ta dùng SetFilePointer và SetFilePointerEx

SetFilePointer ra đời trước tiên, nên không hỗ trợ 64-bits. Tuy nhiên, dùng SetFilePointer sẽ phù hợp và nhanh hơn nếu chỉ xử lí các file nhỏ hơn 4GB.

SetFilePointerEx hỗ trợ con trỏ 64-bit.

lseek trên Unix, và fseek trên standard C có cùng chức năng như SetFilePointer. Nhưng điểm khác biệt là cách sử dụng file pointer với cấu trúc OVERLAPPED trên Window.

1. SetFilePointer

Phiên bản 32-bit, hỗ trợ di chuyển tới 64-bit byte.

DWORD WINAPI SetFilePointer(

\_In\_ HANDLE hFile,

\_In\_ LONG lDistanceToMove,

\_Inout\_opt\_ PLONG lpDistanceToMoveHigh,

\_In\_ DWORD dwMoveMethod

);

* hFile [in]: file Handle với quyền GENERIC\_READ hay GENERIC\_WRITE
* lDistanceToMove [in]: giá trị 32-bit, tính theo byte, bước nhảy cho con trỏ file. Giá trị > 0 sẽ di chuyển lên trước, giá trị <0 sẽ di chuyển ngược lại.
* lpDistanceToMoveHigh [in, out, optional]: Con trỏ giá trị 32-bit.

Trường hợp bạn chỉ muốn di chuyển con trỏ trong khoảng 32-bit giá trị thì để lpDistanceToMoveHigh = NULL.

Trường hợp cần di chuyển 64-bit, phương thức sẽ dùng lDistanceToMove như là byte thấp và lpDistanceToMoveHigh như là byte cao.

* dwMoveMethod [in]: qui định điểm bắt đầu để move con trỏ file

|  |  |
| --- | --- |
| **Value** | **Description** |
| FILE\_BEGIN | Điểm bắt đầu là vị trí đầu tiên của file (zero) |
| FILE\_CURRENT | Điểm bắt đầu là vị trí hiện tại của con trỏ. |
| FILE\_END | Điểm bắt đầu là vị trí cuối cùng của file |

Hàm trả về con trỏ file nếu thành công, ngược lại trả về giá trị 0xFFFFFFFF nếu thất bại.

Dù vậy 0xFFFFFFFF vẫn có thể là con trỏ hợp lệ, chúng ta phải kiểm tra luôn GetLastError để đảm bảo:

* zero hay NO\_ERROR nếu không có lỗi.
* ERROR\_NEGATIVE\_SEEK khi hàm thất bại và trả về giá trị âm cho con trỏ file

1. SetFilePointerEx

Phiên bản 64-bit

BOOL WINAPI SetFilePointerEx(

\_In\_ HANDLE hFile,

\_In\_ LARGE\_INTEGER liDistanceToMove,

\_Out\_opt\_ PLARGE\_INTEGER lpNewFilePointer,

\_In\_ DWORD dwMoveMethod

);

* hFile [in]: file Handle, có quyền GENERIC\_READ hay GENERIC\_WRITE
* liDistanceToMove [in]: giá trị 64-bit, tính theo byte, bước nhảy cho con trỏ file. Giá trị > 0 sẽ di chuyển lên trước, giá trị <0 sẽ di chuyển ngược lại.
* lpNewFilePointer [out, optional]: Con trỏ nhận file pointer trả về. Để NULL nếu không cần trả về.
* dwMoveMethod [in]: qui định điểm bắt đầu để move con trỏ file

|  |  |
| --- | --- |
| **Value** | **Description** |
| FILE\_BEGIN | Điểm bắt đầu là vị trí đầu tiên của file (zero) |
| FILE\_CURRENT | Điểm bắt đầu là vị trí hiện tại của con trỏ. |
| FILE\_END | Điểm bắt đầu là vị trí cuối cùng của file |

Hàm trả về TRUE nếu thành công, ngược lại FALSE nếu thất bại. GetLastError để bug lỗi.

1. Định vị bằng OVERLAPPED

WinAPI vị trí file qua cấu trúc OVERLAPPED được truyền như tham số cuối cùng trong ReadFile, WriteFile.

OVERLAPPED có tất cả 5 thàm viên, thông tin vị trí nằm trong hai dữ liệu thành viên **Offset**, **OffsetHigh**.

Chúng ta có thể đặt giá trị cho chúng, tiến trình I/O sẽ bắt đầu từ vị trí này.

Để ghi tiếp vào cuối file, chỉ cần set 0xFFFFFFFF cho cả **Offset**, **OffsetHigh.**

Các tiến trình I/O trên file chỉ làm thay đổi giá trị con trỏ file, nhưng cấu trúc OVERLAPPED vẫn giữ nguyên.

Thành viên cuối cùng là Handle **hEvent** chỉ dùng riêng cho Async overlapped I/O, sẽ đề cập chương khác.

Khi dùng với Sync I/O, chúng ta để hEvent = NULL.

Cấu trúc OVERLAPPED đặc biệt thuận tiện khi cập nhật bản ghi file, so với việc phải gọi SetFilePointerEx nhiều lần khi đọc, ghi file.

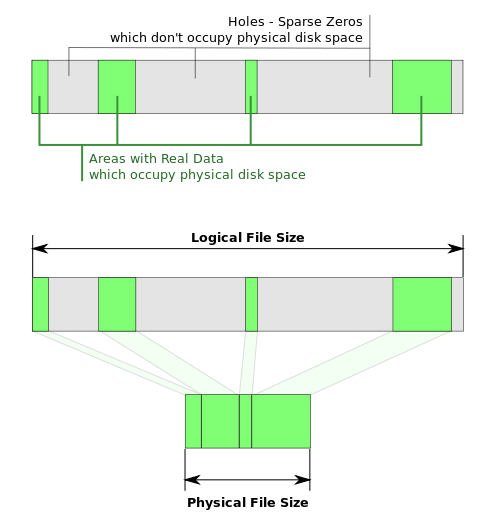
1. Sparse Files

Một file có nhiều vùng dữ liệu zero được xem là chứa sparse data. Những file như vậy thường có kích thước rất lớn, như là file image hay file cơ sở dữ liệu.

Vấn đề đối với sparse file là việc lưu trữ trên đĩa cứng không hiệu quả. NTFS cung cấp 2 giải pháp:

* Tính năng nén file: làm tất cả dữ liệu của file không qua thao tác ghi được set zero. Việc nén file sẽ thu gọn các vùng dữ liệu zero, nhưng bù lại, tốc độc truy xuất sẽ giảm vì phải giải nén.
* Tính năng sparse: hệ thống sẽ không cấp phát vùng đĩa cứng cho dữ liệu zero trong file. Hiệu quả hơn so với nén file.

Khi sparse file kích hoạt: File system sẽ bỏ qua khi thao tác ghi chuẩn bị cho một lượng lớn dữ liệu zero, thay vào đó nó tạo ra một danh sách chứa vị trí zero trong file.

Khi thao tác đọc đi đến vùng dữ liệu zero, file system tham khảo danh sách và trả về tương ứng dữ liệu zero vào bộ đệm đọc.

Giá trị mặc định của dữ liệu sparse trong file là zero, tuy nhiên có thể là giá trị khác.

GetCompressedFileSize trả về dung lượng trên đĩa của sparse file, kích thước này không bao gồm vùng zero data.

GetFileSizeEx trả về file size, bao gồm vùng zero data.

Khi dùng SetFilePointer và SetEndOfFile để mở rộng file sparse, phần nằm giữa phần kết cũ và mới sẽ là vùng sparse zero data.

Hàm SetEndOfFile làm thay đổi kích thước vật lý đối với file thường, và thay đổi kích thước logic với file sparse.

Với sparse file, không gian đĩa cứng chỉ cấp cho dữ liệu được ghi thực sự. Administrator có thể set sparse cho bất kì file, thư mục, ổ đĩa nào.

Chúng ta có thể dùng thao tác ghi file để mở rộng file, nhưng sẽ làm phân mảnh vùng đĩa cứng chứa file

Hầu hết file system hiện nay đều hỗ trợ sparse file như Unix, Apple File System, OS X.

1. Sự phân mảnh sparse file

Báo cáo dung lượng trống từ File System sẽ không chính xác vì phân mảnh bên trong sparse file, hiệu ứng là lỗi ổ đĩa đầy hay vượt quá dung lượng hạn ngạch khi ghi đè sparse file, và sao chép sparse file với ứng dụng không hỗ trợ sparse mà sử dụng toàn bộ kích thước không nén, bao gồm vùng zero data không thực sự nằm trên đĩa cứng – làm mất đi lợi ích của sparse file.

1. Sparse file và phân ngạch dung lượng

Sparse file ảnh hưởng tới hạn ngạch user bởi kích thước logic file, chứ không phải kích thước vật lý.

Khi user ghi dữ liệu vào sparse file, user không để tâm về khả năng vượt quá hạn ngạch, 50 MB file với toàn bộ dữ liệu zero sẽ làm user mất đúng 50 MB.

System Administrator không nên dựa vào kích thước vật lý của sparse file , không nên cho phép user sử dụng vượt quá dung lượng logic còn trống.

1. Tiến trình ghi Sparse file

Để biết file system hỗ trợ sparse, gọi hàm GetVolumeInformation, kiểm tra thành viên lpFileSystemFlags với FILE\_SUPPORTS\_SPARSE\_FILES.

Kiểm tra file với thuộc tính FILE\_ATTRIBUTE\_SPARSE\_FILE để nhận biết sparse file.

Dùng hàm DeviceIOControl với flag FSCTL\_SET\_SPARSE để set sparse cho Handle file.

Dùng FSCTL\_QUERY\_ALLOCATED\_RANGES để lấy các vùng non-zero trong sparse file.

Trường hợp ứng dụng muốn set thuộc tính sparse cho file đã tồn tại: Set thuộc tính sparse cho file. Sau đó quét và dùng FSCTL\_SET\_ZERO\_DATA cho các vùng sparse data. Các vùng sparse sẽ bị thu hồi vùng nhớ trên đĩa cứng.

Trong quá trình đọc sparse file, hệ thống sẽ trả về bộ đệm zero cho vùng sparse mà không thao tác gì trên đĩa cứng.

Sparse file cũng có thể truy xuất ngẫu nhiên, dữ liệu non-zero ghi vào vùng sparse sẽ được ghi lên đĩa cứng. Dữ liệu zero ghi vào vùng non-zero (phải có FSCTL\_SET\_ZERO\_DATA) sẽ thu hồi vùng lưu trữ trên đĩa cho vùng này.

Các công cụ chống phân mảnh phải xử lí được file nén cũng như sparse file trên trên NTFS.

1. GetVolumeInformation

Chúng ta chỉ cần quan tâm tham số lpFileSystemFlags, kiểm tra với flag :

FILE\_FILE\_COMPRESSION : File system có hỗ trợ nén

FILE\_SUPPORTS\_SPARSE\_FILES: File system có hỗ trợ sparse

BOOL WINAPI GetVolumeInformation(

\_In\_opt\_ LPCTSTR lpRootPathName,

\_Out\_opt\_ LPTSTR lpVolumeNameBuffer,

\_In\_ DWORD nVolumeNameSize,

\_Out\_opt\_ LPDWORD lpVolumeSerialNumber,

\_Out\_opt\_ LPDWORD lpMaximumComponentLength,

\_Out\_opt\_ LPDWORD lpFileSystemFlags,

\_Out\_opt\_ LPTSTR lpFileSystemNameBuffer,

\_In\_ DWORD nFileSystemNameSize

);

* lpRootPathName [in, optional]: Chuỗi đường dẫn chứa thư mục gốc, phải có “\” ở cuối. Ví dụ: \\MyServer\MyShare là "\\MyServer\MyShare\", ổ đĩa C là "C:\”
* nVolumeNameSize [in]: Độ dài ký tự cho tên Volume. Set tối đa MAX\_PATH+1 cho ‘\0’. Để 0 cho bộ đệm NULL.
* nFileSystemNameSize [in]: Độ dài ký tự cho tên file system. Set tối đa MAX\_PATH+1 cho ‘\0’. Để 0 cho bộ đệm NULL

Các tham số khác để là NULL.

Nếu để NULL, thư mục gốc của ứng dụng được dùng.

1. FSCTL\_SET\_SPARSE

DeviceIOControl sẽ gửi code kiểm soát tới driver thiết bị, qui định hành vi cho thiết bị hoạt động.

Chúng ta dùng DeviceIOControl để set hay clear thuộc tính FILE\_ATTRIBUTE\_SPARSE\_FILE cho handle file.

Tham số thứ hai **dwIoControlCode** = **FSCTL\_SET\_SPARSE**, định nghĩa Handle file là sparse, dữ liệu rỗng trong file sẽ không được ghi vào địa cứng.

BOOL DeviceIoControl( (HANDLE) hDevice, // handle to a file

FSCTL\_SET\_SPARSE, // dwIoControlCode

(PFILE\_SET\_SPARSE\_BUFFER) lpInBuffer, // input buffer

(DWORD) nInBufferSize, // size of input buffer

NULL, // lpOutBuffer

0, // nOutBufferSize

(LPDWORD) lpBytesReturned, // number of bytes returned

(LPOVERLAPPED) lpOverlapped // OVERLAPPED structure

);

* hDevice[in]: Handle file trả về từ CreateFile, phải có thuộc tính bảo mật FILE\_FLAG\_BACKUP\_SEMANTICS.
* dwIoControlCode[in]: Mã kiểm soát thiết bị, set FSCTL\_SET\_SPARSE
* lpInBuffer[in, optional]: bộ đệm kiểu FILE\_SET\_SPARSE\_BUFFER. Có thể NULL nếu tiến trình không cần dữ liệu đầu vào.

Khi lpInBuffer = NULL, hay khi thành viên SetSparse của cấu trúc FILE\_SET\_SPARSE\_BUFFER là TRUE, DeviceIOControl cũng set handle file là sparse file.

* nInBufferSize[in]: kích thước bộ đệm lpInBuffer theo bytes. Nếu lpInBuffer= NULL thì nInBufferSize = 0.
* lpOutBuffer[out, option]: không dùng trong sparse file, set NULL
* nOutBufferSize[in]: set 0
* lpBytesReturned[out, option]: Trả về số bytes dữ liệu ghi vào lpOutBuffer.

\_ Nếu lpOverlapped = NULL: lpBytesReturned không được NULL.

\_ Nếu lpOverlapped != NULL: lpBytesReturned có thể để NULL. Nếu lpBytesReturned!=NULL và tiến trình có trả về dữ liệu, lpBytesReturned vô nghĩa tới khi tiến trình overlapped hoàn tất. Dùng GetOverlappedResult để lấy số byte trả về.

* lpOverlapped[in,out, option]: Con trỏ OVERLAPPED.

\_ Nếu hDevice không phải Async I/O, để lpOverlapped = NULL.

\_ Nếu hDevice là Async I/O với flag FILE\_FLAG\_OVERLAPPED, thì lpOverlapped phải hợp lệ. DeviceIoControl sẽ return về ngay khi Handle Event trong lpOverlapped được thông báo khi tiến trình hoàn tất, ngược lại DeviceIoControl sẽ không return cho tới khi tiến trình hoàn tất hay xảy ra lỗi.

Hàm trả về TRUE nếu thành công, ngược lại FALSE nếu thất bại. GetLastError để bug lỗi.

Windows Server 2008 R2, Windows 7, Windows Server 2008 và Windows Vista: việc hủy một tiến trình trên file với vùng sparse có thể gây lỗi.

Để kiểm tra vùng sparse trong file, dùng mã kểm soát **FSCTL\_QUERY\_ALLOCATED\_RANGES**

1. FSCTL\_SET\_ZERO\_DATA

Dùng để chuyển một vùng file sang vùng zero. Nếu file là sparse hay nén, NTFS sẽ thu hồi vùng đĩa cứng cho vùng zero data.

FSCTL\_SET\_ZERO\_DATA cũng có thể dùng cho non-sparse file nhưng không vùng zero không bị thu hồi.

BOOL WINAPI DeviceIoControl(

(HANDLE) hDevice, // handle to a file

FSCTL\_SET\_ZERO\_DATA, // dwIoControlCode

(LPVOID) lpInBuffer, // input buffer

(DWORD) nInBufferSize, // size of input buffer

NULL, // lpOutBuffer

0, // nOutBufferSize

(LPDWORD) lpBytesReturned, // number of bytes returned

(LPOVERLAPPED) lpOverlapped // OVERLAPPED structure

);

* lpInBuffer[in, optional]: Bộ đệm cấu trúc FILE\_ZERO\_DATA\_INFORMATION, định nghĩa vùng zero trong file.

typedef struct \_FILE\_ZERO\_DATA\_INFORMATION {

LARGE\_INTEGER FileOffset;

LARGE\_INTEGER BeyondFinalZero;

} FILE\_ZERO\_DATA\_INFORMATION, \*PFILE\_ZERO\_DATA\_INFORMATION;

\_ Thành viên FileOffset chỉ vị trí byte bắt đầu của vùng zero data.

\_ Thành viên BeyondFinalZero chỉ vị trí byte kết thúc vùng zero data.

1. FSCTL\_QUERY\_ALLOCATED\_RANGES

Mã kiểm soát dùng để quét sparse file cho vùng cấp phát vật lý trên đĩa cứng.

BOOL DeviceIoControl(

(HANDLE) hDevice, // handle to file

FSCTL\_QUERY\_ALLOCATED\_RANGES, // dwIoControlCode

(LPVOID) lpInBuffer, // input buffer

(DWORD) nInBufferSize, // size of input buffer

(LPVOID) lpOutBuffer, // output buffer

(DWORD) nOutBufferSize, // size of output buffer

(LPDWORD) lpBytesReturned, // number of bytes returned

(LPOVERLAPPED) lpOverlapped // OVERLAPPED structure

);

Các tham số đáng chú ý:

* lpInBuffer[in, optional]: Con trỏ cấu trúc FILE\_ALLOCATED\_RANGE\_BUFFER, cho biết vùng file cần quét.

typedef struct \_FILE\_ALLOCATED\_RANGE\_BUFFER {

LARGE\_INTEGER FileOffset;

LARGE\_INTEGER Length;

} FILE\_ALLOCATED\_RANGE\_BUFFER, \*PFILE\_ALLOCATED\_RANGE\_BUFFER;

\_ Thành viên FileOffset đánh dấu vị trí byte bắt đầu.

\_ Thành viên Length cho biết độ dài vùng quét.

* nInBufferSize[in, optional]: Kích thước bộ đệm lpInBuffer tính theo byte.
* lpOutBuffer[out]: Con trỏ mảng FILE\_ALLOCATED\_RANGE\_BUFFER, mỗi phần tử cho biết vùng cấp phát vật lý trong file. Các vùng trả về đều nằm gọn trong phạm vi qui định bởi lpInBuffer
* nOutBufferSize[out]: kích thước theo byte của lpOutBuffer

Khi lpOutBuffer không đủ cho dữ liệu trả về, lpOutBuffer vẫn lưu dữ liệu nhận được, hàm trả về FALSE và GetLastError = ERROR\_MORE\_DATA, lpBytesReturned cho biết số bytes lpOutBuffer nhận được. Chúng ta nên gọi DeviceIoControl cho tới khi trả về TRUE.

1. GetFileSizeEx

Chúng ta có thể dùng SetFilePointerEx với flag FILE\_END và liDistanceToMove = 0, giá trị file pointer sẽ là kích thước file.

Ngoài ra có thể dùng GetFileSizeEx:

BOOL WINAPI GetFileSizeEx(

\_In\_ HANDLE hFile,

\_Out\_ PLARGE\_INTEGER lpFileSize

);

* hFile [in]: file Handle, có set thuộc tính FILE\_READ\_ATTRIBUTES hay User có quyền truy xuất thư mục chứa file.
* lpFileSize [out]: Con trỏ kiểu file pointer nhận kích thước theo bytes.

Hàm trả về TRUE nếu thành công, ngược lại là FALSE. GetLastError để bug lỗi.

1. Thay đổi File Size

SetEndOfFile thiết lập kích thước cho file bằng cách sử dụng vị trí con trỏ file hiện tại. Nó đẩy giá trị phần kết file ra xa hơn.

Kích thước file được xem như điểm kết thúc của file.

BOOL WINAPI SetEndOfFile(

\_In\_ HANDLE hFile

);

* hFile [in]: File Handle cần mở rộng hay nối dài. Phải có quyền GENERIC\_WRIT

Hàm trả về TRUE nếu thành công, ngược lại là FALSE. GetLastError để bug lỗi.

SetEndOfFile có thể nối dài hay mở rộng file. Sau khi đã mở rộng dung lượng, nội dung file bắt đầu từ phần kết trước đó đến phần kết sau khi mở rộng không xác định được.

Mỗi file stream gồm:

* File size: là tổng kích thước logic dữ liệu theo byte trong file, bao gồm vùng sparse data
* Allocation size: kích thước vật lý trên đĩa của file, luôn là bội số của cluster, không bao gồm vùng sparse data
* Valid data length: kích thước logic dữ liệu thực của file được ghi, tính theo byte. Giá trị này luôn nhỏ hơn hay bằng File size.

SetEndOfFile thay đổi file size. SetFileValidData thay đổi data length.

1. Thay đổi File data

Lưu ý: không áp dụng cho file nén, sparse hay database, network

BOOL WINAPI SetFileValidData(

\_In\_ HANDLE hFile,

\_In\_ LONGLONG ValidDataLength

);

* hFile [in]: chỉ áp dụng với file Handle có quyền GENERIC\_WRITE và thuộc tính bảo mật SE\_MANAGE\_VOLUME\_NAME.
* ValidDataLength [in]: kích thước mới của dữ liệu trong file. Nên lớn hơn kích thước cũ, và phải nhỏ hơn kích thước vật lý File size.

Hàm trả về TRUE nếu thành công, ngược lại là FALSE. GetLastError để bug lỗi.

SetFileValidData ngăn việc ghi zero data vào file, giúp dữ liệu bên trong file được hợp lệ mà không cần phải ghi. Như vậy tình cờ dữ liệu đã tồn tại trên đĩa từ file cũ lại có ích.

Việc sử dụng SetFileValidData có liên quan đến tính bảo mật của hệ thống.

Đầu tiên user phải có quyền SE\_MANAGE\_VOLUME\_NAME khi mở file. Ứng dụng chỉ nên gọi SetFileValidData trên file có hạn chế quyền truy xuất đối với SE\_MANAGE\_VOLUME\_NAME.

Ứng dụng phải đảm bảo vùng chưa ghi của file được bảo vệ: nếu SetFileValidData

If SetFileValidData is used on a file, the potential performance gain is obtained by not filling the allocated clusters for the file with zeros. Therefore, reading from the file will return whatever the allocated clusters contain, potentially content from other users. This is not necessarily a security issue at this point, because the caller needs to have SE\_MANAGE\_VOLUME\_NAME privilege for SetFileValidData to succeed, and all data on disk can be read by such users. However, this caller can inadvertently expose this data to other users that cannot acquire the SE\_MANAGE\_VOLUME\_PRIVILEGE privilege if the following holds:

If the file was not opened with a sharing mode that denies other readers, a nonprivileged user can open it and read the exposed data.

If the system stops responding before the caller finishes writing up the ValidDataLength supplied in the call, then, on a reboot, such a nonprivileged user can open the file and read exposed content.

If the caller of SetFileValidData opened the file with adequately restrictive access control, the previous conditions would not apply. However, for partially written files extended with SetFileValidData (that is, writing was not completed up to the ValidDataLength supplied in the call) there exists yet another potential privacy or security vulnerability. An administrator could copy the file to a target that is not properly controlled with restrictive ACL permissions, thus inadvertently exposing the extended area's data to unauthorized reading.

It is for these reasons that SetFileValidData is not recommended for general purpose use, in addition to performance considerations, as discussed below.

Dùng SetFileValidData để xử lí file lớn trong nhiều trường hợp sẽ có hiệu suất tốt hơn phương thức khác như SetEndOfFile.

Đặc biệt, khi phần mở rộng dung lượng của file lớn và được ghi ngẫu nhiên, như kiểu database.

Xét trung bình thì hiệu suất của SetFileValidData không cao. Chỉ nên dùng cho trường hợp đặc biệt.

1. Tìm kiếm file và thư mục
2. Giới thiệu

Phần này giới thiệu cách tìm kiếm file trên nhiều thư mục, tên file tìm kiếm theo một mẫu đặc biệt, và chúng ta cũng lấy luôn thông tin thuộc tính của file.

WinAPI cung cấp phương thức tìm kiếm là FindFirstFile, hàm có thể tìm theo ký tự đại diện với (\*,?) cho ra nhiều kết quả. Thông tin file trả về chứa trong cấu trúc WIN32\_FIND\_DATA.

Để lấy thông tin file kế tiếp ta gọi FindNextFile với tham số Handle tìm kiếm trả về từ FindFirstFile hay FindFirstFileEx.

Handle tìm kiếm này không phải kernel Handle nên không dùng CloseHandle, mà dùng FindClose để thu hồi tài nguyên.

Phiên bản sau FindFirstFileEx cung cấp nhiều tính năng hơn như phân biệt chữ hoa thường.

Khi làm việc với Window 10, bản unicode FindFirstFileW, có thể vô hiệu giới hạn của MAX\_PATH.

Thông tin thời gian ftLastWriteTime, chỉ cập nhật khi file bị ghi, ghi đè (khi dùng WriteFile, SetEndOfFile), khi thay đổi thuộc tính và thông tin bảo mật, thông số này không bị thay đổi.

1. Cấu trúc WIN32\_FIND\_DATA

FindFirstFileEx và FindFirstFile trả về kết quả vào cấu trúc WIN32\_FIND\_DATA, chứa thông tin về file hay thư mục tìm thấy đầu tiên.

Tất cả thành viên ftCreationTime ,ftLastAccessTime , ftLastWriteTime sẽ là zero nếu hệ thống không hỗ trợ time.

Lưu ý khi thuộc tính hay security descriptor thay đổi, time sẽ không được cập nhật.

typedef struct \_WIN32\_FIND\_DATA {

DWORD dwFileAttributes;

FILETIME ftCreationTime;

FILETIME ftLastAccessTime;

FILETIME ftLastWriteTime;

DWORD nFileSizeHigh;

DWORD nFileSizeLow;

DWORD dwReserved0;

DWORD dwReserved1;

TCHAR cFileName[MAX\_PATH];

TCHAR cAlternateFileName[14];

} WIN32\_FIND\_DATA, \*PWIN32\_FIND\_DATA, \*LPWIN32\_FIND\_DATA;

* dwFileAttributes: thuộc tính của file.

Thuộc tính FILE\_ATTRIBUTE\_SPARSE\_FILE tự động set nếu có bất kì stream file nào được sparse. dwFileAttributes có thêm vài giá trị mà tham số thuộc tính file trong CreateFile không có.

* ftCreationTime: kiểu FILETIME, thời gian file hay thư mục được tạo.
* ftLastAccessTime: kiểu FILETIME, thời gian file truy xuất gần đây. Đối với thư mục chỉ là thời gian tạo
* ftLastWriteTime: kiểu FILETIME, thời gian file được ghi gần đây. Đối với thư mục chỉ là thời gian tạo
* nFileSizeHigh: 4-byte cao của kích thước file.
* nFileSizeLow: 4-byte thấp của kích thước file.

Công thức tính file size: (nFileSizeHigh \* (MAXDWORD+1)) + nFileSizeLow

* dwReserved0:

If the dwFileAttributes member includes the FILE\_ATTRIBUTE\_REPARSE\_POINT attribute, this member specifies the reparse point tag.

Otherwise, this value is undefined and should not be used.

Tham khảo thêm tại [Reparse Point Tags](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365511(v=vs.85).aspx).

IO\_REPARSE\_TAG\_CSV (0x80000009)

IO\_REPARSE\_TAG\_DEDUP (0x80000013)

IO\_REPARSE\_TAG\_DFS (0x8000000A)

IO\_REPARSE\_TAG\_DFSR (0x80000012)

IO\_REPARSE\_TAG\_HSM (0xC0000004)

IO\_REPARSE\_TAG\_HSM2 (0x80000006)

IO\_REPARSE\_TAG\_MOUNT\_POINT (0xA0000003)

IO\_REPARSE\_TAG\_NFS (0x80000014)

IO\_REPARSE\_TAG\_SIS (0x80000007)

IO\_REPARSE\_TAG\_SYMLINK (0xA000000C)

IO\_REPARSE\_TAG\_WIM (0x80000008)

* dwReserved1: không dùng
* cFileName: Tên file
* cAlternateFileName: Tên thay thế của file, theo định dạng 8.3, thường dùng cho FAT16

1. FindFirstFile

HANDLE WINAPI FindFirstFile(

\_In\_ LPCTSTR lpFileName,

\_Out\_ LPWIN32\_FIND\_DATA lpFindFileData

);

* lpFileName [in]: Thư mục , đường dẫn và tên file. Có thể chứa kí tự đại diện như (\*) hay (?)

*path\\** : tìm tất cả file hay thư mục trong path\

*path\\*.txt* : tìm tất cả file có phần mở rộng .txt

*path\prefix\*.\** : tìm tất cả file có tên bắt đầu là “prefix”

Tham số này không nên là NULL, và phải hợp lệ . User phải có quyền truy xuất tới root hay thư mục con.

* lpFindFileData [out]: con trỏ kiểu WIN32\_FIND\_DATA nhận thông tin trả về.

Hàm trả về Handle tìm kiếm nếu thành công, Handle này dùng cho FindNextFile hay FindClose.

Hàm trả về INVALID\_HANDLE\_VALUE nếu thất bại.

GetLastError trả về ERROR\_FILE\_NOT\_FOUND nếu không tìm thấy.

1. FindNextFIle

BOOL WINAPI FindNextFile(

\_In\_ HANDLE hFindFile,

\_Out\_ LPWIN32\_FIND\_DATA lpFindFileData

);

* hFindFile [in]: Handle tìm kiếm trả về từ FindFirstFile hay FindFirstFileEx
* lpFindFileData [out]: con trỏ kiểu WIN32\_FIND\_DATA nhận thông tin trả về

Hàm trả về TRUE nếu thành công, ngược lại FALSE.

GetLastError trả về ERROR\_NO\_MORE\_FILES nếu không tìm thấy nữa.

1. FindClose

Lưu ý không thể dùng CloseHandle cho Handle tìm kiếm, vì nó không phải kernel Handle.

Chúng ta dùng FindClose

BOOL WINAPI FindClose(

\_Inout\_ HANDLE hFindFile

);

* hFindFile [in, out]: Handle tìm kiếm.

Hàm trả về TRUE nếu thành công, ngược lại FALSE. GetLastEror để bug lỗi.

1. GetFileInformationByHandle

Một cách để lây thông tin file khi đã có file Handle là dùng GetFileInformationByHandle:

BOOL WINAPI GetFileInformationByHandle(

\_In\_ HANDLE hFile,

\_Out\_ LPBY\_HANDLE\_FILE\_INFORMATION lpFileInformation

);

* hFile [in]: file Handle
* lpFileInformation [out]: con trỏ kiểu BY\_HANDLE\_FILE\_INFORMATION, chứa thông tin trả về

Hàm trả về TRUE nếu thành công ,ngược lại FALSE. GetLastError để bug lỗi.

Cấu trúc BY\_HANDLE\_FILE\_INFORMATION có nhiều thành viên tương tự như WIN32\_FIND\_DATA, có vài trường đáng chú ý:

* dwVolumeSerialNumber: số seri ổ đĩa chứa file.
* nNumberOfLinks: số hard link của file tạo ra từ CreateHardLink.

1. Thuộc tính file và thư mục
2. GetFileTime

Lấy thông tin về time của file hay thư mục

BOOL WINAPI GetFileTime(

\_In\_ HANDLE hFile,

\_Out\_opt\_ LPFILETIME lpCreationTime,

\_Out\_opt\_ LPFILETIME lpLastAccessTime,

\_Out\_opt\_ LPFILETIME lpLastWriteTime

);

* hFile [in]: File Handle với quyền GENERIC\_READ
* lpCreationTime [out, optional]: con trỏ file time nhận thời gian tạo.
* lpLastAccessTime [out, optional]: con trỏ file time nhận thời gian truy xuất gần đây.
* lpLastWriteTime [out, optional] con trỏ file time nhận thời gian ghi gần đây nhất

Các tham số file time lpCreationTime, lpLastAccessTime, lpLastWriteTime có thể để NULL, nếu bạn không quan tâm.

Hàm trả về TRUE nếu thành công ,ngược lại FALSE. GetLastError để bug lỗi.

1. GetFileAttributes

Lấy thông tin thuộc tính của file

DWORD WINAPI GetFileAttributes(

\_In\_ LPCTSTR lpFileName

);

* lpFileName [in]: tên file

Hàm trả về giá trị gồm các Macro thuộc tính được set nếu thành công.

Ngược lại, trả về INVALID\_FILE\_ATTRIBUTES nếu thất bại. GetLastError để bug lỗi.

1. SetFileAttributes

Thiết lập thuộc tính mới cho file

BOOL WINAPI SetFileAttributes(

\_In\_ LPCTSTR lpFileName,

\_In\_ DWORD dwFileAttributes

);

* lpFileName [in]: tên file
* dwFileAttributes [in]: Tập thuộc tính set cho file.

Hàm trả về TRUE nếu thành công ,ngược lại FALSE. GetLastError để bug lỗi.

1. Duyệt và Set thuộc tính

Chúng ta thể lấy thuộc tính file từ GetFileAttributes hay GetFileAttributesEx.

CreateFile và SetFileAttributes có thể set nhiều thuộc tính. Tuy nhiên, ứng dụng không thể set hết thuộc tính được.

Ứng dụng minh họa dùng CopyFile để sao chép tất cả file trong thư mục hiện hành tới thư mục mới.

Ứng dụng tạo thư mục mới tương ứng với tham số command line. (thư mục mới không được tồn tại)

Dùng FindFirstFile và FindNextFile để tìm và duyệt tất cả file theo tham số tìm kiếm path/search\_text.

Cú pháp: -copyb path/search\_text newDir

Nếu file gốc không phải read-only, thì chuyển sang thư mục mới newDir set file sao chép sang read-only.

The application searches the current directory for all text files by using the FindFirstFile and FindNextFile functions. Each text file is copied to the \TextRO directory. After a file is copied, the GetFileAttributes function determines whether or not a file is read only. If the file is not read only, the application changes directories to \TextRO and converts the copied file to read only by using the SetFileAttributes function.

After all text files in the current directory are copied, the application closes the search handle by using the FindClose function.

1. File temp
2. Giới thiệu

Window tạo temp file trong 3 trường hợp sau:

1. Ứng dụng ghi file như Excel chẳng hạn: nó tạo temp file cho phép người dùng undo các thao tác.
2. Ứng dụng MS-DOS: để có thể swap sang đĩa cứng, tiết kiệm vùng nhớ cho việc khác.
3. Khi Print tài liệu với spooler được kích hoạt: file temp lưu trên đĩa cứng, Window spools print job vào file temp, sau đó gửi tới máy in.

Tối thiểu file temp cần 2 MB trên đĩa cứng.

Tên file temp có tiền tố “~”, phần mở rộng là .TMP.

File temp bị đóng và xóa bởi Window khi kết thúc phiên làm việc. Nếu Window bị tắt ngang, file temp sẽ không bị xóa.

Phần thiết lập thư mục mặc định của file temp nằm trong AUTOEXEC.BAT:

SET TEMP=C:\WINDOWS\TEMP

Nếu SET TEMP chỉ đến ổ đĩa RAM, đảm bảo RAM drive đủ bộ nhớ.

1. GetTempFileName

Hàm dùng để lấy tên file temp dùng cho việc tạo file trong CreateFile.

UINT WINAPI GetTempFileName(

\_In\_ LPCTSTR lpPathName,

\_In\_ LPCTSTR lpPrefixString,

\_In\_ UINT uUnique,

\_Out\_ LPTSTR lpTempFileName

);

* lpPathName [in]: đường dẫn thư mục chứa file temp. ký tự “.” đại diện cho thư mục hiện hành.
* lpPrefixString [in]: chuỗi ký tự tiền tố để tìm tên file, tối đa là 3 kí tự. Chuỗi phải có ‘/0’
* uUnique [in]: thường để zero để hàm tự sinh ra file temp dựa trên giờ hệ thống.
* lpTempFileName [out]: bộ đệm nhận tên file, tối đa là MAX\_PATH + 1 cho ‘/0’.

Hàm trả về số độc nhất dùng cho tên file temp nếu thành công. Nếu uUnique khác zero, hàm sẽ trả về cùng giá trị này.

Hàm trả về zero nếu thất bại. GetLastError trả về ERROR\_BUFFER\_OVERFLOW nếu độ dài lpPathName lớn hơn MAX\_PATH–14

Tên file temp tạo ra có định dạng: **<path>\<pre><uuuu>.TMP**

* <path> : Đường dẫn lpPathName
* <pre> : Kí tự tiền tố từ lpPrefixString
* <uuuu> : 4 kí tự là giá trị hexa từ uUnique

hệ thống không tự động xóa File temp tạo bởi hàm GetTempFileName, phải gọi DeleteFile để xóa

1. GetTempPath

Trả về đường dẫn thư mục chứa temp file.

DWORD WINAPI GetTempPath(

\_In\_ DWORD nBufferLength,

\_Out\_ LPTSTR lpBuffer

);

* nBufferLength [in]: độ dài bộ đệm
* lpBuffer [out]: bộ đệm chuỗi TCHAR nhận đường dẫn trả về, có ký tự phân cách “\” ở cuối, ví dụ "C:\TEMP\"

Số ký tự tối đa vẫn là MAX\_PATH+1.

Nếu thành công, hàm trả về độ dài chuỗi không bao gồm ‘\0’. Nếu độ dài chuỗi trả về dài hơn bộ đệm, hàm trả vẫn trả về độ dài chuỗi.

Nếu thất bại, hàm trả về zero. GetLastError để bug lỗi.

GetTempPath chỉ kiểm tra các biến môi trường theo thứ tự sau:

1. TMP environment variable.
2. TEMP environment variable.
3. USERPROFILE environment variable.
4. Thư mục Windows.

GetTempPath không xác nhận đường dẫn tồn tại hay kiểm tra process có quyền truy xuất đường dẫn hay không.

Ứng dụng nên kiểm tra đường dẫn có thực sự tồn tại hay không, và process có quyền truy xuất và thực thi I/O không.

1. Sử dụng temp file

Chúng ta sẽ demo ứng dụng sao chép dữ liệu dùng file temp.

Cú pháp command line: -copyf Input Output

Thông qua GetTempFileName và GetTempPath, chúng ta lấy được tên file temp và đường dẫn.

Dùng CreateFile để tạo file temp.

Ứng dụng đọc các dữ liệu Input vào bộ đệm, ghi toàn nội dung bộ đệm vào file temp.

Sau đó đóng 2 file lại, đổi tên file temp sang tên file Output và dùng MoveFileEx để move sang đường dẫn mới.

Ứng dụng nên có giải pháp khi temp path không tồn tại hay user không có quyền truy xuất temp path.

1. File locking
2. Khóa file

Vấn đề phát sinh trên máy tính chạy nhiều process là việc truy xuất tài nguyên dùng chung, như file.

Window có thể khóa file, toàn bộ hay một phần nội dung, không cho process khác hay thread có thể truy xuất.

File đã khóa chỉ có thể đọc (khi chia sẻ) hay ghi trên vùng được phép (thao tác read write trên cùng process). Quan trọng hơn, khóa thuộc về process, bất kì cố gắng truy xuất tới phần nội dung file đã khóa hay ý định khóa file sẽ thất bại.

Nếu process sở hữu khóa file, file sẽ bị giới hạn khi đồng bộ giữa các process và thread chạy song song.

Phương thức phổ biến nhất là LockFileEx, và phiên bản 32-bit LockFile

LockFileEx là thành viên của lớp I/O mở rộng.

Cấu trúc OVERLAPPED được dùng để xác định vị trí file và WriteFile, và còn để xác định vùng file sẽ bị khóa.

1. LockFileEx

Khóa file Handle bởi process hiện hành. Phương thức hoạt động cho cả Sync và Async I/O

BOOL WINAPI LockFileEx(

\_In\_ HANDLE hFile,

\_In\_ DWORD dwFlags,

\_Reserved\_ DWORD dwReserved,

\_In\_ DWORD nNumberOfBytesToLockLow,

\_In\_ DWORD nNumberOfBytesToLockHigh,

\_Inout\_ LPOVERLAPPED lpOverlapped

);

* hFile [in]: File Handle có quyền GENERIC\_READ hoặc GENERIC\_WRITE
* dwFlags [in]: Gồm 2 tham số:

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| **LOCKFILE\_EXCLUSIVE\_LOCK**  0x00000002 | Hàm yêu cầu Khóa là duy nhất, đọc-ghi.  Nếu không set, hàm yêu cầu khóa chia sẻ (chỉ đọc). |
| **LOCKFILE\_FAIL\_IMMEDIATELY**  0x00000001 | Hàm trả về ngay lập tức nếu không lấy được khóa.  Nếu không set, hàm sẽ blocking, đợi cho tới khi lấy được khóa. |

* dwReserved: Để zero
* nNumberOfBytesToLockLow [in]: 2-byte thấp của vùng khóa
* nNumberOfBytesToLockHigh [in]: 2-byte cao của vùng khóa
* lpOverlapped [in, out]: Con trỏ cấu trúc OVERLAPPED, định vị trí byte bắt đầu của vùng khóa. Thành viên hEvent phải hợp lệ nếu là Async I/O, ngược lại là zero.

Hàm trả về TRUE nếu thành công ,ngược lại FALSE. GetLastError để bug lỗi.

LockFileEx cho phép ứng dụng dùng 2 loại khóa:

* LOCKFILE\_EXCLUSIVE\_LOCK: Khóa riêng sẽ từ chối tất cả process khác đọc và ghi vào vùng khóa của file. Chỉ process đang nắm khóa được đọc và ghi thôi.
* NON - LOCKFILE\_EXCLUSIVE\_LOCK: Khóa chia sẻ từ chối tất cả process khác ghi vào vùng khóa của file, bao gồm luôn cả process đang nắm khóa. Khóa này tạo ra vùng chỉ đọc trên file.

Khóa file không ảnh hưởng tới việc đọc từ mapped file.

1. Mở khóa UnlockFileEx

Phương thức UnlockFile hay UnlockFileEx để mở khóa file.

BOOL WINAPI UnlockFileEx(

\_In\_ HANDLE hFile,

\_Reserved\_ DWORD dwReserved,

\_In\_ DWORD nNumberOfBytesToUnlockLow,

\_In\_ DWORD nNumberOfBytesToUnlockHigh,

\_Inout\_ LPOVERLAPPED lpOverlapped

);

* hFile [in]: file Hanlde có quyền GENERIC\_READ hoặc GENERIC\_WRITE
* dwReserved: để zero
* nNumberOfBytesToUnlockLow [in]: 2-byte thấp của vùng khóa
* nNumberOfBytesToUnlockHigh [in]: 2-byte cao của vùng khóa
* lpOverlapped [in, out]: Con trỏ OVERLAPPED, định vị trí byte bắt đầu của vùng khóa. Thành viên hEvent phải hợp lệ nếu là Async I/O, ngược lại là zero.

Hàm trả về TRUE nếu thành công, ngược lại FALSE. GetLastError để bug lỗi.

Mỗi thao tác LockFileEx thành công, phải có UnLockFileEx. Nếu mở khóa thất bại hay khóa bị giữ lâu hơn cần thiết, process khác sẽ không thể truy xuất, gián tiếp làm giảm hiệu suất hệ thống.

Do đó, chương trình cần phải triển khai, thiết kế hợp lý.

1. Nguyên tắc vùng khóa

Một số điều cần lưu ý:

* Vùng mở khóa phải khớp hoàn toàn với vùng khóa trước đó, nếu không hàm sẽ trả về và hệ thống báo lỗi.
* 2 vùng liền kề của file không thể bị khóa riêng lẻ. Không thể kết hợp 2 vùng khóa hay mở khóa nhiều vùng cùng lúc.
* Vùng khóa này không được chồng lên vùng khóa trước.
* Được phép khai báo vùng khóa vượt ra ngoài kích thước file. Cách này dùng khi process hay thread đang mở rộng file.
* Khóa không được thừa kế cho process mới tạo ra.

1. Nguyên tắc khóa

* Khóa riêng không thể bị chồng hay đè lên các khóa khác.
* Khóa chia sẻ có thể bị chồng hay đè lên bởi các khóa chia sẻ khác.
* Khóa chia sẻ có thể đè lên một khóa riêng nếu cả 2 khóa cùng 1 file Handle: Trường hợp này, process chỉ được phép đọc.
* Nếu vùng khóa vừa chịu khóa riêng, vừa chịu khóa chi sẻ: phải có 2 thao tác mở khóa cho vùng này. Thứ tự là mở khóa riêng trước, mở khóa chia sẻ sau.
* Nếu process ngắt trước khi mở khóa: hệ điều hành tự động mở khóa. Tuy nhiên, thời gian mở khóa nhanh hay chậm lệ thuộc vào trạng thái tài nguyên. Do đó, ứng dụng nên làm đúng thủ tục mở khóa trước khi thoát. File sẽ ở trong tình trạng cách ly cho tới khị được OS unlock.

Bảng logic lock sau có thể tham khảo, thậm chí đối với process đã sở hữu khóa đang yêu cầu thêm khóa.

|  |  |  |
| --- | --- | --- |
| **Kiểu khóa hiện tại** | **Kiểu khóa yêu cầu mới** | |
| **Khóa chia sẻ** | **Khóa riêng** |
| Chưa có | Được phép | Được phép |
| Khóa chia sẻ (1 hay nhiều) | Được phép | Không được |
| Khóa riêng | Không được. Trừ khi cùng file Handle | Không được |

Khi process cố đọc hay ghi vào file đang bị khóa 1 hay nhiều vùng, các khóa sở hữu bởi nhiều process riêng biệt, ta có thể tham khảo bảng logic sau.

Tiến trình đọc hay ghi dừng lại khi nó đi tới vùng file bị tranh chấp:

|  |  |  |
| --- | --- | --- |
| **Kiểu khóa hiện tại** | **Tiến trình I/O** | |
| **Read** | **Write** |
| Chưa có | Thành công | Thành công |
| Khóa chia sẻ (1 hay nhiều) | Thành công | Thất bại |
| Khóa riêng | Thành công nếu process hiện hành sở hữu lock.  Thất bại nếu process hiện hành không sở hữu lock | Thành công nếu process hiện hành sở hữu lock.  Thất bại nếu process hiện hành không sở hữu lock |

Vùng tranh chấp khóa chỉ bị phát hiện vào lúc gọi hàm, kiểm tra giá trị trả về để biết đã lấy được khóa hay chưa.

1. Tình huống tranh chấp

Hãy điểm qua một vài tình huống như sau:

* Process A đang nắm khóa chia sẻ, process C bị block khi yêu cầu khóa riêng. Process B sau đó vẫn lấy được khóa chia sẻ một cách hợp lệ. Process C sẽ bị block tới khi tất cả process (A và B) mở khóa, dù B lấy khóa sau khi C bị block.  
  Trong trường hợp này, có khả năng C sẽ bị block vô hạn. Tương tự như deadlock trong tranh chấp mutex
* Process A đang nắm khóa chia sẻ, process B có ý định đọc file mà không yêu cầu khóa chia sẻ. Thao tác đọc của B vẫn diễn ra thành công trên vùng không tranh chấp, dù nó không có khóa.  
  Thao tác ghi hay đọc vẫn tiếp diễn tới khi gặp vùng tranh chấp. Khi WriteFile, ReadFile trả về, số byte thành công sẽ nhỏ hơn số byte yêu cầu.

1. Sử dụng Lock-Unlock File

Cú pháp command line: -concat desFile srcFile

Ứng dụng demo cách nối 1 file vào file khác, áp dụng cho 2 file text .txt.

desFile mở với chế độ FILE\_APPEND\_DATA, share mode là FILE\_SHARE\_READ: cho phép process khác đọc.

Tình huống là file cần nối vào có thể bị process khác truy xuất với quyền read-only. Chúng ta sẽ lock file trong quá trình ghi để đảm bảo dữ liệu toàn vẹn.

Dùng OVERLAPPED để định vị sau mỗi thao tác ghi, lock, unlock.

Trong quá trình ghi, dùng LockFileEx khóa vùng file sắp ghi. Mỗi khi ghi thành công, dùng UnlockFileEx để mở vùng đã ghi.

1. Trên Unix
2. Tìm kiếm

opendir, readdir và closedir trên UNIX có tính năng tương ứng với FindFirstFile, FindNextFile, FindClose trên Window.

Những hàm này sẽ lấy luôn thông tin file size, file time, thông tin bảo mật Unix như user sở hữu và user group…

Fstat và lstat có thể lấy đủ kiểu thông tin cần thiết.

Utime tương tự như SetFileTime.

1. Lock – Unlock File

UNIX has advisory file locking; an attempt to obtain a lock may fail (the logic is the same as in Table 3–1), but the process can still perform the I/O. Therefore, UNIX can achieve locking between cooperating processes, but any other process can violate the protocol.

To obtain an advisory lock, use options to the fcntl function. The commands (the second parameter) are F\_SETLK, F\_SETLKW (to wait), and F\_GETLK. An additional block data structure contains a lock type that is one of F\_RDLCK , F\_WRLCK,or F\_UNLCK and the range.

Mandatory locking is also available in some UNIX systems using a file’s set-group-ID and group-execute, both using chmod.

UNIX file locking behavior differs in many ways. For example, locks are inherited through an exe call.

The C library does not support locking, although Visual C++ does supply non standard locking extensions.

### Quản lý thư mục

Mỗi process đều có thư mục hiện hành gồm 2 dạng:

Đường dẫn nội bộ: C: hay C:\directory

Đường dẫn mạng: \\servername\sharename

Lưu ý không nên dùng đường dẫn nội bộ trong ứng dụng, thư viện đa luồng với SetCurrentDirectory(), GetCurrentDirectory() và GetFullPathName().

Thư mục hiện hành là một giá trị global của process, các thao tác thread có thể làm giá trị này thay đổi, khiến thread khác bị lỗi khi truy xuất.

1. Handle Directory

Để tạo thư mục, dùng CreateDirectory hay CreateDirectoryEx.

Chúng ta chỉ lấy được Handle của thư mục đã tồn tại qua CreateFile, với giá trị duy nhất hợp lệ dwCreationDisposition = OPEN\_EXISTING.

Phải chỉ rõ FILE\_FLAG\_BACKUP\_SEMANTICS cho dwFlagsAndAttributes: Handle có quyền ưu tiên SE\_BACKUP\_NAME và SE\_RESTORE\_NAME, không bị kiểm tra tính bảo mật hợp lệ.

Handle thư mục được dùng trong các API sau:

|  |  |
| --- | --- |
| **Function** | **Description** |
| [**BackupRead**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa362509(v=vs.85).aspx) | Sao lưu file, thư mục với thông tin bảo mật. |
| [**BackupSeek**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa362510(v=vs.85).aspx) | Seeks forward in a data stream initially accessed by using the [**BackupRead**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa362509(v=vs.85).aspx) or [**BackupWrite**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa362511(v=vs.85).aspx) function. |
| [**BackupWrite**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa362511(v=vs.85).aspx) | Phục hồi file, thư mục được sao lưu bởi [**BackupRead**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa362509(v=vs.85).aspx). |
| [**GetFileInformationByHandle**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa364952(v=vs.85).aspx) | Lây thông tin Handle file, thư mục. |
| [**GetFileSize**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa364955(v=vs.85).aspx) | Lấy kích thước logic của Handle, tính theo bytes. |
| [**GetFileTime**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms724320(v=vs.85).aspx) | Lấy thông tin FILE\_TIME của Handle. |
| [**GetFileType**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa364960(v=vs.85).aspx) | Xác định Handle là file, thư mục hay socket, pipe... |
| [**ReadDirectoryChangesW**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365465(v=vs.85).aspx) | Retrieves information that describes the changes within the specified directory. |
| [**SetFileTime**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms724933(v=vs.85).aspx) | Thiết lập thông tin FILE\_TIME cho handle. |

1. CreateDirectory

Nếu file system hỗ trợ bảo mật, hàm sẽ áp dụng thuộc tính bảo mật vào thư mục tạo ra.

BOOL WINAPI CreateDirectory(

\_In\_ LPCTSTR lpPathName,

\_In\_opt\_ LPSECURITY\_ATTRIBUTES lpSecurityAttributes

);

* lpPathName [in]: tên thư mục được tạo
* pSecurityAttributes [in, optional]: Nên để NULL cho thông số bảo mật mặc định. Đây là con trỏ cấu trúc SECURITY\_ATTRIBUTES.

Cũng giống như thao tác CreateFile(), hàm yêu cầu thông tin bảo mật cho thư mục. Thông số mặc định được thừa kế từ thư mục cha.

Hàm trả về true nếu thành công, ngược lại trả về false. GetLastError() có thể trả về:

**ERROR\_ALREADY\_EXISTS:** Lỗi tên thư mục đĩa tồn tại.

**ERROR\_PATH\_NOT\_FOUND:** Lỗi đường dẫn tới thư mục không tồn tại.

1. RemoveDirectory

Yêu cầu thư mục xóa phải trống.

BOOL WINAPI RemoveDirectory(

\_In\_ LPCTSTR lpPathName

);

* lpPathName [in]: đường dẫn thư mục xóa.

Hàm không quan tâm trạng thái của thư mục và sẽ vô hiệu mọi truy xuất tới thư mục.

Hàm trả về true nếu thành công, ngược lại trả về false

RemoveDirectory() chỉ đánh dấu thư mục này sẽ bị xóa khi mọi Handle liên quan được đóng.

Để xóa toàn bộ file trong thư mục, chúng ta có thể dùng SHFileOperation().

1. SetCurrenDirectory

Hàm SetCurrenDirectory() thay đổi thư mục cho process hiện tại.

BOOL WINAPI SetCurrentDirectory(

\_In\_ LPCTSTR lpPathName

);

* lpPathName [in]: Đường dẫn tới thư mục, phải có “\” ở cuối

Hàm trả về true nếu thành công, ngược lại trả về false.

Để truy xuất tới thư mục cha ta truyền tham số: “..”

1. GetCurrentDirectory

Hàm lấy thư mục hiện hành cho process đang thực thi ở dạng bộ đệm chuỗi lpBuffer

DWORD WINAPI GetCurrentDirectory(

\_In\_ DWORD nBufferLength,

\_Out\_ LPTSTR lpBuffer

);

* nBufferLength [in]: độ dài bộ đệm
* lpBuffer [out]: con trỏ tới bộ đệm TCHAR, chứa đường dẫn trả về. Hàm tự động chèn kí tự kết chuỗi ‘/0’ cuối chuỗi

Trường hợp độ dài bộ đệm không đủ, hàm cũng trả về độ dài cần thiết bao gồm ‘/0’

Hàm trả về độ dài kí tự nếu thành công, ngược lại trả về zero nếu bị lỗi. Dùng GetLastError() bug lỗi.

Như vậy chúng ta phải kiểm tra 2 điều kiện để đảm bảo hàm thành công:

* Hàm trả về khác zero.
* Số kí tự trả về <= độ dài bộ đệm

Cách triển khai an toàn là Set lpBUffer = NULL, nBufferLength = 0 , hàm sẽ điền độ dài cần thiết cho bạn.

Sau đó gọi hàm lần thứ 2 với bộ đệm đã khởi tạo với nBufferLength.

### Quản lý Volume

1. Volume

Volume thuộc tầng cao hơn file system, một ổ đĩa vật lý chứa ít nhất 1 partition (ổ logic).

Volume manager là lớp vật lý trừu tượng, thông qua driver điều khiển đĩa cứng, như Logical Disk Manager (LDM), và VERITAS Logical Volume Manager (LVM)

Volume manager còn có cơ chế bảo vệ dữ liệu giống như RAID

1. Tên Volume

Tên ký tự dùng cho tên ổ đĩa logic chỉ dành cho end-user, ký tự ổ đĩa không phân biệt hoa thường. Một volume có thể có nhãn, ký tự ổ đĩa, dùng SetVolumeLabel để set nhãn cho ổ đĩa.

Chúng ta không thể quản lý Volume theo ký tự ổ đĩa, hay nhãn vì chúng có thể thay đổi và không phải duy nhất.

Windows dùng đường dẫn GUID để nhận diện volume. Đường dẫn GUID là chuỗi có dạng:

"\\?\Volume{GUID}\"

Ví dụ: “\\?\Volume{4039899d-f63b-11d9-9648-806d6172696f}\”

Tuy nhiên một volume có thể có nhiều hơn 1 GUID, chỉ có GUID đầu tiên trả về khi tìm kiếm.

Tiền tố “\\?\”không phải là 1 phần của đường dẫn, chỉ để thông báo đây là đường dẫn mở rộng (32.767 ký tự), và Windows dùng Unicode API.

Phải chỉ rõ đường dẫn đầy đủ khi dùng đường dẫn GUID với tiền tố “\\?\”

Kích thước cho bộ đệm nhận GUID path hợp lý là 50 kí tự.

1. Liệt kê danh sách Volume

Để truy vấn danh sách volume hay volume ảo trên máy, chúng ta dùng các API sau:

* Gọi FindFirstVolume để tìm
* Dùng FindNextVolume trong vòng lặp để lấy từng volume, các volume trả về không theo thứ tự nào cả.
* Dùng FindVolumeClose để kết thúc

Ba phương thức trên hoạt động giống như FindFirstFile, FindNextFile, và FindClose.

Dùng GetVolumePathnamesForVolumeName để lấy Volume Path, ví dụ: “C:\”; “D:\”

1. FindFirstVolume

Dùng để dò Volume

HANDLE WINAPI FindFirstVolume(

\_Out\_ LPTSTR lpszVolumeName,

\_In\_ DWORD cchBufferLength

);

* lpszVolumeName [out]: Bộ đệm chuỗi có ‘\0’, nhận đường dẫn GUID của Volume đầu tiên tìm thấy.
* cchBufferLength [in]: Độ dài bộ đệm tính theo ký tự.

Hàm trả về INVALID\_HANDLE\_VALUE nếu thất bại. Dùng GetLastError để bug lỗi.

1. FindNextVolume

Trả về tên Volume tìm thấy sau mỗi lần gọi

BOOL WINAPI FindNextVolume(

\_In\_ HANDLE hFindVolume,

\_Out\_ LPTSTR lpszVolumeName,

\_In\_ DWORD cchBufferLength

);

* hFindVolume [in]: Handle volume tìm kiếm trả về từ FindFirstVolume
* lpszVolumeName [out]: Đường dẫn GUID trả về cho Volume tìm thấy
* cchBufferLength [in]: Kích thước bộ đệm tính theo số ký tự TCHAR.

Hàm trả về TRUE nếu tìm thấy. Trả về FALSE nếu thất bại, GetLastError = EROR\_NO\_MORE\_FILE nếu không còn volume để tìm.

1. FindVolumeClose

Đóng Handle Volume tìm kiếm

BOOL WINAPI FindVolumeClose(

\_In\_ HANDLE hFindVolume

);

* hFindVolume [in]: Handle volume tìm kiếm

Hàm trả về TRUE nếu thành công. FALSE nếu thất bại, dùng GetLastError để bug lỗi.

1. GetVolumePathNamesForVolumeName

Lấy danh sách driver path hay đường dẫn mounted folder đối với volume ảo.

BOOL WINAPI GetVolumePathNamesForVolumeName(

\_In\_ LPCTSTR lpszVolumeName,

\_Out\_ LPTSTR lpszVolumePathNames,

\_In\_ DWORD cchBufferLength,

\_Out\_ PDWORD lpcchReturnLength

);

* lpszVolumeName [in]: Đường dẫn GUID của volume
* lpszVolumePathNames [out]: Bộ đệm nhận danh sách driver path và đường dẫn mounted folder của volume. Danh sách có dạng nhiều chuỗi ‘\0’, kết thúc bởi ‘\0’. Có dạng “C:\”  
  Bộ đệm chứa nhiều dữ liệu nhất có thể nếu không đủ chứa.
* cchBufferLength [in]: Độ dài bộ đệm tính theo số ký tự TCHAR, bao gồm ký tự ‘\0’
* lpcchReturnLength [out]: trả về số ký tự ghi vào bộ đệm nếu thành công. Ngược lại, trả về kích thước bộ đệm phải đủ để chứa danh sách.

Hàm trả về TRUE nếu thành công, FALSE nếu thất bại. GetLastError để bug lỗi.

GetLastError trả về ERROR\_MORE\_DATA khi bộ đệm không đủ chứa.

Để xuất thông tin danh sách Volume path, dùng vòng lặp và con trỏ, ví dụ:

if (GetVolumePath(VolumeName, VolumePath, &VolumePathLen)){

TCHAR\* ptr = VolumePath;

//Xuat thong tin Volume path

while(\*ptr!=L'\0'){

wprintf(L"\nVolume Paths: %s\n", VolumePath);

ptr = ptr + lstrlen(ptr) + 1;

}

}

1. GetDriveType

Để nhận biết Volume là CD-ROM, RAM Disk, fixed, removable hay network drive

UINT WINAPI GetDriveType(

\_In\_opt\_ LPCTSTR lpRootPathName

);

* pRootPathName [in, optional]: thư mục gốc, phải có “\” ở cuối như “C:\”; “D:\”. Hoặc là Volume name (GUID path)

Nếu để Null, hàm sẽ dùng Volume hiện hành.

Một số giá trị trả về:

|  |  |
| --- | --- |
| **Return code/value** | **Description** |
| **DRIVE\_UNKNOWN**  0 | Không xác định được. |
| **DRIVE\_NO\_ROOT\_DIR**  1 | Đường dẫn không hợp lệ |
| **DRIVE\_REMOVABLE**  2 | Removable media; như floppy drive, thumb drive, hay flash card reader. |
| **DRIVE\_FIXED**  3 | Fixed media; như HDD hay flash drive. |
| **DRIVE\_REMOTE**  4 | Network drive. |
| **DRIVE\_CDROM**  5 | CD-ROM drive. |
| **DRIVE\_RAMDISK**  6 | RAM disk. |

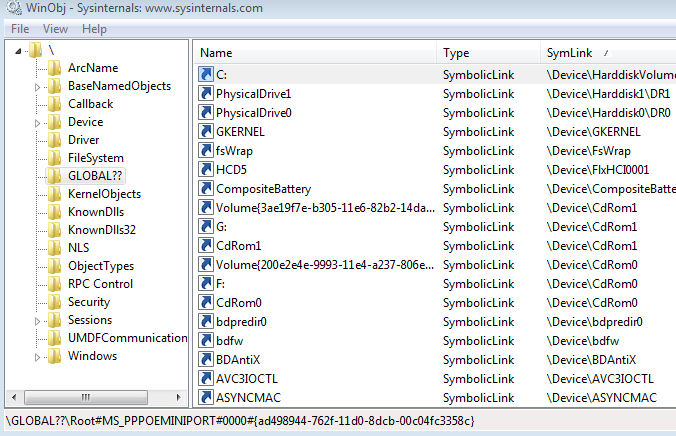
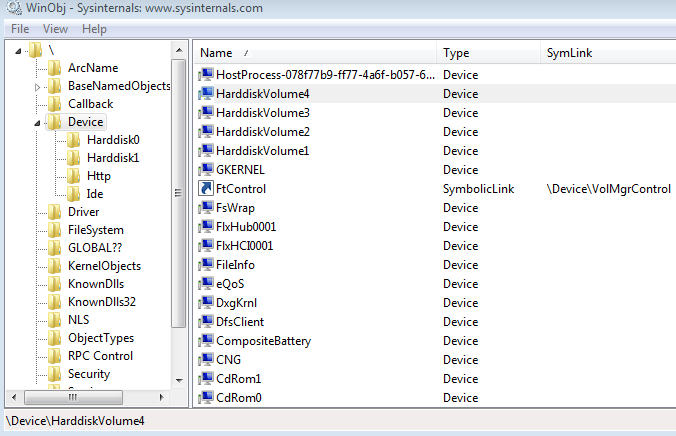
1. MS-DOS namespace
2. MS-DOS Device Name

MS-DOS device name là junction point trỏ tới đường dẫn MS-DOS device.

* Dùng DefineDosDevice để định nghĩa MS-DOS device name, nó sẽ nhìn thấy bởi tất cả process.
* DefineDosDevice và SetVolumeMountPoint để tạo MS-DOS device name và junction
* DeleteVolumeMountPoint xóa junction tạo bởi SetVolumeMountPoint

Khi user đăng nhập, session tương ứng được tạo ra kèm theo mã số xác thực gọi là là LUID (Locally unique identifier) cho user, LUID dùng để nhận diện MS-DOS devices.

Phạm vi truy xuất của device:

* MS-DOS device trong Global namespace cho phép tất cả user truy xuất.  
  
* MS-DOS device trong Local namespaces chỉ truy xuất bởi user có token security chứa đúng ID của MS-DOS device  
  

Có rất nhiều Local MS-DOS Device namespaces , nhưng chỉ có một Global MS-DOS Device namespace.

Chỉ có process local mới có thể dùng DefineDosDevice để tạo MS-DOS device trong Global MS-DOS device namespace.

Khi truy vấn MS-DOS device name với QueryDosDevice, phương thức sẽ tìm trong Local Ms-DOS device namespace, nếu tìm thấy nó sẽ trả về. Nếu chưa thấy, nó tìm trong Global MS-DOS Device namespace.

QueryDosDevice trả về kết quả là danh sách device name và :

* Nếu process chạy trên LocalSystem: chỉ có device trên Global MS-DOS Device namespace trả về.
* Nếu process không chạy trên LocalSystem: danh sách trên 2 namespace sau khi tổng hợp được trả về.
* Nếu device name có cả trên 2 namespace: chỉ trả về trên Local MS-DOS Device namespace

Luật áp dụng luôn cho danh sách trả về bởi GetLogicalDrives và GetLogicalDriveStrings.

Lưu ý: Process hay User không phải trên LocalSystem chỉ tạo được device name khi device name này không có trên Local và Global MS-DOS Device namespaces. Kiểm tra device name đã tồn tại chưa với QueryDosDevice trước khi gọi DefineDosDevice.

User không thuộc LocalSystem chỉ tạo được device name trong Local MS-DOS Device namespace.

Xét tình huống tạo device name:

User A, không thuộc LocalSystem, tạo device name trong Local MS-DOS Device namespace, device name chưa có trên Global MS-DOS Device namespace

User B, chạy trên LocalSystem, tạo cùng device name trong Global MS-DOS Device namespace.

Vậy sau đó, nếu User A muốn dùng device name trên Global MS-DOS Device namespace sẽ không được, vì QueryDosDevice luôn trả về Local device của User A.

Trừ khi User A xóa hay đổi tên device name trong Local MS-DOS Device namespace.

Do đó, MS-DOS drive letters device chỉ nằm trên Global MS-DOS Device namespace bắt đầu từ C: và kết thúc là Z:.

1. DefineDosDevice

Cho phép tạo, định nghĩa lại, xóa MS-DOS device name.

BOOL WINAPI DefineDosDevice(

\_In\_ DWORD dwFlags,

\_In\_ LPCTSTR lpDeviceName,

\_In\_opt\_ LPCTSTR lpTargetPath

);

* dwFlags [in]: Gồm một hay nhiều giá trị sau:

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| **DDD\_EXACT\_MATCH\_ON\_REMOVE**  0x00000004 | If this value is specified along with **DDD\_REMOVE\_DEFINITION**, the function will use an exact match to determine which mapping to remove. Use this value to ensure that you do not delete something that you did not define. |
| **DDD\_NO\_BROADCAST\_SYSTEM**  0x00000008 | Do not broadcast the [**WM\_SETTINGCHANGE**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms725497(v=vs.85).aspx) message. By default, this message is broadcast to notify the shell and applications of the change. |
| **DDD\_RAW\_TARGET\_PATH**  0x00000001 | Uses the lpTargetPath string as is. Otherwise, it is converted from an MS-DOS path to a path. |
| **DDD\_REMOVE\_DEFINITION**  0x00000002 | Removes the specified definition for the specified device. To determine which definition to remove, the function walks the list of mappings for the device, looking for a match of lpTargetPath against a prefix of each mapping associated with this device. The first mapping that matches is the one removed, and then the function returns.  If lpTargetPath is **NULL** or a pointer to a **NULL** string, the function will remove the first mapping associated with the device and pop the most recent one pushed. If there is nothing left to pop, the device name will be removed.  If this value is not specified, the string pointed to by the lpTargetPath parameter will become the new mapping for this device. |

* lpDeviceName [in]: Bộ đệm chứa tên device để tạo mới, định nghĩa lại hay xóa. Không được có (\) ở cuối, ví dụ device name của ổ C là “C:”
* lpTargetPath [in, optional]: Đường dẫn tới đối tượng MS-DOS device trỏ tới

A pointer to a path string that will implement this device. The string is an MS-DOS path string unless the DDD\_RAW\_TARGET\_PATH flag is specified, in which case this string is a path string.

Hàm trả về TRUE nếu thành công, FALSE nếu thất bại. GetLastError để bug lỗi.

Phương thức đơn giản là tùy chỉnh junction liên kết của device name tới MS-DOS device namespace

1. QueryDosDevice

Truy vấn thông tin MS-DOS device. Có thể lấy hết toàn bộ MS-DOS device.

DWORD WINAPI QueryDosDevice(

\_In\_opt\_ LPCTSTR lpDeviceName,

\_Out\_ LPTSTR lpTargetPath,

\_In\_ DWORD ucchMax

);

* lpDeviceName [in, optional]: là MS-DOS device name, không có (\) ở cuối, ví dụ: "C:", không phải "C:\".  
  Tham số có thể NULL, hàm sẽ query tất cả device vào bộ đệm.
* lpTargetPath [out]: bộ đệm chuỗi chứa kết quả trả về, là danh sách tên 1 hay nhiều nhiều thiết bị dưới dạng nhiều chuỗi có ‘\0’. Chuỗi kết thúc bởi ‘\0’.

Chuỗi đầu tiên trong danh sách trả về sẽ là device đang được ánh xạ tới, các chuỗi còn lại chỉ là các ánh xạ chưa bị xóa.

* ucchMax [in]: Số kí tự TCHAR tối đa mà bộ đệm chứa được.

Hàm trả về số kí tự trong bộ đệm nhận nếu thành công.

Hàm trả về zero hay FALSE nếu thất bại. GetLastError để bug lỗi.

Khi bộ đệm không đủ, hàm FALSE và GetLastError trả về ERROR\_INSUFFICIENT\_BUFFER.

1. Volume Information

The GetVolumeInformation function retrieves information about the file system on a given volume. This information includes the volume name, volume serial number, file system name, file system flags, maximum length of a file name, and so on. Before you access files and directories on a given volume, you should determine the capabilities of the file system by using the GetVolumeInformation function. This function returns values that you can use to adapt your application to work effectively with the file system.

In general, you should avoid using static buffers for file names and paths. Instead, use the values returned by GetVolumeInformation to allocate buffers as you need them. If you must use static buffers, reserve 256 characters for file names and 260 characters for paths.

The GetSystemDirectory and GetWindowsDirectory functions retrieve the paths to the system directory and the Windows directory, respectively.

The GetDiskFreeSpace function retrieves organizational information about a volume, including the number of bytes per sector, the number of sectors per cluster, the number of free clusters, and the total number of clusters. However, GetDiskFreeSpace cannot report volume sizes that are greater than 2 GB. To ensure that your application works with large capacity hard drives, use the GetDiskFreeSpaceEx function.

The GetLogicalDrives function identifies the volumes present. The GetLogicalDriveStrings function retrieves a null-terminated string for each volume present. Use these strings whenever a root directory is required.

1. GetLogicalDriveStrings

Lấy đường dẫn thư mục gốc các Volume dạng nhiều chuỗi ‘\0’, kết thúc bởi ‘\0’.

DWORD WINAPI GetLogicalDriveStrings(

\_In\_ DWORD nBufferLength,

\_Out\_ LPTSTR lpBuffer

);

* nBufferLength [in]: Kích thước tối đa của bộ đệm theo ký tự TCHAR.
* lpBuffer [out]: Bộ đệm nhận chuỗi trả về, ví dụ chuỗi gồm 2 “C:\\**\0**D:\\**\0\0**”

Để lpBuffer = NULL và nBufferLength = 0, hàm sẽ trả về kích thước cần thiết của bộ đệm.

Hàm luôn trả về số ký tự phải ghi vào bộ đệm dù thành công hay FALSE nếu thất bại. GetLastError để bug lỗi.

Kết quả trả về có thể dùng cho GetDriveType và GetDiskFreeSpace.

1. GetVolumePathName

Để lấy đường dẫn thư mục gốc từ file system path hay từ Volume mounted point

BOOL WINAPI GetVolumePathName(

\_In\_ LPCTSTR lpszFileName,

\_Out\_ LPTSTR lpszVolumePathName,

\_In\_ DWORD cchBufferLength

);

* lpszFileName [in]: Đường dẫn file tuyệt đối, UNC hay relative path.  
  Nếu để chuỗi rỗng, hàm FALSE và GetLastError trả về ERROR\_SUCCESS.
* lpszVolumePathName [out]: Bộ đệm nhận giá trị trả về, dạng “C:\”. Tối đa là MAX\_PATH + 1 cho kí tự Null
* cchBufferLength [in]: Độ dài ký tự TCHAR của bộ đệm.

Hàm trả về TRUE nếu thành công, FALSE nếu thất bại. GetLastError để bug lỗi.

Nếu đường dẫn là mounted folder path, hàm trả về Volume mount point, ví dụ :

Volume E mounted bởi "C:\Mnt\Ddrive\Mnt\Edrive".

Ta có file: "E:\Dir\Subdir\MyFile". Thì "C:\Mnt\Ddrive\Mnt\Edrive\Dir\Subdir\MyFile" tới GetVolumePathName, path trả về "C:\Mnt\Ddrive\Mnt\Edrive\".

* Nếu không ghi rõ thư mục gốc, hàm tự chọn boot Volume.
* Nếu file hay thư mục không tồn tại, hàm vẫn thành công và trả về Volume name.
* Nếu dùng Win32 namespace path, phải chính xác là “\\.\PhysicX”, ví dụ thiết bị đĩa cứng X.
* Không dùng NT namespace path. Nếu dùng "\DosDevices\H:" hay "\Device\HardDiskVolume6", hàm trả về kí tự ổ đĩa boot.

Nếu dùng local path, GetVolumePathName trả về full path whose prefix is the longest prefix that represents a volume.

Lưu ý khi dùng network path (remote) , GetVolumePathName trả về path ngắn nhất, những tình huống path không hợp lệ :

* Cú pháp sai.
* Path không tồn tại.
* User không có quyền truy xuất.
* Path là Junction Points và Mounted Folders

Nếu đường dẫn là junction point, hàm trả về Volume trỏ tới. Ví dụ: junct point “W:\Adir” trỏ tới “C:\Adir”, nếu truyền tham số “W:\Adir\Afile” hàm trả về “C:\”.

Nếu junction point trỏ tới nhiều điểm khác nhau, phương thức sẽ chọn liên kết cuối cùng.

Nếu remote path là mounted folder hay junction point, nó sẽ bị bỏ qua. Ví dụ: “C:\Dir\_C” trỏ tới “D:\Dir\_D” và “C:” ánh xạ tới “X:”trên máy remote, hàm sẽ

If a remote path to a mounted folder or junction point is specified, the path is parsed as a remote path, and the mounted folder or junction point are ignored. For example if C:\Dir\_C is linked to D:\Dir\_D and C: is mapped to X: on a remote computer, calling GetVolumePathName and specifying X:\Dir\_C on the remote computer returns X:\.

Ví dụ: U: ánh xạ tới remote computer \\*YourComputer*\C$, và Q là ổ logic.

|  |  |
| --- | --- |
| **Đường dẫn tham số** | **Giá trị trả về** |
| \\*YourComputer*\C$\Windows | \\*YourComputer*\C$\ |
| \\?\UNC\*YourComputer*\C$\Windows | \\?\UNC\*YourComputer*\C$\ |
| Q:\Windows | Q:\ |
| \\?\Q:\Windows | \\?\Q:\ |
| \\.\Q:\Windows | \\.\Q:\ |
| \\?\UNC\W:\Windows | **FALSE** with error 123 because a specified remote path was not valid; W$ share does not exist or no user access granted. |
| C:\COM2 (which exists) | \\.\COM2\ |
| C:\COM3 (non-existent) | **FALSE** with error 123 because a non-existent COM device was specified. |

Các trường hợp path không hợp lệ:

|  |  |
| --- | --- |
| **Specified path** | **Function returns** |
| G:\invalid (invalid path) | G:\ |
| \\.\I:\aaa\invalid (invalid path) | \\.\I:\ |
| \\*YourComputer*\C$\invalid (invalid trailing path element) | \\*YourComputer*\C$\ |

1. GetVolumeInformation

Lấy thông tin volume tương ứng với thư mục gốc

BOOL WINAPI GetVolumeInformation(

\_In\_opt\_ LPCTSTR lpRootPathName,

\_Out\_opt\_ LPTSTR lpVolumeNameBuffer,

\_In\_ DWORD nVolumeNameSize,

\_Out\_opt\_ LPDWORD lpVolumeSerialNumber,

\_Out\_opt\_ LPDWORD lpMaximumComponentLength,

\_Out\_opt\_ LPDWORD lpFileSystemFlags,

\_Out\_opt\_ LPTSTR lpFileSystemNameBuffer,

\_In\_ DWORD nFileSystemNameSize

);

lpRootPathName : Thư mục gốc có “\” ở cuối như C:\. Hoặc là Volume name (GUID path).

1. GetDiskFreeSpace

Lấy thông tin dung lượng trống của đĩa và tổng dung lượng được cấp, hay còn trống cho user hiện hành nếu có phân ngạch dung lượng.

BOOL WINAPI GetDiskFreeSpaceEx(

\_In\_opt\_ LPCTSTR lpDirectoryName,

\_Out\_opt\_ PULARGE\_INTEGER lpFreeBytesAvailable,

\_Out\_opt\_ PULARGE\_INTEGER lpTotalNumberOfBytes,

\_Out\_opt\_ PULARGE\_INTEGER lpTotalNumberOfFreeBytes

);

* lpDirectoryName [in, optional]: Tham số chấp nhận bất kỳ thư mục nào trên đĩa. Nếu để NULL, hàm sẽ dùng thư mục gốc hiện hành

Nếu là tên chuẩn UNC, phải có “\” ở cuối như "\\MyServer\MyShare\".

Ứng dụng phải có quyền FILE\_LIST\_DIRECTORY trên thư mục truy xuất

* lpFreeBytesAvailable [out, optional]: Nhận tổng số free byte trên đĩa của user hiện hành. Có thể NULL
* lpTotalNumberOfBytes [out, optional]: Nhận tổng số bytes trên đĩa cấp cho user hiện hành. Có thể NULL
* lpTotalNumberOfFreeBytes [out, optional]: Nhận tổng số free byte của đĩa.Có thể NULL.

Hàm trả về TRUE nếu thành công, FALSE nếu thất bại. Dùng GetLastError để bug lỗi.

Dùng IOCTL\_DISK\_GET\_LENGTH\_INFO với DeviceIoControl để biết tổng số byte của đĩa hay volume

Các tham số kiểu ULARGE\_INTEGER, không được rút xuống 32 bits.

GetDiskFreeSpaceEx trả về (0) cho lpTotalNumberOfFreeBytes và lpFreeBytesAvailable đối với ổ CD trừ khi là CD chưa ghi trong CD-RW drive.

Nếu tham số lpDirectoryName là Symbolic link, hàm sẽ tham chiếu tới đối tượng link vào.

### Đĩa cứng và Partition

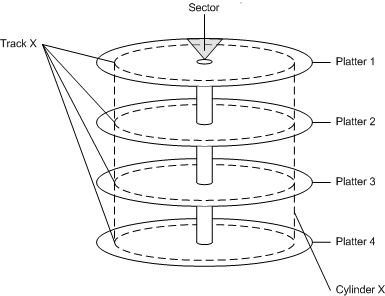
1. Physic Disk

Đĩa cứng bao gồm một bộ đĩa, mỗi dĩa có 2 đầu đọc/ghi ở mỗi bên.

Mỗi bên đĩa chia làm nhiều track, sector là đơn vị nhỏ nhất trên một track, kích thước 1 sector = 512-bytes.

Trong ví dụ bên dưới:

\_ Cylinder X gồm 8 track, 4 track mỗi bên nằm ở cùng một ví trên mỗi đĩa

Một đĩa cứng có nhiều vùng logic hay partition, được tạo ra khi định dạng basic disk.

Sector đầu tiên của basic disk chứa cấu trúc dữ liệu Master Boot Record (MBR) gồm:

* ứng dụng boot (lên tới 442 bytes)
* Mã số nhận dạng của đĩa (Số 4-byte)
* Bảng partition (tới 4 entries, tối đa 4 partition)
* Phần kết MBR (giá trị 0x55AA)

1. Basic Disk

Thuật ngữ basic và dynamic disk ám chỉ một dạng của cơ chế lưu trữ trên đĩa cứng.

Basic disk là dạng lưu trữ phổ biến trên Windows: basic disk có thể gồm nhiều partition, gồm primary paritions và các phân vùng logic.

Partition là Volume đã được định dạng với file system.

Basic disk hỗ trợ clustered disk (IEEE 1394) và ổ di động USB (universal serial bus). Basic disk tương thích ngược với MS-DOS khi dùng cùng kiểu MBR.

Basic disk không hỗ trợ GUID partition table.

Chúng ta có thể mở rộng, rút gọn primary partition và logic partition trên cùng đĩa cứng.

Để mở rộng Volume basic, nó phải được định dạng NTFS.

Các thao tác trên basic disk gồm:

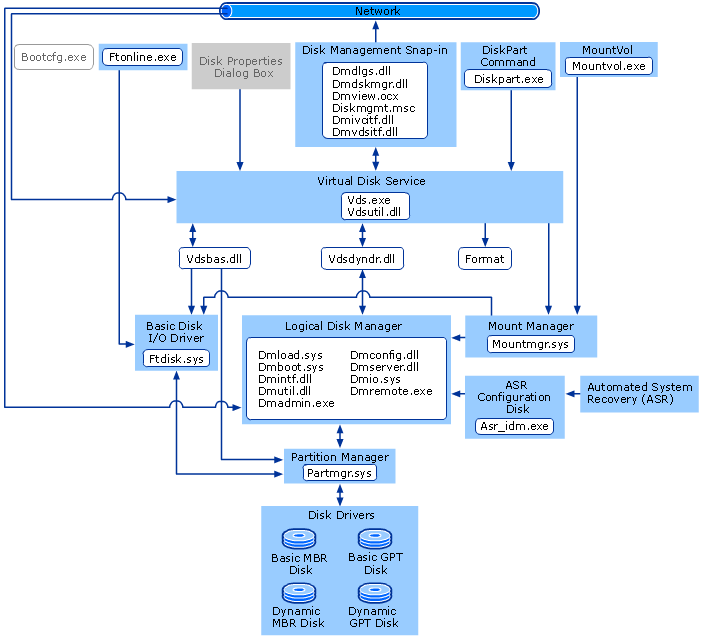
* Tạo và xóa primary hay partiton mở rộng
* Tạo và xóa logic drive trong partition mở rộng
* Định dạng partition và dánh dấu active

1. Kiến trúc Basic Disk

Basic disk và Volume lệ thuộc bởi Logical Disk Manager (LDM) và Virtual Disk Service (VDS)

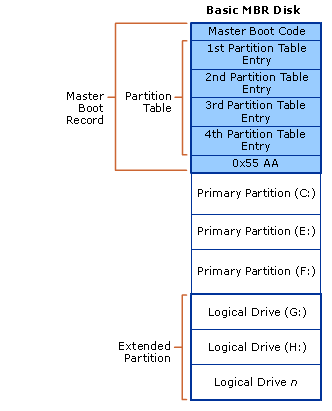
LDM và VSD cho phép thực hiện các tác vụ như chuyển basic thành dynamic disk, tạo fault-tolerant volumes

**Logical Disk Manager và Virtual Disk Service Components**



**Logical Disk Manager and Virtual Disk Service Components**

1. Basic MBR partition



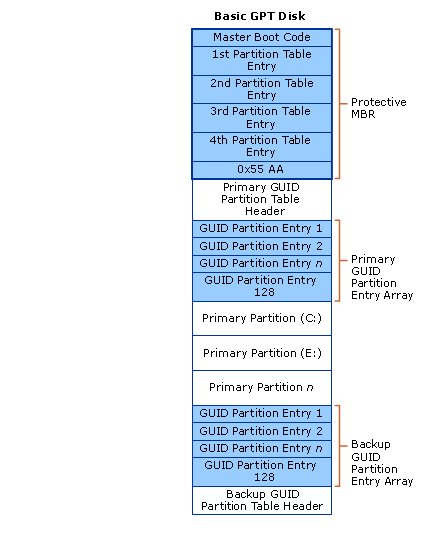
MBR partition yêu cầu basic disk chứa bảng MBR, gồm thông tin các partition.

Tối đa tạo được 4 partition trên một đĩa cứng dùng MBR: 4 primary partition hoặc 3 primary parition và 1 parition mở rộng.

Partition mở rộng có thể chứa nhiều ổ logic, và nằm ở đâu cũng được, không nhất thiết ở cuối.

Các partition nằm trên một dải liên tục, có thể định dạng với file system và chứa Windows.

1. Basic GUID partition



Ra đời từ Window Server 2003 SP1, với tên gọi globally uniquie identifier (GUID) partition table hay GPT. Basic disk dùng GPT partition có thể có tới 128 primary partitions, nên không cần phải tạo partition mở rộng với ổ logic

Dynamic disk chỉ có một LDM partition với MBR.

GPT partition có các đặc tính:

* Cho phép partition lớn hơn 2 terabytes
* Phục hồi và check lỗi CRC tốt hơn cho partition table
* Hỗ trợ GUID partition định nghĩa bởi OEMs, nhà phát triển độc lập và OS khác.

1. Kiến trúc Dynamic Disk

Dynamic disk hỗ trợ nhiều tính năng mới như chia spanned và striped volume, fault-tolerant volumes (mirrored và RAID-5 volume).

Dynamic disk có thể dùng MBR hay GPT partition. Tất cả Volume trên dynamic disk gọi là dynamic volume.

Dynamic disk linh hoạt hơn vì dùng database để quản lý thông tin các dynamic volume, dynamic disk khác.

Mỗi dynamic disk đều chứa bản sao các dynamic disk khác trong database, khi database 1 disk bị lỗi, có thể phục hồi từ disk khác.

Database của MBR partition chiếm 1-MB, của GPT chiếm 1-MB ẩn

Dynamic disk lệ thuộc vào Logical Disk Manager (LDM) và Virtual Disk Service (VDS) : các tính năng này cho phép thực hiện các thao tác chuyển basic disk sang dynamic va tạo fault-tolerant volumes.

Các thao tác trên dynamic disk gồm:

* Tạo, xóa trên simple, spanned, striped, mirrored và RAID-5 volumes.
* Mở rộng simple hay spanned volume
* Xóa một mirror từ mirrored volume và phân mirrored volume thành 2 volumes.
* Sửa mirrored hay RAID-5 volume
* Kích hoạt lại disk bị mất hay offline.

Volume trên dynamic disk có thể nằm rải rác trên nhiều vùng đĩa cứng, trong khi basic disk phải là một vùng liên tiếp.

Window không thể chuyển basic disk sang dynamic nếu không đủ 1 MB trống trên đĩa.

Có thể tạo tối đạ 2000 dynamic volumes trên hệ thống.

Các thao tác phổ biến trên basic và dynamic disk:

* Đều hỗ trợ MBR và GPT partition.
* Kiểm tra thông tin đĩa như dung lượng, vùng trống, các trạng thái…
* Xem thuộc tính partition như offset, kích thước, loại và boot partition hay không.
* Xem thuộc tính volume như kích thước, kí tự ổ địa, tên, kiểu, đường dẫn Win32, partition type và file system.
* Đặt tên kí tự ổ đĩa cho partition hay volume, và CD-ROM devices.
* Chuyển qua lại giữa basic và dynamic.

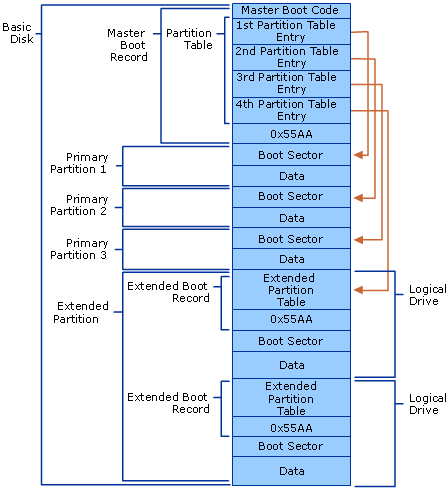
Windows khởi tạo partition trên đĩa cứng mặc định là basic disk. Bạn phải tự chuyển basic sang dynamic, tuy nhiên có điều kiện vùng trống tối thiểu khi chuyển sang dynamic.

1. Basic MBR

Khi đĩa cứng được định dạng, MBR sẽ được set trên Sector đầu tiên của basic disk gồm:

* Boot code : 442 bytes
* Mã số nhận dạng của đĩa tại offset 0x01B8: 4-bytes
* Bảng partition (tới 4 entries, tối đa 4 partition) : 64-bytes
* Dấu hiệu kết thúc MBR (giá trị 0x55AA) : 2-bytes

Window dùng mã số nhận dạng của đĩa để lưu trữ và truy xuất thông tin đĩa, như tên kí tự ổ đĩa trong registry.



Bên trên là MBR chi tiết gồm 3 primary partition và 1 partition mở rộng cùng EBR.

Mỗi primary partition đều có Volume Boot record (hay còn gọi là Boot sector) nằm trên sector đầu tiên của partition, chứa code cho các trình khởi động máy (như hệ điều hành).

Code trong MBR và VBR được nạp theo cùng một cách

1. Master boot code

Công việc của Master boot code:

1. Quét partition table cho active partition
2. Tìm sector bắt đầu của active partition
3. Nạp bản sao boot sector của active partition vào bộ nhớ
4. Chuyển quyền điều khiển thực thi cho boot sector.

Nếu một trong các thao tác thất bại, màn hình sẽ hiển thị:

Invalid partition table.

Error loading operating system.

Missing operating system.

**Lưu ý: Đĩa mềm và ổ di động không chứa MBR, sector đầu tiên chỉ chứa boot sector.**

**Tất cả đĩa cứng mặc định đều có MBR, nhưng master boot code chỉ thực thi khi đĩa cứng chứa active primary partition.**

1. Partition Table

Là cấu trúc 64-bytes để nhận diện kiểu và vị trí sector đầu tiên, kết thúc của partition trên đĩa cứng.

1 table entry dài 16-bytes, MBR tối đa 4 entry. Các Offset – thứ tự bytes của 4 partition:

* Partition 1   0x01BE   (446)
* Partition 2   0x01CE   (462)
* Partition 3   0x01DE   (478)
* Partition 4   0x01EE   (494)

Giao diện của hexa của một MBR có 3 partition, partition thứ 4 không có nên bằng zero.

000001B0:                                             80 01                 ..

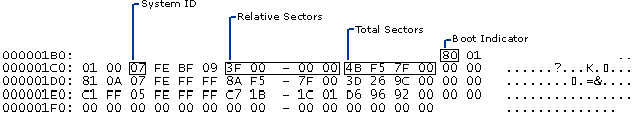
000001C0: 01 00 07 FE BF 09 3F 00 - 00 00 4B F5 7F 00 00 00   ......?...K....

000001D0: 81 0A 07 FE FF FF 8A F5 - 7F 00 3D 26 9C 00 00 00   .........=&....

000001E0: C1 FF 05 FE FF FF C7 1B - 1C 01 D6 96 92 00 00 00   ................

000001F0: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00         ..............

1. Entry partition table



Entry partition table đầu tiên gồm:

* Boot Indicator: 1-bytes, offset 0x01B0
* System ID: 1-bytes, offset 0x01C2
* Relative Sectors: 4-bytes, offset 0x01C6
* Total Sectors: 4-bytes, offset 0x01C7

Ta có thể tính nhanh cho các partition còn lại bằng công thức cộng thêm với PartitionNo \* 16dec hay (10Hex), với PartitionNo từ 0 tới 3.

1. **Cấu trúc entry table**

|  |  |  |  |
| --- | --- | --- | --- |
| **Byte Offset** | **Kích thước** | **Giá trị minh họa** | **Field Name and Definition** |
| 0x01BE | 1 byte | 80 | **Boot Indicator**: Cho biết có phải active partition hay không, có 2 giá trị hợp lệ:  00: không phải.  80. Active partition. |
| 0x01BF | 1 byte | 01 | **Starting Head**: đầu đọc bắt đầu |
| 0x01C0 | 6 bits | 01 \*2 | **Starting Sector**: Sector bắt đầu của partition, chỉ dùng bit thứ 0 – 5. |
| 0x01C1 | 10 bits | 00 \* | **Starting Cylinder**: Cylinder bắt đầu của partition. Giá trị tối đa là 1023 |
| 0x01C2 | 1 byte | 07 | **System ID: Cho biết loại file system trên Volume, như FAT16, FAT32, hay NTFS**  **System ID còn cho biết có phải partition mở rộng.** |
| 0x01C3 | 1 byte | FE | **Ending Head**: đầu đọc kết thúc |
| 0x01C4 | 6 bits | BF \* | **Ending Sector** Sector kết thúc của partition, chỉ dùng bit thứ 0 – 5.. |
| 0x01C5 | 10 bits | 09 \* | **Ending Cylinder**: Cylinder bắt đầu của partition. Giá trị tối đa là 1023 |
| 0x01C6 | 4 bytes | 3F 00 00 00 | **Relative Sectors**: Số sector tính từ offset đầu tiên của MBR (hay của đĩa cứng) tới offset bắt đầu của partition. (tính luôn Boot sector)  Trường này còn gọi là Logical block address (LBA) |
| 0x01CA | 4 bytes | 4B F5 7F 00 | **Total Sectors**. Tổng số sector của Volume. |

Các giá trị lưu ở dạng little endian,ví dụ 3F 00 00 00 tương ứng với giá trị số hex 0x0000003F, tức 63dec.

1. **Giá trị System ID**

|  |  |
| --- | --- |
| **ID Value** | **Partition Type** |
| 0x01 | FAT12 primary partition hay logical drive (tối đa 32,680 sectors cho 1 volume) |
| 0x04 | FAT16 partition hay logical drive (32,680–65,535 sectors hay 16 MB–33 MB) |
| 0x05 | Đây là partition mở rộng |
| 0x06 | BIGDOS FAT16 partition hay logical drive (33 MB–4 GB) |
| 0x07 | NTFS partition hay logical drive (256 terabyte) |
| 0x0B | FAT32 partition hay logical drive |
| 0x0C | FAT32 partition hay logical drive dùng BIOS INT 13h extensions |
| 0x0E | BIGDOS FAT16 partition hay logical drive dùng BIOS INT 13h extensions |
| 0x0F | Partition mở rộng dùng BIOS INT 13h extensions |
| 0x12 | EISA partition hay OEM partition |
| 0x42 | Dynamic volume |
| 0x84 | Power management hibernation partition |
| 0x86 | Multidisk FAT16 volume tạo bởi Windows NT 4.0 |
| 0x87 | Multidisk NTFS volume tạo bởi Windows NT 4.0 |
| 0xA0 | Laptop hibernation partition |
| 0xDE | Dell OEM partition |
| 0xFE | IBM OEM partition |
| 0xEE | GPT partition |
| 0xEF | EFI System partition trên MBR disk |

Windows NT 4.0 trở về trước dùng System ID: 0x86, 0x87, 0x8B, hay 0x8C.

MS-DOS chỉ có thể truy xuất Volume có System ID : 0x01, 0x04, 0x05, và 0x06.

1. CHS Fields

Starting - Ending Cylinder, Head và Sector fields (được biết với tên CHS fields) được Master boot code dùng để định vị active primary partition, nạp boot sector.

Thông số CHS của non-active partition sẽ trỏ tới boot sector của các partition này

Đối với partition mở rộng, CHS trỏ tới Extended boot record (EBR) nằm trên ổ logic đầu tiên của partition mở rộng. Trong ví dụ Entry table thì partition thứ 3 có System ID là 0x05, tức partition mở rộng.

Biết được sector bắt đầu của partition mở rộng rất quan trọng khi phân tích lỗi đĩa cấp thấp. Nếu ổ đĩa bị lỗi, phải bắt đầu phân tích từ vị trí bắt đầu để lấy dữ liệu.

* Trường Cylinder dài 10-bits giới hạn từ 0 tới 1023.
* Trường Head dài 1-bytes giới hạn từ 0 tới 255.
* Trường Sector dài 6-bits bắt đầu từ 1 tới 63, tối đa 1 track có 63 sector

Mặc định 1 sector là 512-bytes. Như vậy một partition có dung lượng tối đa là:

M = sector size \* cylinders (10 bits) \* heads (8 bits) \* sectors per track (6 bits).

Nếu dùng giá trị lớn nhất thì M = 512 \* 1024 \* 256 \* 63 (hay 512 x 224) = 8,455,716,864 bytes hay 7.8 gigabytes (GB).

Phiên bản Window hỗ trợ file system BIOS INT 13h extensions, có thể quản lý hơn 7.8 GB, bằng cách bỏ qua CHS, chỉ dùng thông số Relative Sectors và Total Sectors fields, tức tối đa 2 terabyte.

Các bản Windows 95, Windows 98, và Windows Millennium Edition (đều hỗ trợ BIOS INT 13h extensions) sẽ dùng CHS nếu partition không lớn hơn 7.8 GB.

MS-DOS và các bản Windows khác không hỗ trợ BIOS INT 13h extensions, OS sẽ không nhận được partition định dạng này với System ID 0x0C, 0x0E và 0x0F.

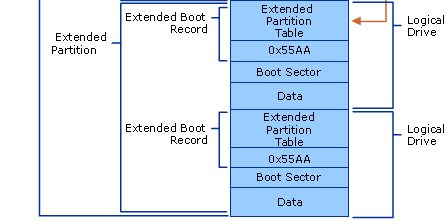
1. Relative Sectors và Total Sectors Fields

Bằng cách bỏ qua thông số CHS, Relative Sectors và Total Sectors hỗ trợ tối đa 232 sectors (32-bits).

Với sector = 512 bytes, partition có thể mở rộng tới 2 terabytes (2,199,023,255,552 bytes).

1. Basic EBR

EBR chứa thông tin về side đầu tiên của cylinder đầu tiên mỗi ổ logic trên partition mở rộng.

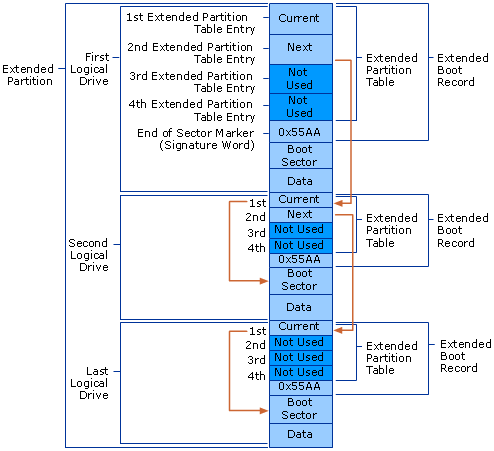


Boot sector của ổ logic thường nằm tại Relative Sector 32 hay 63

Partition mở rộng được tổ chức ở dạng danh sách liên kết EBR tương ứng với mỗi ổ logic. EBR gồm

* Entry trỏ tới boot sector của ổ logic
* Entry trỏ tới EBR tiếp theo: nếu không còn ổ logic nào, entry này là zero.
* 2 entry không dùng
* Entry đánh dấu kết thúc EBR: có giá trị 0x55AA

**Hình minh họa Partition mở rộng gồm 3 logic drive trên Basic Disk**



1. Entry partition table mở rộng

|  |  |
| --- | --- |
| **Entry** | **Contents** |
| 1 | Thông tin về ổ logic hiện tại, có chứa địa chỉ boot sector bắt đầu. |
| 2 | Thông tin về ổ logic tiếp theo, có chứa địa chỉ sector bắt đầu của EBR kế tiếp.  Nếu không có ổ logic tiếp theo, entry = zero. |
| 3 | Not used. |
| 4 | Not used. |

Cấu trúc các entry cũng tương tự như MBR partition table

Thông số Relative Sectors là số sector tính từ offset đầu tiên của parition mở rộng tới sector đầu tiên của ổ logic.

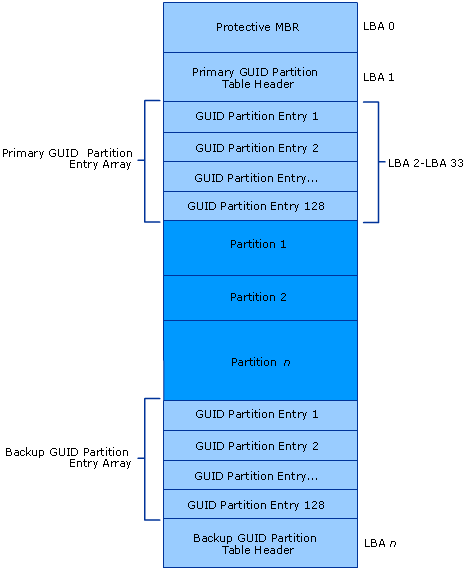
MBR và EBR sector rất quan trọng, nếu bị lỗi sẽ không thể truy xuất đĩa cứng.

1. Baisc GPT
2. Disk Sectors

GUID partition table (GPT) disks use primary and backup partition structures to provide redundancy. These structures are located at the beginning and the end of the disk. GPT identifies these structures by their logical block address (LBA) rather than by their relative sectors. Using this scheme, sectors on a disk are numbered from 0 to n-1, where n is the number of sectors on the disk.

As shown in the following figure, the first structure on a GPT disk is the Protective MBR in LBA 0, followed by the primary GPT header in LBA 1. The GPT header is followed by the primary GUID partition entry array, which includes a partition entry for each partition on the disk.

Partitions on the disk are located between the primary and backup GUID partition entry arrays. The partitions must be placed within the first usable and last usable LBAs, as specified in the GPT partition header.



1. Protective MBR

The Extensible Firmware Interface (EFI) specification requires that LBA 0 be reserved for compatibility code and a Protective MBR. The Protective MBR has the same format as an existing MBR, and it contains one partition entry with a System ID value of 0xEE. This entry reserves the entire space of the disk, including the space used by the GPT header, as a single partition. The Protective MBR is included to prevent disk utilities that were designed for MBR disks from interpreting the disk as having available space and overwriting GPT partitions. The Protective MBR is ignored by EFI; no MBR code is run.

Ví dụ về MBR:

000001B0: 00 00 00 00 00 00 00 00 - 04 06 04 06 00 00 00 00   ................

000001C0: 02 00 EE FF FF FF 01 00 - 00 00 FF FF FF FF 00 00   ................

000001D0: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00   ................

000001E0: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00   ................

000001F0: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 55 AA   ..............U.

Các trường trong protective MBR:

|  |  |  |  |
| --- | --- | --- | --- |
| **Byte Offset** | **Field Length** | **Sample Value1** | **Field Name and Definition** |
| 0x01BE | 1 byte | 00 | **Boot Indicator**. Must be set to 00 to indicate that this partition cannot be booted. |
| 0x01BF | 1 byte | 00 | **Starting Head**. Matches the Starting LBA of the single partition. |
| 0x01C0 | 1 byte | 02 | **Starting Sector**. Matches the Starting LBA of the single partition. |
| 0x01C1 | 1 byte | 00 | **Starting Cylinder**. Matches the Starting LBA of the GPT partition. |
| 0x01C2 | 1 byte | EE | **System ID**. Must be EE to specify that the single partition is a GPT partition. If you move a GPT disk to a computer running Windows 2000 with Service Pack 1 or greater or Windows Server 2003, the partition is displayed as a **GPT Protective Partition** and cannot be deleted. |
| 0x01C3 | 1 byte | FF | **Ending Head**. Matches the Ending LBA of the single partition. If the Ending LBA is too large to be represented here, this field is set to FF. |
| 0x01C4 | 1 byte | FF | **Ending Sector**. Matches the Ending LBA of the single partition. If the Ending LBA is too large to be represented here, this field is set to FF. |
| 0x01C5 | 1 byte | FF | **Ending Cylinder**. Matches the Ending LBA of the single partition. If the Ending LBA is too large to be represented here, this field is set to FF. |
| 0x01C6 | 4 bytes | 01 00 00 00 | **Starting LBA**. Always set to 1. The Starting LBA begins at the GPT partition table header, which is located at LBA 1. |
| 0x01CA | 4 bytes | FF FF FF FF | **Size in LBA**. The size of the single partition. Must be set to FF FF FF FF if this value is too large to be represented here. |

1. Numbers larger than one byte are stored in little endian format, or reverse-byte ordering. Little endian format is a method of storing a number so that the least significant byte appears first in the hexadecimal number notation.

1. GPT Partition Table Header

The GPT header defines the range of logical block addresses that are usable by partition entries. The GPT header also defines its location on the disk, its GUID, and a 32-bit cyclic redundancy check (CRC32) checksum that is used to verify the integrity of the GPT header.

GPT disks use a primary and a backup GUID partition table (GPT) header:

The primary GPT header is located at LBA 1, directly after the Protective MBR.

The backup GPT header is located in the last sector of the disk. No data follows the backup GPT header.

EFI verifies the integrity of the GPT headers by using a CRC32 checksum, which is a calculated value that is used to test data for the presence of errors. If the primary GPT header is corrupted, the system checks the backup GPT header checksum. If the backup checksum is valid, then the backup GPT header is used to restore the primary GPT header. This restoration process works in reverse if the primary GPT header is valid but the backup GPT header is corrupted. If both the primary and backup GPT headers are corrupted, then the 64-bit versions of Windows Server 2003 cannot access the disk.

**Note**

Do not use disk editing tools such as DiskProbe to make changes to GPT disks because any change that you make renders the checksums invalid, which might cause the disk to become inaccessible. To make changes to GPT disks, do either of the following:

Use Diskpart.efi in the firmware environment.

Use Diskpart.exe or Disk Management in the 64-bit versions of Windows Server 2003.

Ví dụ về GPT header:

00000000: 45 46 49 20 50 41 52 54 - 00 00 01 00 5C 00 00 00   EFI PART....\...

00000010: 27 6D 9F C9 00 00 00 00 - 01 00 00 00 00 00 00 00   'm..............

00000020: 37 C8 11 01 00 00 00 00 - 22 00 00 00 00 00 00 00   7.......".......

00000030: 17 C8 11 01 00 00 00 00 - 00 A2 DA 98 9F 79 C0 01   .............y..

00000040: A1 F4 04 62 2F D5 EC 6D - 02 00 00 00 00 00 00 00   ...b/..m........

00000050: 80 00 00 00 80 00 00 00 - 27 C3 F3 85 00 00 00 00   ........'.......

00000060: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00   ................

Các trường trong GPT header:

|  |  |  |  |
| --- | --- | --- | --- |
| **Byte Offset** | **Field Length** | **Sample Value1** | **Field Name and Definition** |
| 0x00 | 8 bytes | 45 46 49 20 50 41 52 54 | **Signature**. Used to identify all EFI-compatible GPT headers. The value must always be 45 46 49 20 50 41 52 54. |
| 0x08 | 4 bytes | 00 00 01 00 | **Revision**. The revision number of the EFI specification to which the GPT header complies. For version 1.0, the value is 00 00 01 00. |
| 0x0C | 4 bytes | 5C 00 00 00 | **Header Size**. The size, in bytes, of the GPT header. The size is always 5C 00 00 00 or 92 bytes. The remaining bytes in LBA 1 are reserved. |
| 0x10 | 4 bytes | 27 6D 9F C9 | **CRC32 Checksum**. Used to verify the integrity of the GPT header. The 32-bit cyclic redundancy check (CRC) algorithm is used to perform this calculation. |
| 0x14 | 4 bytes | 00 00 00 00 | **Reserved**. Must be 0. |
| 0x18 | 8 bytes | 01 00 00 00 00 00 00 00 | **Primary LBA**. The LBA that contains the primary GPT header. The value is always equal to LBA 1. |
| 0x20 | 8 bytes | 37 C8 11 01 00 00 00 00 | **Backup LBA**. The LBA address of the backup GPT header. This value is always equal to the last LBA on the disk. |
| 0x28 | 8 bytes | 22 00 00 00 00 00 00 00 | **First Usable LBA**. The first usable LBA that can be contained in a GUID partition entry. In other words, the first partition begins at this LBA. In the 64-bit versions of Windows Server 2003, this number is always LBA 34. |
| 0x30 | 8 bytes | 17 C8 11 01 00 00 00 00 | **Last Usable LBA**. The last usable LBA that can be contained in a GUID partition entry. |
| 0x38 | 16 bytes | 00 A2 DA 98 9F 79 C0 01 A1 F4 04 62 2F D5 EC 6D | **Disk GUID**. A unique number that identifies the partition table header and the disk itself. |
| 0x48 | 8 bytes | 02 00 00 00 00 00 00 00 | **Partition Entry LBA**. The starting LBA of the GUID partition entry array. This number is always LBA 2. |
| 0x50 | 4 bytes | 80 00 00 00 | **Number of Partition Entries**. The maximum number of partition entries that can be contained in the GUID partition entry array. In the 64-bit versions of Windows Server 2003, this number is equal to 128. |
| 0x54 | 4 bytes | 80 00 00 00 | **Size of Partition Entry**. The size, in bytes, of each partition entry in the GUID partition entry array. Each partition entry is 128 bytes. |
| 0x58 | 4 bytes | 27 C3 F3 85 | **Partition Entry Array CRC32**. Used to verify the integrity of the GUID partition entry array. The 32-bit CRC algorithm is used to perform this calculation. |
| 0x5C | 420 bytes |  | **Reserved**. Must be 0. |

1. Numbers larger than one byte are stored in little endian format, or reverse-byte ordering. Little endian format is a method of storing a number so that the least significant byte appears first in the hexadecimal number notation.

1. Mảng Entry GUID partition

Similar to the partition table on MBR disks, the GUID partition entry array contains partition entries that represent each partition on the disk. The 64-bit versions of Windows Server 2003 create an array that is 16,384 bytes, so the first usable block must start at an LBA greater than or equal to 34. (LBA 0 contains the protective MBR; LBA 1 contains the GPT header; and LBAs 2 through 33 are used by the GUID partition entry array.)

Each GPT disk contains two GUID partition entry arrays:

The primary GUID partition entry array is located after the GUID partition table header and ends before the first usable LBA.

The backup GUID partition entry array is located after the last usable LBA and ends before the backup GUID partition table header.

A CRC32 checksum of the GUID partition entry array is stored in the GPT header. When a new partition is added, this checksum is updated in the primary and backup GUID partition entries, and then the GPT header size checksum is updated.

1. Entry GUID partition

A GUID partition entry defines a single partition and is 128 bytes long. Because the 64-bit versions of Windows Server 2003 create a GUID partition entry array that has 16,384 bytes, you can have a maximum of 128 partitions on a basic GPT disk.

Each GUID partition entry begins with a partition type GUID. The 16-byte partition type GUID, which is similar to a System ID in the partition table of an MBR disk, identifies the type of data that the partition contains and identifies how the partition is used. The 64-bit versions of Windows Server 2003 recognize only the partition type GUIDs described in the following table, and do not mount any other type of partition. However, original equipment manufacturers (OEMs) and independent software vendors (ISVs), as well as other operating systems might define additional partition type GUIDs.

**Partition Type GUIDs**

|  |  |
| --- | --- |
| **Partition Type** | **GUID Value** |
| Unused entry | {00000000–0000–0000–0000–000000000000} |
| EFI System partition | {28732AC1–1FF8–D211–BA4B–00A0C93EC93B} |
| Microsoft Reserved partition | {16E3C9E3–5C0B–B84D–817D–F92DF00215AE} |
| Primary partition on a basic disk | {A2A0D0EB–E5B9–3344–87C0–68B6B72699C7} |
| LDM Metadata partition on a dynamic disk | {AAC80858–8F7E–E042–85D2–E1E90434CFB3} |
| LDM Data partition on a dynamic disk | {A0609BAF–3114–624F–BC68–3311714A69AD} |

The following example illustrates a partial hexadecimal printout of the GUID partition entry array on a basic GPT disk. This printout shows three partition entries: an EFI System partition, a Microsoft Reserved partition, and a primary partition. The partition type GUIDs are bold and match the entries in the previous table.

00000000: 28 73 2A C1 1F F8 D2 11 - BA 4B 00 A0 C9 3E C9 3B   (s\*......K...>.;

00000010: C0 94 77 FC 43 86 C0 01 - 92 E0 3C 77 2E 43 AC 40   ..w.C.....<w.C.@

00000020: 3F 00 00 00 00 00 00 00 - CC 2F 03 00 00 00 00 00   ?......../......

00000030: 00 00 00 00 00 00 00 00 - 45 00 46 00 49 00 20 00   ........E.F.I. .

00000040: 73 00 79 00 73 00 74 00 - 65 00 6D 00 20 00 70 00   s.y.s.t.e.m. .p.

00000050: 61 00 72 00 74 00 69 00 - 74 00 69 00 6F 00 6E 00   a.r.t.i.t.i.o.n.

00000060: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00   ................

00000070: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00   ................

00000080: 16 E3 C9 E3 5C 0B B8 4D - 81 7D F9 2D F0 02 15 AE   ....\..M.}.-....

00000090: 80 BC 80 FC 43 86 C0 01 - 50 7B 9E 5F 80 78 F5 31   ....C...P{.\_.x.1

000000A0: CD 2F 03 00 00 00 00 00 - D0 2A 04 00 00 00 00 00   ./.......\*......

000000B0: 00 00 00 00 00 00 00 00 - 4D 00 69 00 63 00 72 00   ........M.i.c.r.

000000C0: 6F 00 73 00 6F 00 66 00 - 74 00 20 00 72 00 65 00   o.s.o.f.t. .r.e.

000000D0: 73 00 65 00 72 00 76 00 - 65 00 64 00 20 00 70 00   s.e.r.v.e.d. .p.

000000E0: 61 00 72 00 74 00 69 00 - 74 00 69 00 6F 00 6E 00   a.r.t.i.t.i.o.n.

000000F0: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00   ................

00000100: A2 A0 D0 EB E5 B9 33 44 - 87 C0 68 B6 B7 26 99 C7   ......3D..h..&..

00000110: C0 1B 0B 00 44 86 C0 01 - F1 B3 12 71 4F 75 88 21   ....D......qOu.!

00000120: D1 2A 04 00 00 00 00 00 - 4E 2F 81 00 00 00 00 00   .\*......N/......

00000130: 00 00 00 00 00 00 00 00 - 42 00 61 00 73 00 69 00   ........B.a.s.i.

00000140: 63 00 20 00 64 00 61 00 - 74 00 61 00 20 00 70 00   c. .d.a.t.a. .p.

00000150: 61 00 72 00 74 00 69 00 - 74 00 69 00 6F 00 6E 00   a.r.t.i.t.i.o.n.

00000160: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00   ................

00000170: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00   ................

Các trường trong GUID partition Entry:

|  |  |  |  |
| --- | --- | --- | --- |
| **Byte Offset** | **Field Length** | **Sample Value1** | **Field Name and Definition** |
| 0x00 | 16 bytes | 28 73 2A C1 1F F8 D2 11 BA 4B 00 A0 C9 3E C9 3B | **Partition Type GUID**. Identifies the type of partition. The partition type GUID in this example identifies Microsoft Reserved partitions. For a description of partition type GUIDs, see the table titled Partition Type GUIDs earlier in this section. |
| 0x 10 | 16 bytes | C0 94 77 FC 43 86 C0 01 92 E0 3C 77 2E 43 AC 40 | **Unique Partition GUID**. A unique ID created for each partition entry. |
| 0x 20 | 8 bytes | 3F 00 00 00 00 00 00 00 | **Starting LBA**. The starting LBA of the partition that is defined by this partition entry. |
| 0x 28 | 8 bytes | CC 2F 03 00 00 00 00 00 | **Ending LBA**. The ending LBA of the partition that is defined by this partition entry. |
| 0x 30 | 8 bytes | 00 00 00 00 00 00 00 00 | **Attribute Bits**. Describe how the partition is used. For a description of the attribute used by the 64-bit versions of Windows Server 2003, see the table titled “GUID Partition Entry Attributes Used by the 64-Bit Editions of Windows on Itanium-based Computers” later in this section. |
| 0x 38 | 72 bytes | EFI system partition | **Partition Name**. A 36-character Unicode string that can be used to name the partition. |

1. Numbers larger than one byte are stored in little endian format, or reverse-byte ordering. Little endian format is a method of storing a number so that the least significant byte appears first in the hexadecimal number notation.

1. Thuộc tính của GUID partition Entry

GUID partition entry attributes are descriptors for how a partition is used. The attributes are specified within a 64-bit value, so EFI supports up to 64 different attributes. The 64-bit versions of Windows Server 2003 use two attributes as described in the following table.

**GUID Partition Entry Attributes Used by the 64-Bit Editions of Windows on Itanium-based Computers**

|  |  |
| --- | --- |
| **Bits** | **Description** |
| Bit 0 | Specifies that this partition is required for the platform to function. All original equipment manufacturer (OEM) partitions must have this bit set to protect the OEM partition from being overwritten by the disk tools supplied with Windows Server 2003. |
| Bit 60 | Marks the partition as read-only. Used only for primary basic partitions of type {EBD0A0A2-B9E5-4433-87C0-68B6B72699C7}. |
| Bit 62 | Marks the partition as hidden. Used only for primary basic partitions of type {EBD0A0A2-B9E5-4433-87C0-68B6B72699C7}. |
| Bit 63 | Prevents the system from assigning a default drive letter to the partition. Used only for primary basic partitions of type {EBD0A0A2-B9E5-4433-87C0-68B6B72699C7}. |

**Boot Sectors on GPT Disks**

Boot sectors on GPT disks are similar to boot sectors on MBR disks, except that EFI ignores all x86 code in the boot sector. Instead, EFI uses its own file system driver to read the BIOS parameter block (BPB) and then mount the volume.

**Basic Disks and Volumes Processes and Interactions**

The following basic disk and volume processes and interactions assume that your computer has at least one basic disk and that the basic disk is functioning properly.

**Creating a Basic Volume**

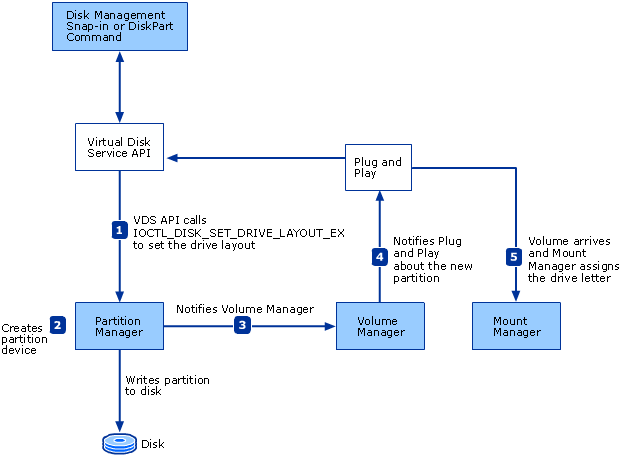
When you create a basic volume, the Virtual Disk Service (VDS) uses the IOCTL, IOCTL\_DISK\_SET\_DRIVE\_LAYOUT\_EX, to set the drive layout, and add a new partition.

VDS calls the disk driver. Partition manager (Partmgr.sys), sits on top of the disk driver as a filter). Partition manager creates the partition device and notifies the volume manager that there is a new partition. The volume manager announces the volume device to Plug and Play and to the system. Plug and Play notifies the mount point manager (Mountmgr.sys) that a new volume has arrived and the mount point manager sets up a drive letter as long as AutoMount is enabled.

**Note**

AutoMount is disabled by default on Windows Server 2003, Enterprise Edition, and Windows Server 2003, Datacenter Edition.

**What Happens When a Basic Volume Is Created**

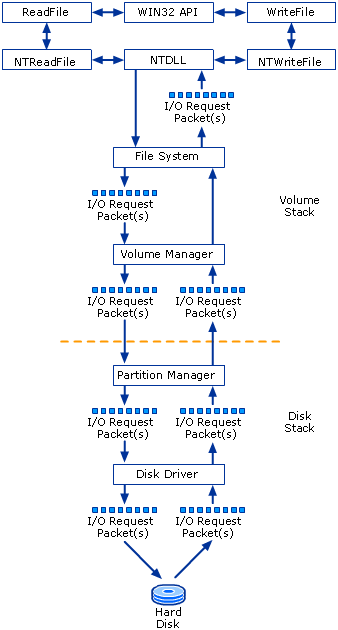


**Reading or Writing to a Basic Volume**

When you read or write data to basic disk or volume, the input/output (I/O) is sent through the file system to the volume manager. The volume manager creates a new I/O request packet (IRP) and sends it to the partition manager (Partmgr.sys) in the disk stack, which relays it to the disk driver. The volume manager waits for the completed IRP to return from the disk stack. Then, the volume manager completes and returns the file system IRP.

The following figure illustrates the processes involved when reading or writing to a basic volume.

**Basic Disk and Volume Input/Output (I/O) Processes**



**Converting a Basic Disk into a Dynamic Disk**

You can convert a basic disk to dynamic by using Disk Management or by using DiskPart, a command-line tool that provides the same functions as Disk Management.

When you convert a disk to dynamic, the following events occur:

All existing primary partitions and logical drives become simple volumes.

The disk joins the local disk group and receives a copy of the dynamic disk database.

**Note**

For certain disks, the menu command to convert the disk to dynamic is unavailable in Disk Management.

You can convert basic disks to dynamic at any time. In most cases, you do not need to restart your computer to complete the conversion. However, you must restart the computer if the disks you are converting contain any of the following volumes:

**System volume (x86-based computers only).** The system volume contains hardware-specific files such as Ntldr and Boot.ini. These files are needed to load Windows Server 2003 in x86-based computers.

**Boot volume.** The boot volume contains the Windows Server 2003 operating system and its support files. In x86-based computers, the boot volume can be, but does not have to be, the same volume as the system volume. In Itanium-based computers, the boot volume is never the same volume as the EFI System partition.

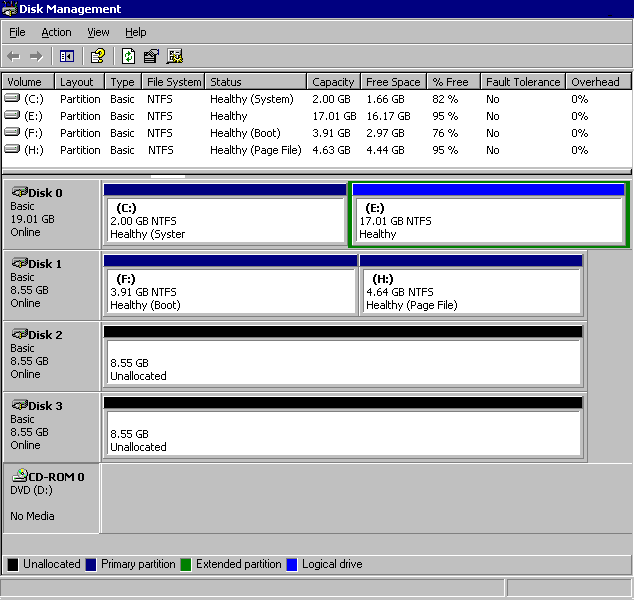
**Volumes that contain the paging file.** The paging file is a hidden file on the hard disk that Windows Server 2003 uses to hold parts of programs and data files that do not fit in memory.

**Note**

When you convert MBR disks that contain the system, boot, or paging file volumes to dynamic, you are prompted to restart the computer two times. You must restart the computer both times to complete the conversion.

As shown in the following figure the Disk Management snap-in identifies the system and boot volumes, as well as those that contain the paging file in the graphical and disk list views. If you have a combined system and boot volume that also contains the paging file (the most common scenario), then only **System** is shown.

**How Disk Management Identifies Separate System, Boot, and Paging File Volumes for an x86-based Computer**



The list volume command in DiskPart shows the system, boot, and paging file volumes as follows:

Volume ### Ltr Label Fs Type Size Status Info

---------- --- ----------- ----- ---------- --------- --------- --------

Volume 0 G DVD-ROM 0 B Healthy

Volume 1 C NTFS Partition 2048 MB Healthy System

Volume 2 E NTFS Partition 17418 MB Healthy

Volume 3 F NTFS Partition 4003 MB Healthy Boot

Volume 4 H NTFS Partition 751 MB Healthy Pagefile

Even after you convert a disk to dynamic, some types of primary partitions do not become dynamic volumes. These partitions retain their partition entries in the partition table and are shown as primary partitions in Disk Management. These partitions are:

Known OEM partitions (usually displayed in Disk Management as EISA Configuration partitions).

EFI System partitions on GPT disks.

**Before Converting Disks to Dynamic**

Converting a disk to dynamic changes the partition layout on the disk and creates the dynamic disk database. The result of these changes is increased flexibility for volume management in Windows Server 2003. However, these changes are not easily reversed, and the structure of dynamic disks is not compatible with some operating systems. Therefore, you must consider the following issues before you convert disks to dynamic.

**If the disk contains partitions displayed as Healthy (Unknown) in Disk Management**

Do not convert a disk to dynamic if it contains unknown partitions created by other operating systems. Windows Server 2003 converts unrecognized partitions to dynamic, making them unreadable to other operating systems.

**If a disk contains shadow copies**

If you use separate disks to store the source volume and the shadow copies, and you convert the disk that contains the source volume to dynamic, the shadow copies are lost. Shadow copies are retained only if the source files and the shadow copies are stored on the same volume.

**If the disk contains an OEM partition that is not at the beginning of the disk**

Do not convert a disk to dynamic if it contains an OEM partition that is not at the beginning of the disk. (In Disk Management, an OEM partition usually appears as an EISA Configuration partition.) When you convert a disk to dynamic, Windows Server 2003 preserves the OEM partition only if it is the first partition on the disk. Otherwise the partition is deleted during the conversion to dynamic.

**If you want to extend a dynamic volume**

You can extend dynamic volumes that do not retain their partition entries in the partition table. The following volumes retain their entries in the partition table and cannot be extended:

The system volume and boot volume of the operating system that you used to convert the disk to dynamic.

Any basic volume that was present on the disk when you converted the disk from basic to dynamic by using the version of Disk Management included with Windows 2000.

Simple volumes on which you run the DiskPart command **retain**. This command adds a partition entry to the partition table. However, after you use this command, you can no longer extend the volume.

**Note**

The **retain** command adds an entry to the partition table of an MBR disk only for simple volumes that are contiguous, start at cylinder-aligned offsets, and are an integral number of cylinders in size. If a volume does not meet these requirements, the **retain** command fails.

The following examples describe volumes on which the **retain** command will succeed:

The simple volume is contiguous and starts at the beginning of the disk.

The simple volume was present on the disk when the disk was converted to dynamic.

The only way to add more space to the system or boot volume on a dynamic disk is to back up all data on the disk, repartition and reformat the disk, reinstall Windows Server 2003, convert the disks to dynamic, and then restore the data from backup.

The following volumes do not have partition entries and can be extended:

Simple volumes and spanned volumes created from unallocated space on a dynamic disk.

A basic volume that meets the following criteria:

The basic volume is not the system or boot volume.

The basic volume is on a disk that was converted from basic disk to dynamic disk by using Windows Server 2003.

Although striped, mirrored, or RAID-5 volumes do not have entries in the partition table, Windows Server 2003 does not support extending them. You can add more space to a striped, mirrored, or RAID-5 volume by backing up the data, deleting the volume, recreating the volume by using Windows Server 2003, and then restoring the data.

**If you want to install Windows Server 2003 on a dynamic volume**

You can install Windows Server 2003 only on dynamic volumes that retain their partition entries in the partition table. The only dynamic volumes listed in the partition table are the following:

The system volume and boot volume of the operating system (Windows Server 2003 or Windows 2000) that you used to convert the disk to dynamic. The system volume and boot volume can be simple or mirrored volumes.

Any basic volume that was present on the disk when you used Windows 2000 to convert the disk from basic to dynamic.

Simple volumes on which you run the DiskPart command **retain**. This command adds a partition entry to the partition table so that you can install Windows Server 2003 on the simple volume.

A basic mirror set that was converted to a dynamic mirrored volume by using Windows 2000. If you break this mirrored volume into two simple volumes, you can also install Windows Server 2003 on either simple volume because they both retain their partition entries.

Because these dynamic volumes retain their partition entries, you can install Windows Server 2003 on them. However, you cannot extend any of these volumes because you can only extend volumes that do not have entries in the partition table.

**If you want to access the disk by using Windows Millennium Edition or earlier, or Windows NT 4.0**

If you plan to move the disk after you convert it to dynamic, note that you can access dynamic disks only from computers that are running Windows 2000, Windows XP Professional, Windows XP 64-Bit Edition, or Windows Server 2003. You cannot access dynamic disks from computers running Windows NT 4.0 or earlier.

When moving disks, note that access to dynamic disks is further restricted by the partition style used on the dynamic disk:

**Dynamic MBR disks**. Only computers running Windows 2000, Windows XP Professional, Windows XP 64-Bit Edition, or Windows Server 2003 can access dynamic MBR disks.

**Dynamic GPT disks**. Only Itanium-based computers running Windows XP 64-Bit Edition or the 64-bit versions of Windows Server 2003 can access dynamic GPT disks.

**Note**

Volumes on dynamic MBR and GPT disks are available across a network to computers running MS-DOS, Windows 95, Windows 98, Windows Millennium Edition, Windows NT 4.0 or earlier, Windows XP, and Windows Server 2003.

**If a disk or computer contains multiple copies of Windows XP Professional, Windows Server 2003 or Windows 2000**

Do not convert a disk to dynamic if it contains multiple copies of Windows XP Professional, Windows Server 2003, or Windows 2000. Even though these operating systems support dynamic disks, they require certain registry entries that allow them to start from dynamic disks. If the operating systems are installed on the same disk and you use one of the operating systems to convert the disk to dynamic, the registry of the other operating system becomes out-of-date because the drivers required to start the operating system from a dynamic disk are not loaded. Therefore, you can no longer start the other operating system.

One way that you can use dynamic disks with Windows XP Professional, Windows Server 2003, and Windows 2000 in a multiple-boot configuration is to install each operating system to a different disk. However, startup problems can also occur if you boot from one of the operating systems and then convert the disks that contain the other operating systems to dynamic.

To ensure that each operating system can start, start each operating system and then convert only the disk that contains the current boot volume to dynamic. For example, install Windows XP Professional on disk 1 and Windows Server 2003 on disk 2. Use Windows XP Professional to convert disk 1 to dynamic, and then use Windows Server 2003 to convert disk 2 to dynamic. By using this method, you ensure that the registries are updated for each boot volume.

**Disks That Cannot Be Converted to Dynamic**

Windows Server 2003 Setup and Disk Management ensure that disks initialized by Windows Server 2003 can be converted to dynamic. However, on some disks the conversion fails or the **Convert to Dynamic Disk** command is not available when you right-click a basic disk. The following conditions prevent you from converting a basic disk to dynamic.

**Cluster disks**

You cannot convert cluster disks to dynamic if they are connected to shared SCSI or Fibre Channel buses. Windows Cluster service cannot read disks that are dynamic and makes dynamic disks unavailable to programs or services that are dependent on these disk resources in the server cluster. For this reason, the option to convert these disks to dynamic is unavailable.

You must use Veritas Volume Manager to use dynamic disks with Cluster service.

**Removable disks**

You cannot use dynamic disks on the following:

Removable media, such as Iomega Zip or Jaz disks, CDs, DVDs, or floppy disks.

Disks that use universal serial bus (USB) or IEEE 1394 (also called FireWire) interfaces.

**Disks with sectors larger than 512 bytes**

A sector is a unit of storage on a hard disk. The majority of hard disks use 512-byte sectors. Windows Server 2003 supports converting basic disks to dynamic only if the sector size of the basic disk is 512 bytes.

**GPT disks with non-contiguous partitions**

If an unknown partition lies between two known partitions on a GPT disk, you cannot convert the disk to dynamic. Unknown partitions are created by operating systems or utilities that use partition type GUIDs that the 64-bit versions of Windows Server 2003 do not recognize.

**MBR disks that do not have space for the dynamic disk database**

An MBR disk requires 1 MB of free space at the end of the disk to be used for the dynamic disk database. Windows Server 2003 and Windows 2000 automatically reserve 1 MB or one cylinder, whichever is greater, when creating partitions on a disk, but in rare cases, disks with partitions created by other operating systems might not have this space available. If this space is not available, you cannot convert the disk to dynamic.

To convert the disk to dynamic, you must back up or move the data, delete the partitions, recreate the partitions, restore the data, and then convert the disk to dynamic. By using Windows Server 2003 to create the partitions, you ensure that the necessary space is available for the dynamic disk database.

This limitation does not affect GPT disks because the database is created in its own partition with space borrowed from the Microsoft Reserved partition.

1. Tiến trình Boot

1. The BIOS and CPU initiate the Power-on self-test (POST)

2. The BIOS searches for a boot device (HDD, Floppy, CD, etc.)

3. The BIOS reads the first sector of the device (Master Boot Record) into memory, and transfers CPU execution to that memory address.

Now that we have covered the boot process, lets look at what happens once the CPU transfers execution to the MBR.

Boot Code

The goal of the Master Boot Record's boot code is to find the bootable partition and transfer execution to it. To find the bootable partition the following checks must occur:

1. Search for an active partition (where the partition status is 0x80 or "Bootable")

2. Checks that no other partitions are marked as bootable

3. If partition is the only bootable partition, then the boot code will read the partition's Volume Boot Record (first sector, 512 bytes, of the partition) and transfer CPU execution to that memory address.

4. The VBR is checked for the proper end signature (0xAA55).

Errors

If steps 2 - 4 fail then an execution is halted and an error message is displayed. Below are the error messages for each step.

Step 2: If another partition is marked as bootable, the message "Invalid partition table" is displayed.

Step 3: If there is a problem loading the VBR, the message "Error loading operating system" is displayed.

Step 4: If the end signature is not present, the message "Missing operating system" is displayed.

Now that the we have covered how the boot code interacts with the partition table, lets talk about the structure of the partition table and its entries.

### File System layout

1. FAT-32 layout
2. Tổng quát

Hầu hết FAT file system phát triển bởi IBM, cấu trúc FAT trên đĩa lưu ở dạng little endian.

Nếu bạn đang dùng hệ máy big endian, phải chuyển sang big-endian khi đọc dữ liệu từ đĩa cứng.

FAT volume gồm 4 vùng cơ bản:

0 – Reserved Region

1 – FAT Region

2 – Root Directory Region (FAT32 volumes không dùng vùng này)

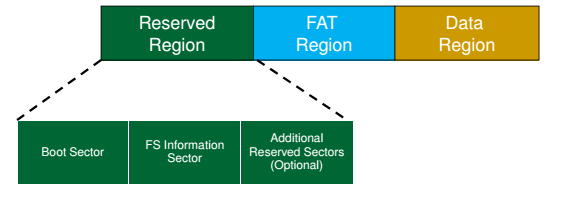
3 – File and Directory Data Region

FAT32 chỉ gồm 3 vùng chính:

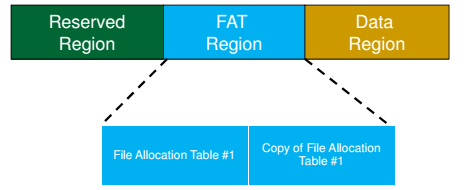


Vùng dành riêng sẽ bao gồm:

* Boot Sector: Sector đầu tiên của đĩa cứng, chứa tham số BIOS Parameter Block (BPB) và Boot Sector
* File System Information Sector
* Các Sector dành riêng



Vùng FAT là bản đồ dùng để duyệt dữ liệu gồm bảng FAT và bản sao của nó:



1. Boot Sector và BPB

BPB là khối tham số BIOS, cung cấp các thông tin như CHS, số sector…

Các phiên bản FAT trước FAT-32 chỉ hỗ trợ đĩa cứng dung lượng giới hạn 65.536 sector bởi trường “total sectors” chỉ dài 16-bits.

Kể từ Window 95, FAT-32 được mở rộng tới 32-bit hỗ trợ partition lớn hơn 2GB.

Các trường FAT32 BPB giống hệt với phiên bản FAT12/16, chỉ khác đi từ offset byte 36.

Lưu ý: các trường có tiền tố BPB\_ thuộc phạm vi của BPB. Tiền tố BS\_ thuộc phạm vi Boot sector.

Cấu trúc Boot Sector và BPB

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Offset (byte)** | **Size (bytes)** | **Description** |
| BS\_jmpBoot | 0x000 (0) | 3 | Chỉ thị jump tới boot code. Trường này có hai dạng hợp lệ:  **jmpBoot[0] = 0xEB, jmpBoot[1] = 0x??, jmpBoot[2] = 0x90**  và  **jmpBoot[0] = 0xE9, jmpBoot[1] = 0x??, jmpBoot[2] = 0x??**  **0x??** indicates that any 8-bit value is allowed in that byte. What this forms is a three-byte Intel x86 unconditional branch (jump) instruction that jumps to the start of the operating system bootstrap code. This code typically occupies the rest of sector 0 of the volume following the BPB and possibly other sectors. Either of these forms is acceptable. **JmpBoot[0] = 0xEB** is the more frequently used format. |
| BS\_OEMName | 0x003 (3) | 8 | Chỉ là chuỗi kí tự tên, FAT driver có thể dùng trường này.  Mặc định là “MSWIN4.1”, không nên thay đổi tránh bị lỗi. |
| BPB\_BytsPerSec | 0x00B (11) | 2 | Số byte trên một sector, có thể là: 512, 1024, 2048 hay 4096. Hầu hết các FAT file system hiểu mặc định 512 mà không cần kiểm tra trường này.  Microsoft OS có thể hỗ trợ 1024, 2048, và 4096.  **Note:** Do not misinterpret these statements about maximum compatibility. If the media being recorded has a physical sector size N, you must use N and this must still be less than or equal to 4096. Maximum compatibility is achieved by only using media with specific sector sizes. |
| BPB\_SecPerClus | 0x00D (13) | 1 | Số sector trên một đơn vị cấp phát (Cluster). Giá trị này phải lớn hơn 0 và là lũy thừa của 2, giá trị hợp lệ gồm: 1, 2, 4, 8, 16, 32, 64, và 128.  Giá trị này không hợp lệ nếu số bytes/cluster (BPB\_BytsPerSec \* BPB\_SecPerClus) lớn hơn 32KB.  Một số hệ thống cho phép 64KB/cluster. Nhiều ứng dụng không thể hoạt động trên FAT volume như vậy. |
| BPB\_RsvdSecCnt | 0x00E (14) | 2 | Số sector dành riêng trong vùng Reserved region  FAT12/16 mặc định là 1, mà không cần kiểm tra trường này.  FAT32 thường là 32 (0x20).  Microsoft OS không chấp nhận zero cho trường này. |
| BPB\_NumFATs | 0x010 (16) | 1 | Số bản sao FAT trên volume. Mặc định là 2 cho mọi phiên bản FAT.  Giá trị >= 1 là hợp lệ nhưng nhiều phần mềm và FAT driver chỉ làm việc với giá trị 2.  Tất cả driver của Microsoft file system hỗ trợ >=1.  Lý do 2 là giá trị mặc định vì nếu sector bị lỗi trên 1 FAT, dữ liệu sẽ không bị mất trên FAT kia.  Trên các thiết bị lưu trữ không phải dạng đĩa như FLASH memory cards, tính năng này bị bỏ đi, giá trị = 1 để tiết kiệm vùng nhớ, một số FAT file system driver sẽ không làm việc trên Volume này. |
| BPB\_RootEntCnt | 0x011 (17) | 2 | Số entry thư mục gốc.  FAT16 là 512  FAT32 là 0. |
| BPB\_TotSec16 | 0x013 (19) | 2 | Tổng số sector trên volume, bao gồm luôn 4 vùng cơ bản.  FAT12/16 volume dùng trường này.  FAT32 volumes phải để là 0.  Số sector nhỏ hơn 0x10000 nên dùng FAT16 |
| BPB\_Media | 0x015 (21) | 1 | Qui định loại thiết bị lưu trữ là cố định hay đi động.  0xF8 là cố định  0xF0 là đi động.  Các giá trị hợp lệ là : 0xF0, 0xF8, 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE, và 0xFF. The only other important point is that whatever value is put in here must also be put in the low byte of the FAT[0] entry. This dates back to the old MS-DOS 1.x media determination noted earlier and is no longer usually used for anything. |
| BPB\_FATSz16 | 0x016 (22) | 2 | Tổng số sector trên một FAT. FAT12/16 volume dùng trường này  FAT32 volumes không dùng, phải để là 0 |
| BPB\_SecPerTrk | 0x018 (24) | 2 | Sectors trên một track cho interrupt 0x13. This field is only relevant for media that have a geometry (volume is broken down into tracks by multiple heads and cylinders) and are visible on interrupt 0x13. |
| BPB\_NumHeads | 0x01A (26) | 2 | Tổng số Head cho interrupt 0x13. This field is relevant as discussed earlier for BPB\_SecPerTrk. This field contains the one based “count of heads”. Đĩa 1.44 MB 3.5-inch luôn là 2. |
| BPB\_HiddSec | 0x01C (28) | 4 | Tổng số sector ẩn trước FAT volume. This field is generally only relevant for media visible on interrupt 0x13. This field should always be zero on media that are not partitioned. Exactly what value is appropriate is operating system specific. |
| BPB\_TotSec32 | 0x020 (32) | 4 | FAT12/16 volume :Tổng số sector của volume, bao gồm luôn 4 vùng cơ bản.  FAT32 volumes: trường này phải khác 0. |

FAT16 dùng trường BPB\_TotSec16, FAT32 dùng trường BPB\_TotSec32.

Kể từ Offset byte 36, BPB/boot sector FAT12/16 sẽ khác với FAT32

Cấu trúc Fat12 / Fat16 tính từ Offset 36

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Offset (byte)** | **Size (bytes)** | **Description** |
| BS\_DrvNum | 0x024 (36) | 1 | Int 0x13 drive number (e.g. 0x80). Trường này dùng bởi MS-DOS bootstrap và được set is set to the INT 0x13 drive number của thiết bị (0x00 cho floppy disks, 0x80 cho hard disks).  **NOTE:** This field is actually operating system specific. |
| BS\_Reserved1 | 0x025 (37) | 1 | Dành riêng cho Windows NT. FAT volumes phải set 0. |
| BS\_BootSig | 0x026 (38) | 1 | boot signature mở rộng có giá trị 0x29, tức 3 trường phía sau có sử dụng cho boot sector. |
| BS\_VolID | 0x027 (39) | 4 | Số serial của volume. Cùng với trường BS\_VolLab, hỗ trợ giám sát volume trên thiết bị di động. Cho phép FAT driver phát hiện thiết bị non-removable drive  Mã số seri hay ID được sinh ra bằng cách kết hợp ngày tháng và thời gian hiện tại thành số 32-bit. |
| BS\_VolLab | 0x02B (43) | 11 | Tên nhãn của volume, khớp với 11-byte nhãn volume trong thư mục root.  **NOTE:** FAT file system drivers phải đảm bảo cập nhận trường này khi file nhãn volume trong thư mục root bị đổi hay được tạo.  Nếu Volume không có nhãn phải để là : “**NO NAME** ”. |
| BS\_FilSysType | 0x036 (54) | 8 | Chuỗi kí tự “**FAT12** ”, “**FAT16** ”, hay “**FAT** ”.  **NOTE:** Trường này không dùng để xác định file System trên Volume, thông tin có thể là sai hay đúng vì không thuộc phạm vi của BPB |

Cấu trúc FAT32 tính từ Offset 36

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Offset (byte)** | **Size (bytes)** | **Description** |
| BPB\_FATSz32 | 0x024 (36) | 4 | Tổng số sector của 1 FAT. |
| BPB\_ExtFlags | 0x028 (40) | 2 | Bits 0-3 : Số thứ tự của active FAT (based-zero) . Chỉ hợp lệ khi mirroring bị vô hiệu.  Bits 4-6 : Dành riêng chưa dùng.  Bit 7 \_ Nếu bằng 0 nghĩa là tính năng mirrored đang active cho nhiều FATs.  \_ Nếu bằng 1 nghĩa là chỉ có duy nhất 1 FAT là active.  Bits 8-15: Dành riêng chưa dùng . |
| BPB\_FSVer | 0x02A (42) | 2 | Chỉ số phiên bản của FAT32 volume. Giúp ứng dụng nhận dạng version và tương thích ngược cho phiên bản cũ hơn.   * High byte chỉ số phiên bản chính. * Low byte chỉ số phiên bản mở rộng.   This document defines the version to 0:0. If this field is non-zero, back-level Windows versions will not mount the volume.  **NOTE:** không nên làm việc trên FAT32 volume có version khác với version thiết kế cho ứng dụng. FAT32 file system drivers phải kiểm tra trường này và không mount volume nếu version không hỗ trợ trong driver. |
| BPB\_RootClus | 0x02C (44) | 4 | Số thứ tự cluster đầu tiên của thư mục root, thường là 2.  **NOTE:** ứng dụng xử lí đĩa nếu có thay đổi thư mục root, phải đặt cluster đầu tiên của thư mục root vào cluster non-bad của ổ đĩa, giúp trình xử lí tìm được thư mục root khi trường này vô tình bị zero. |
| BPB\_FSInfo | 0x030 (48) | 2 | Số thứ tự sector của cấu trúc FSINFO trong vùng dành riêng của FAT32 volume. Thường là 1.  **NOTE:** Luôn có một bản sao cấu trúc FSINFO trong BackupBoot, but only the copy pointed to by this field will be kept up to date (i.e., both the primary and backup boot record will point to the same FSINFO sector). |
| BPB\_BkBootSec | 0x032 (50) | 2 | Số thứ tự sector trong vùng dành riêng chứa bản sao boot record. Thường là sector thứ 6 và không được vượt qua 6.  Có thể là Zero |
| BPB\_Reserved | 0x034 (52) | 12 | Trường này dành riêng về sau. Phải để Zero toàn bộ. |
| BS\_DrvNum | 0x040 (64) | 1 | Int 0x13 drive number (e.g. 0x80). Trường này dùng bởi MS-DOS bootstrap và được set is set to the INT 0x13 drive number của thiết bị (0x00 cho floppy disks, 0x80 cho hard disks).  **NOTE:** This field is actually operating system specific. |
| BS\_Reserved1 | 0x041 (65) | 1 | Dành riêng cho Windows NT. FAT volumes phải set 0. |
| BS\_BootSig | 0x042 (66) | 1 | boot signature mở rộng có giá trị 0x29, tức 3 trường phía sau có sử dụng cho boot sector. |
| BS\_VolID | 0x043 (67) | 4 | Số serial của volume. Cùng với trường BS\_VolLab, hỗ trợ giám sát volume trên thiết bị di động. Cho phép FAT driver phát hiện thiết bị non-removable drive  Mã số seri hay ID được sinh ra bằng cách kết hợp ngày tháng và thời gian hiện tại thành số 32-bit. |
| BS\_VolLab | 0x047 (71) | 11 | Tên nhãn của volume, khớp với 11-byte nhãn volume trong thư mục root.  **NOTE:** FAT file system drivers phải đảm bảo cập nhận trường này khi file nhãn volume trong thư mục root bị đổi hay được tạo.  Nếu Volume không có nhãn phải để là : “**NO NAME** ”. |
| BS\_FilSysType | 0x052 (82) | 8 | Luôn là chuỗi kí tự ”**FAT32** ”.  **NOTE:** Trường này không dùng để xác định file System trên Volume, thông tin có thể là sai hay đúng vì không thuộc phạm vi của BPB |

**NOTE:** Giá trị 0x55AA signature luôn nằm ở offset 510, 511 của boot sector.

1. Các tham số cần tính

Được tổ chức dạng danh sách liên kết đơn các cluster của một file. Thư mục được xem như file có thuộc tính là thư mục, nội dung của nó chứa chuỗi các entry thư mục 32-byte FAT.

FAT ánh xạ vùng dữ liệu của volume theo số cluster, cluster dữ liệu đầu tiên luôn bắt đầu từ 2.

Kiến trúc partition entry dài 16-byte như hình dưới:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Boot flag | CHS begin | Type code | CHS end | LBA begin | Total sectors |

Chúng ta chỉ cần để ý đến trường Type Code là 0x0B hay 0x0C (dùng cho FAT32), và trường LBA

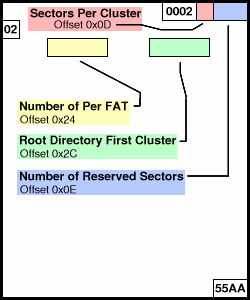
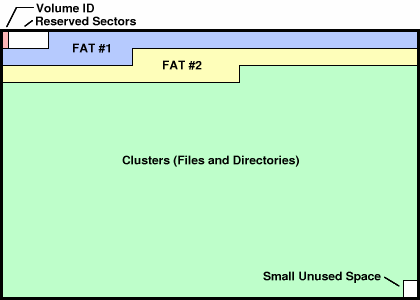
Các trường cần dùng trong Boot Sector:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Field** | **Microsoft's Name** | **Offset** | **Size** | **Value** |
| Bytes Per Sector | BPB\_BytsPerSec | 0x0B | 16-Bits | 512 Bytes |
| Sectors Per Cluster | BPB\_SecPerClus | 0x0D | 8-Bits | 1,2,4,8,16,32,64,128 |
| Number of Reserved Sectors | BPB\_RsvdSecCnt | 0x0E | 16-Bits | Thường là 0x20 |
| Number of FATs | BPB\_NumFATs | 0x10 | 8-Bits | Luôn là 2 |
| Sectors Per FAT | BPB\_FATSz32 | 0x24 | 32-Bits | Tùy vào kích thước đĩa |
| Root Directory First Cluster | BPB\_RootClus | 0x2C | 32 Bits | Thường là 0x00000002 |
| Signature | (none) | 0x1FE | 16 Bits | 0xAA55 |

* Tổng số Sector trên một FAT: **(DWORD) Sectors\_Per\_FAT = BPB\_FATSz32**
* Số Sector khoảng cách logic từ đầu đĩa tới FAT:  
  **(DWORD) Fat\_Begin\_LBA = Partition\_LBA\_Begin + Number\_of\_Reserved\_Sectors**

Số Sector khoảng cách logic từ đầu đĩa tới Cluster bắt đầu vùng data:  
**(DWORD) Cluster\_Begin\_LBA = Fat\_Begin\_LBA + (Number\_of\_FATs \* Sectors\_Per\_FAT)**Root\_Dir\_LBA luôn bằng Cluster\_Begin\_LBA.

* Tổng Sector của một FAT: **(DWORD) Sectors\_Per\_Cluster = BPB\_SecPerClus;**
* Cluster bắt đầu thư mục root : **(DWORD) Root\_Dir\_1st\_Cluster = BPB\_RootClus**



* Công thức tính số LBA Sector bắt đầu của cluster data # N:   
  **Cluster\_Data\_LBA = Cluster\_Begin\_LBA + (N - 2) \* Sectors\_Per\_Cluster**Phải lấy số cluster trừ 2 vì cluster data bắt đầu là #2
* Tổng số Sector vùng data:   
  **Data\_Sectors = BPB\_TotSec32** – **(Number\_of\_Reserved\_Sectors + (Number\_of\_FATs \* Sectors\_Per\_FAT))**

1. Chọn kiểu FAT định dạng

Để lựa chon kiểu FAT12, FAT16, hay FAT32—cho đĩa cứng , chúng ta phải đếm cluster data trên volume bằng công thức

**Data\_Clusters = Data\_Sectors / Sectors\_Per\_Cluster**

Lưu ý: kết quả phép toán đều phải làm tròn xuống.

* **Data\_Clusters** < 4085: FAT12
* **Data\_Clusters** < 65525: FAT16
* Ngoài ra là FAT32

**NOTE:** Chúng ta nên định dạng FAT volume tương thích tối đa cho tất cả phiên bản FAT đã có. Để ý tới các mốc 4085 hay 65525, để định dạng phù hợp.

1. Khởi tạo FAT Volume

Khi xử lí một volume chưa định dạng, Microsoft OS dùng giá trị cố định và vài thuật toán phức tạp để xác định và tính toán thông số cho trường BPB\_SecPerClus và BPB\_FATSz16 hay BPB\_FATSz32

* Nếu thiết bị > 4MB và < 512MB dùng FAT16 với BPB\_SecPerClus nhỏ
* Nếu thiết bị >= 512MB dùng FAT32

Đầu tiên định nghĩa cấu trúc bảng FAT DSKSZTOSECPERCLUS:

struct DSKSZTOSECPERCLUS {

DWORD DiskSize;

BYTE SecPerClusVal;

};

Tiếp theo chọn thông số Sectors\_Per\_Cluster tùy theo kích thước Volume:

/\*

\* This is the table for FAT32 drives. NOTE that this table includes

\* entries for disk sizes smaller than 512 MB even though typically

\* only the entries for disks >= 512 MB in size are used.

\* The way this table is accessed is to look for the first entry

\* in the table for which the disk size is less than or equal

\* to the DiskSize field in that table entry. For this table to

\* work properly BPB\_RsvdSecCnt must be 32, and BPB\_NumFATs

\* must be 2. Any of these values being different may require the first

\* table entries DiskSize value to be changed otherwise the cluster count

\* may be to low for FAT32.

\*/

DSKSZTOSECPERCLUS DskTableFAT32 [] = {

{ 66600, 0}, /\* thiết bị <= 32.5 MB, số cluster = 0 sẽ xảy ra lỗi \*/

{ 532480, 1}, /\* thiết bị <= 260 MB, .5k cluster \*/

{ 16777216, 8}, /\* thiết bị <= 8 GB, 4k cluster \*/

{ 33554432, 16}, /\* thiết bị <= 16 GB, 8k cluster \*/

{ 67108864, 32}, /\* thiết bị <= 32 GB, 16k cluster \*/

{ 0xFFFFFFFF, 64} /\* thiết bị > 32GB, 32k cluster \*/

};

Cuối cùng là tính số sector của bảng FAT entry cho trường BPB\_FATSz32.

Giả sử BPB\_RsvdSecCnt và BPB\_NumFATs đã cấu hình. Volume\_Size là kích thước của Volume set cho trường BPB\_TotSec32

DWORD Root\_Dir\_Sectors = 0;

DWORD TmpVal1 = Volume\_Size – (BPB\_ResvdSecCnt + Root\_Dir\_Sectors);

DWORD TmpVal2 = (256 \* BPB\_SecPerClus) + BPB\_NumFATs;

DWORD TmpVal2 = TmpVal2 / 2;

DWORD SectorPerFAT = (TMPVal1 + (TmpVal2 – 1)) / TmpVal2;

BPB\_FATSz16 = 0;

BPB\_FATSz32 = FATSz;

1. Cấu trúc FAT32 FSInfo Sector và Backup Boot Sector

"FS Information Sector" có từ FAT32 giúp tăng tốc độ truy xuất. Địa chỉ Sector logic qui định trong FAT32 EBPB tại Offset byte 0x030, thường là 1, tức nằm ngay sau Boot Sector của Volume

FAT32 FSInfo Sector Structure và Backup Boot Sector

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Offset (byte)** | **Size (bytes)** | **Description** |
| FSI\_LeadSig | 0 | 4 | Value 0x41615252. This lead signature is used to validate that this is in fact an FSInfo sector. |
| FSI\_Reserved1 | 4 | 480 | This field is currently reserved for future expansion. FAT32 format code should always initialize all bytes of this field to 0. Bytes in this field must currently never be used. |
| FSI\_StrucSig | 484 | 4 | Value 0x61417272. Another signature that is more localized in the sector to the location of the fields that are used. |
| FSI\_Free\_Count | 488 | 4 | Contains the last known free cluster count on the volume. If the value is 0xFFFFFFFF, then the free count is unknown and must be computed. Any other value can be used, but is not necessarily correct. It should be range checked at least to make sure it is <= volume cluster count. |
| FSI\_Nxt\_Free | 492 | 4 | This is a hint for the FAT driver. It indicates the cluster number at which the driver should start looking for free clusters. Because a FAT32 FAT is large, it can be rather time consuming if there are a lot of allocated clusters at the start of the FAT and the driver starts looking for a free cluster starting at cluster 2. Typically this value is set to the last cluster number that the driver allocated. If the value is 0xFFFFFFFF, then there is no hint and the driver should start looking at cluster 2. Any other value can be used, but should be checked first to make sure it is a valid cluster number for the volume. |
| FSI\_Reserved2 | 496 | 12 | This field is currently reserved for future expansion. FAT32 format code should always initialize all bytes of this field to 0. Bytes in this field must currently never be used. |
| FSI\_TrailSig | 508 | 4 | Value 0xAA550000. This trail signature is used to validate that this is in fact an FSInfo sector. Note that the high 2 bytes of this value—which go into the bytes at offsets 510 and 511—match the signature bytes used at the same offsets in sector 0. |

Another feature on FAT32 volumes that is not present on FAT16/FAT12 is the BPB\_BkBootSec field. FAT16/FAT12 volumes can be totally lost if the contents of sector 0 of the volume are overwritten or sector 0 goes bad and cannot be read. This is a “single point of failure” for FAT16 and FAT12 volumes. The BPB\_BkBootSec field reduces the severity of this problem for FAT32 volumes, because starting at that sector number on the volume—6—there is a backup copy of the boot sector information including the volume’s BPB.

In the case where the sector 0 information has been accidentally overwritten, all a disk repair utility has to do is restore the boot sector(s) from the backup copy. In the case where sector 0 goes bad, this allows the volume to be mounted so that the user can access data before replacing the disk.

This second case—sector 0 goes bad—is the reason why no value other than 6 should ever be placed in the BPB\_BkBootSec field. If sector 0 is unreadable, various operating systems are “hard wired” to check for backup boot sector(s) starting at sector 6 of the FAT32 volume. Note that starting at the BPB\_BkBootSec sector is a *complete* boot record. The Microsoft FAT32 “boot sector” is actually three 512-byte sectors long. There is a copy of all three of these sectors starting at the BPB\_BkBootSec sector. A copy of the FSInfo sector is also there, even though the BPB\_FSInfo field in this backup boot sector is set to the same value as is stored in the sector 0 BPB.

**NOTE:** All 3 of these sectors have the 0xAA55 signature in sector offsets 510 and 511, just like the first boot sector does (see the earlier discussion at the end of the BPB structure description).

1. FAT entry
2. Cluster chain

Directory entry chỉ cho biết cluster bắt đầu của file từ đâu, các cluster cấp phát cho file chưa chắc đã liền kề, để biết chính xác các cluster của file phải dùng bảng FAT để duyệt

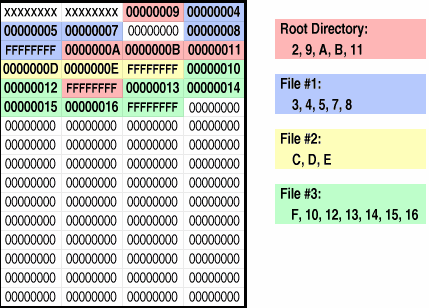
FAT32 ám chỉ entry FAT dài 32-bits, tương tự FAT12/16 là 12 và 16-bits, cách tổ chức bảng FAT đều giống nhau, tuy nhiên FAT12 sẽ phức tạp hơn vì các entry sẽ không khít trong một sector.

* Bảng File Allocation Table dùng như một mảng số nguyên 32-bit không dấu, FAT entry tổ chức theo dạng danh sách liên kết đơn.
* Mỗi entry chứa giá trị là vị trí FAT entry kế tiếp tương ứng với vị trí cluster kế tiếp của file. Lưu ý rằng Root nằm từ cluster thứ 2, nên 2 entry FAT đầu tiên không dùng.
* Mỗi Sector FAT chứa gọn 128 entry.
* 4-bit cao của FAT entry được dành riêng, không được dùng. Chỉ khi định dạng Volume, tất cả entry FAT phải set zero, bao gồm 4-bits cao  
  Ví dụ các entry cluster 32-bit : 0x10000000, 0xF0000000, and 0x00000000 chỉ ra các cluster đều FREE, bỏ qua 4-bit cao, 28-bits sau là zero.  
  Để lấy giá trị FAT entry, dùng AND với mask 0x0FFFFFFF.
* Bit 7-27 chỉ vị trí Sector FAT của entry kế tiếp. Dùng phép AND với mask 0x0FFFFF80, sau đó dịch phải 7-bits để lấy đúng Sector number.
* Bit 0-6 chỉ vị trí entry kế tiếp trong 128 entry của Sector. Dùng phép AND với mask 0x7F.
* Chuỗi entry đánh dấu kết thúc bởi giá trị lớn hơn hay bằng 0xFFFFFFF8, thực tế thì lại là 0xFFFFFFFF được ghi.
* Zero FAT entry báo cluster đang trống.

Bên dưới là ví dụ minh họa Root gồm 5 cluster tương tứng với 5 entry FAT, ổ đĩa hiện tại chỉ gồm 3 file với 3 màu khác nhau.

Entry root đầu tiên là “00000009h”, tức entry kế tiếp nằm trong Sector zero và có vị trí thứ 9.

Thư mục root chứa càng nhiều file thì số cluster sẽ được mở rộng ra, các entry sẽ không liên tiếp do các file sau này tạo thêm được cấp phát cluster mới



Vị trí entry thứ 11 đánh dấu cluster kết thúc của Root (0xFFFFFFFF), chúng ta phải duyệt cluster này tới khi gặp direcotry entry có byte đầu tiên zero.

File có độ dài zero byte sẽ không được cấp phát cluster nào.

File có độ dài khít trong một cluster thì FAT entry tại vị trí tương ứng chỉ đánh dấu là EOF (0xFFFFFFFF).

1. Bad Sector

Để dánh dấu bad sector cho FAT entry 0x30000000, chúng ta set giá trị 0x0FFFFFF7 cho nó. Entry sẽ có giá trị là 0x3FFFFFF7.

Số bytes trên một Sector luôn chia hết cho 2 và 4 nên FAT16/32 entry luôn nằm khít trong một sector, ngoại trừ FAT12.

If(FATContent >= 0x0FFFFFF8)

IsEOF = TRUE;

}

Note that the cluster number whose cluster entry in the FAT contains the EOC mark is allocated to the file and is also the last cluster allocated to the file. Microsoft operating system FAT drivers use the EOC value 0x0FFF for FAT12, 0xFFFF for FAT16, and 0x0FFFFFFF for FAT32 when they set the contents of a cluster to the EOC mark. There are various disk utilities for Microsoft operating systems that use a different value, however.

There is also a special “BAD CLUSTER” mark. Any cluster that contains the “BAD CLUSTER” value in its FAT entry is a cluster that should not be placed on the free list because it is prone to disk errors. The “BAD CLUSTER” value is 0x0FF7 for FAT12, 0xFFF7 for FAT16, and 0x0FFFFFF7 for FAT32. The other relevant note here is that these bad clusters are also lost clusters—clusters that appear to be allocated because they contain a non-zero value but which are not part of any files allocation chain. Disk repair utilities must recognize lost clusters that contain this special value as bad clusters and not change the content of the cluster entry.

NOTE: It is not possible for the bad cluster mark to be an allocatable cluster number on FAT12 and FAT16 volumes, but it is feasible for 0x0FFFFFF7 to be an allocatable cluster number on FAT32 volumes. To avoid possible confusion by disk utilities, no FAT32 volume should ever be configured such that 0x0FFFFFF7 is an allocatable cluster number.

1. Free cluster và reserved cluster

The list of free clusters in the FAT is nothing more than the list of all clusters that contain the value 0 in their FAT cluster entry. Note that this value must be fetched as described earlier as for any other FAT entry that is not free. This list of free clusters is not stored anywhere on the volume; it must be computed when the volume is mounted by scanning the FAT for entries that contain the value 0.

Trên FAT32 volumes, BPB\_FSInfo sector chứa thông số cluster free của volume

2 FAT Entry đầu tiên là FAT[0], FAT[1] không được sử dụng để lưu trữ thông tin. Khi forrmat ổ đĩa, tùy vào mỗi loại FAT mà 2 Entry này được gán giá trị khác nhau:

FAT12: FAT[0] = 0xF??; FAT[1] = 0xFFF;

FAT16: FAT[0] = 0xFF??; FAT[1] = 0xFFFF;

FAT32: FAT[0] = 0xFFFFFF??; FAT[1] = 0xFFFFFFFF;

“??” có giá trị bằng giá trị ở trường BPB\_Media. FAT[0] cũng không mang chức năng gì đặc biệt. Còn FAT[1], trong trường hợp FAT16/FAT32 thì chúng được sử dụng như sau:

Lịch sử trạng thái khi máy được tắt (FAT16: bit15、FAT32: bit31): 1:Bình thường, 0: Xuất hiện việc bị tắt bất thường.

Lịch sử lỗi phần cứng (FAT16: bit14、FAT32: bit30): 1: Không có lỗi、0: Có lỗi phần cứng.

Những bit trên để chỉ đến lỗi có thể phát sinh trên ổ đĩa. Nhưng OS support cái này sẽ dựa vào đó để biết được trạng thái của ổ đĩa luân lý lúc trước mà có thao tác tự động phụ hồi, kiểm tra lỗi thích hợp.

For FAT16:

ClnShutBitMask = 0x8000;

HrdErrBitMask = 0x4000;

For FAT32:

ClnShutBitMask = 0x08000000;

HrdErrBitMask = 0x04000000;

Bit ClnShutBitMask – If bit is 1, volume is “clean”.

If bit is 0, volume is “dirty”. This indicates that the file system driver did not Dismount the volume properly the last time it had the volume mounted. It would be a good idea to run a Chkdsk/Scandisk disk repair utility on it, because it may be damaged.

Bit HrdErrBitMask – If this bit is 1, no disk read/write errors were encountered.

If this bit is 0, the file system driver encountered a disk I/O error on the Volume the last time it was mounted, which is an indicator that some sectors may have gone bad on the volume. It would be a good idea to run a Chkdsk/Scandisk disk repair utility that does surface analysis on it to look for new bad sectors.

2 điều quan trọng về vùng FAT:

1. Sector cuối cùng của vùng FAT không nhất thiết thuộc về FAT. FAT dừng lại tại entry **Data\_Sectors + 1**, chúng ta không nên duyệt entry **Data\_Sectors + 1** về sau.

The last sector of the FAT is not necessarily all part of the FAT. The FAT stops at the cluster number in the last FAT sector that corresponds to the entry for cluster number CountofClusters + 1 (see the CountofClusters computation earlier), and this entry is not necessarily at the end of the last FAT sector. FAT code should not make any assumptions about what the contents of the last FAT sector are after the CountofClusters + 1 entry. FAT format code should zero the bytes after this entry though.

2. Giá trị BPB\_FATSz16 (BPB\_FATSz32 cho FAT32 volumes) có thể lớn hơn giá trị thực của nó, có thể có các sector FAT chưa dùng trong mỗi bản FAT. Vì thế, sector cuối cùng của bảng FAT luôn được tính toán dùng **Data\_Sectors + 1**, mà không dùng giá trị BPB\_FATSz16/32. FAT code không nên xác thực nội dung các sector extra FAT này và khi format phải zero nội dung các extra sector FAT này.

Liên quan đến vùng FAT có 2 điểm quan trọng.

Thứ nhất, không nhất thiết phải dùng đến hết đến Sector cuối cùng. Hầu hết FAT sử dụng một dãy Sector nhất định thôi.FAT Driver không được phép giả định về giá trị của phần chưa được sử dụng. Khi tạo ổ đĩa luân lý, phải khởi tạo vùng FAT về zero hết (Trừ FAT[0], FAT[1]).

Thêm một điều nữa, trường thể hiện kích thước của ổ đĩa (BPB\_FATSz16/32) nhiều khi cũng được gán giá trị lớn hơn thực tế. Vì thế, có thể có những Sector không được sử dụng nằm ở phía sau ổ đĩa FAT. Vì thế, để điều chỉnh lại các vùng này, người ta sinh ra các thao tác Aligment để điều chỉnh lại.

1. Giá trị của FAT entry

Dưới đây là bảng tóm tắt các khoảng giá trị của FAT Entry

|  |  |  |  |
| --- | --- | --- | --- |
| **FAT12** | **FAT16** | **FAT32** | **Mô tả** |
| 0x000 | 0x0000 | 0x00000000 | Cluster trống |
| 0x001 | 0x0001 | 0x00000001 | Dự trữ |
| 0x002:0xFF6 | 0x0002:0xFFF6 | 0x00000002:0x0FFFFFF6 | Chưa phải Cluster cuối, phía sau còn Cluster khác nữa trong day mô tả dữ liệu |
| 0xFF7 | 0xFFF7 | 0x0FFFFFF7 | Bad Sector |
| 0xFF8:0xFFF | 0xFFF8:0xFFFF | 0x0FFFFFF8:0x0FFFFFFF | Là Cluster cuối của dãy mô tả dữ liệu |

1. Directory entry:

Tất cả file, thư mục hay thư mục root đều có directory entry nằm trên cluster bắt đầu, directory entry chỉ cho biết làm sao tìm được cluster file và thư mục con.

Ngoài ra directory entry còn cho biết thuộc tính file. Để duyệt hết cluster của file, phải dùng FAT entry, nhưng trước hết cần biết file bắt đầu từ đâu.

Directory entry tổ chức theo dạng 32-byte record, như vậy mỗi sector chứa đủ 16 records và không có entry nào vượt qua ranh giới sector (FAT16/32), ngoại trừ FAT12.

1. Phân loại entry

Chúng ta có 4 kiểu entry:

1. **Normal entry với short filename** - Attrib là normal
2. **Long filename entry** - Attrib có 4-bits set ()
3. **Unused** – byte đầu tiên là 0xE5 : là kết quả của việc xóa file, byte đầu sẽ tạm thời ghi đè với 0xE5 nhưng dữ liệu chưa bị xóa hẳn. Khi có file mới tạo ra, các entry này được tái sử dụng và ghi đè dữ liệu
4. **End of directory** – byte đầu là zero : tại đầu cuối của directory entry sẽ là record bắt đầu là zero, ám chỉ không còn entry nào phía sau nữa, tất cả record phía sau có byte đầu đều là zero.

Record không bắt đầu là 0xE5 hay zero đều là direcotry entry hợp lệ và phải kiểm tra byte thuộc tính để xác định kiểu record.

1. Normal entry

Chúng ta sẽ tìm hiểu normal directory records có định dạng tên ngắn 8.3

Trong FAT32, tất cả file và thư mục đều có tên ngắn, chúng ta có thể truy xuất file mà không cần decode long name record.

Bên dưới là cấu trúc của normal directory record:

Short Directory Entry

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Field** | **Microsoft's Name** | **Offset** | **Size** | **Description** |
| Short Filename | DIR\_Name | 0x00 | 11 Bytes | Tên file theo chuẩn 8.3. 8-byte đầu là tên, 3-byte sau là phần mở rộng.  Nếu tên ngắn hơn 8 kí tự, byte trống phải là khoảng trắng (0x20) |
| Attrib Byte | DIR\_Attr | 0x0B | 8 Bits | Byte thuộc tính |
| First Cluster High | DIR\_FstClusHI | 0x14 | 16 Bits | 16-bits cao của thứ tự cluster bắt đầu |
| First Cluster Low | DIR\_FstClusLO | 0x1A | 16 Bits | 16-bits thấp của thứ tự custer bắt đầu |
| File Size | DIR\_FileSize | 0x1C | 32 Bits | Kích thước file. Nếu entry có thuộc tính là thư mục con hay nhãn volume thì file size là zero.  Long name record không để zero cho trường này |

Số thứ tự cluster bắt đầu dài 4-bytes, cho biết dữ liệu của file bắt đầu ở đâu và căn cứ vào File Size để xác định cluster kết thúc.

Byte thuộc tính gồm 6-bits định nghĩa. Hầu hết firmware phải kiểm tra byte thuộc tính để xác định 32-byte entry là normal record hay long file name record, là file hay thư mục con.

|  |  |  |  |
| --- | --- | --- | --- |
| **Attrib Bit** | **Function** | **LFN** | **Comment** |
| 0 (LSB) | Read Only | 1 | Chỉ đọc |
| 1 | Hidden | 1 | File ẩn, không hiển thị khi liệt kê thư mục , file. |
| 2 | System | 1 | File hệ thống |
| 3 | Volume ID | 1 | Tên file là nhãn Volume |
| 4 | Directory | x | Là thư mục con (32-byte records), cluster-chain của entry này duyệt theo kiểu thư mục thay vì kiểu file. Entry thư mục có filesize là zero. |
| 5 | Archive | x | Công cụ backup dùng bit này để xác định file được sao lưu trên volume đã bị thay đổi hay chưa |
| 6 | Ununsed | 0 | Để zero |
| 7 (MSB) | Ununsed | 0 | Để zero |

Long name record có bits 0-3 được set trong byte thuộc tính.

Entry nhãn Volume chỉ có duy nhất bit 3 được set và 2-bytes thấp, 2-bytes cao của cluster bắt đầu là zero. Entry Volume label luôn phải nằm trong thư mục root và là entry đầu tiên của cluster zero.

1. Long name entry

Long File Names (LFNs) record lưu trữ trên FAT dùng một mẹo – chèn vào trước normal record. Các entry LFNs được set thuộc tính Volume label, System, Hidden và Read-Only, giá trị là 0x0F.

Mặt nạ xác định thuộc tính long file name entry:

ATTR\_LONG\_NAME\_MASK ATTR\_READ\_ONLY **|** ATTR\_HIDDEN **|** ATTR\_SYSTEM **|** ATTR\_VOLUME\_ID (0x0F)

Môi trường MS-DOS và các ứng dụng thứ 3 nên bỏ qua các entry LFNs. Lý do theo sau chuỗi LFN entry phải có normal entry với tên 8.3 vì MS-DOS chỉ nhìn thấy tên 8.3, LFN không hợp lệ trong môi trường MS-DOS

Các LFNs entry chứa tối đa 13 kí tự (chuẩn Unicode hay 26-bytes) bằng cách tận dụng các trường File size, nhãn thời gian, nhưng 2 trường cluster bắt đầu phải set zero.

Long file name có kí tự NULL ở cuối (0x0000), các kí tự không dùng còn lại phải là 0xFFFF. Nếu long filename vừa khít (bộ số của 13 kí tự Unicode), thì sẽ không có kí tự NULL và không chèn 0xFFFF đằng sau.

**FAT Long Directory Entry Structure**

|  |  |  |
| --- | --- | --- |
| **Byte Offset** | **Length (bytes)** | **Description** |
| 0x00 | 1 | bit 6: Để xác định entry bắt đầu của chuỗi LFN, chứa phần cuối cùng của tên file. Dùng mặt nạ LAST\_LONG\_ENTRY (0x40)  bit 5: Zero  bits 0-4: số thứ tự của entry trong chuỗi LFN: 0x01 tới 0x14 (dùng mặt nạ 0x1F)  Deleted entry: 0xE5 |
| 0x01 | 10 | 5 kí tự tên |
| 0x0B | 1 | Byte thuộc tính ATTR\_LONG\_NAME (0x0F) |
| 0x0C | 1 | Kiểu LFN entry (0x00 cho LFN, các giá trị khác dành riêng về sau) |
| 0x0D | 1 | Checksum của tên file 8.3 tương ứng |
| 0x0E | 12 | 6 kí tự tên |
| 0x1A | 2 | Cluster bắt đầu (luôn để 0x0000) |
| 0x1C | 4 | 2 kí tự tên |

Chuỗi LFN luôn bắt đầu từ entry cuối cùng có số thứ tự lớn nhất, được nhận biết bằng bit 6 của byte đầu tiên được set (0x40), có số thứ tự là 1, giá trị byte đầu là 0xE5 tức file hay thư mục bị xóa

Ví dụ tên file "File with very long filename.ext" có chuỗi LFN:

|  |  |
| --- | --- |
| **Sequence number** | **Entry data** |
| 0x43 | "me.ext" |
| 0x02 | "y long filena" |
| 0x01 | "File with ver" |
| ??? | Normal 8.3 entry record |

Trường checksum để xác thực tên LFN khớp với tên 8.3, trường hợp không khớp xảy ra nếu file bị xóa và tạo lại trong môi trường DOS cho cùng vị trí thư mục.

Giải thuật tính checksum DOS filename, pFCBName là con trỏ tới tên 8.3:

//-----------------------------------------------------------------------------

// ChkSum()

// Returns an unsigned byte checksum computed on an unsigned byte

// array. The array must be 11 bytes long and is assumed to contain

// a name stored in the format of a MS-DOS directory entry.

// Passed: pFcbName Pointer to an unsigned byte array assumed to be

// 11 bytes long.

// Returns: Sum An 8-bit unsigned checksum of the array pointed

// to by pFcbName.

//------------------------------------------------------------------------------

unsigned char ChkSum (unsigned char \*pFcbName)

{

short FcbNameLen;

unsigned char Sum;

Sum = 0;

for (FcbNameLen=11; FcbNameLen!=0; FcbNameLen--) {

// NOTE: The operation is an unsigned char rotate right

Sum = ((Sum & 1) ? 0x80 : 0) + (Sum >> 1) + \*pFcbName++;

}

return (Sum);

}

Ví dụ về 2 LFN entry theo sau bằng 1 normal entry cho file name “The quick brown.fox”:



The heuristics used to "auto-generate" a short name from a long name are explained in a later section.

1. Tên 8.3 SFN

* Chỉ dùng kí tự hoa, kí tự thường không được chấp nhận trong tên file
* Giá trị 0x20 phải điền vào phần trống ngay sau của tên chính hay tên mở rộng. Ví dụ: "Readme.txt" sẽ là "README␠␠TXT".
* Các byte phải có giá trị nhỏ hơn 0x20, ngoại trừ 0x05 cho byte đầu tiên, đây là giá trị đặc biệt cho chuẩn tên Nhật.
* Các kí tự đặc biệt sau được phép dùng: “$ % ' - \_ @ ~ ` ! ( ) { } ^ # &” tương ứng với giá trị  
  0x22, 0x2A, 0x2B, 0x2C, 0x2E, 0x2F, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F, 0x5B, 0x5C, 0x5D, 0x7C.

Một vài ví dụ về cách đặt tên:

|  |  |
| --- | --- |
| **8.3 name** | **File Name** |
| "foo.bar" | "FOO BAR" |
| "FOO.BAR" | "FOO BAR" |
| "Foo.Bar" | "FOO BAR" |
| "foo" | "FOO " |
| "foo." | "FOO " |
| "PICKLE.A" | "PICKLE A " |
| "prettybg.big" | "PRETTYBGBIG" |
| ".big" | Không hợp lệ, tên file không thể là 0x20 |

Trong một thư mục FAT, tên file phải là duy nhất. 3 tên đầu tiên đều trỏ tới cùng một file, chỉ có thể duy nhất một file có tên “FOO BAR” trong bất kì thư mục nào.

Microsoft OS dùng 2 bit trên Offset byte 0x0C của normal entry qui định tên file là kí tự thường toàn bộ hay một phần.

* Bit 3: qui định tên chính là chữ thường
* Bit 4: qui định tên mở rộng là chữ thường

Ví dụ "example.TXT" hay "HELLO.txt", không phải dạng hỗ hợp "Mixed.txt".

Tổng số kí tự đường dẫn cho shortname không được vượt quá 80 (64 kí tự đường dẫn + 3 kí tự ổ đĩa + 11 kí tự tên 8.3 + 1 kí tự NULL)

1. Tên LFN

* Chuỗi LFN tối đa 20 entry, giới hạn 255 kí tự Unicode, không gồm kí tự NULL.
* Đường dẫn tối đa cho LFN dài tối đa 260 kí tự bao gồm kí tự NULL.
* Kí tự “.” được dùng nhiều lần trong tên.
* Khoảng trắng được xem là kí tự hợp lệ.
* 6 kí tự đặc biệt sau được phép dùng: + , ; = [ ]
* Các khoảng trắng phía trước và sau LFN sẽ bị bỏ đi
* Các kí tự “.” Cuối LFN sẽ bị bỏ đi

1. Quy ước tên file

Tên 8.3 và LFN cùng với nhau tạo thành đặc điểm nhận dạng cho file trong thư mục.

* Tên 8.3 hay LFN chỉ được duy nhất trong cùng một thư mục. Nếu bạn có LFN “FooBar” với tên 8.3 “FOOBAR”, sẽ không thể tạo LFN “foobar” vì sẽ trùng tên 8.3.
* Các API cho phép tùy chỉnh LFN nhưng không cho tác động riêng lẻ lên tên 8.3, đó là do tên 8.3 được sinh ra từ LFN,.
* Window NT dùng giải thuật tạo ra tên 8.3 từ LFN sao cho không xung đột với các tên 8.3 trong cùng thư mục, giải thuật tạo tên không phân biệt hoa thường trên LFN.
* Giải thuật chia làm 2 phần cho *basis-name(tên cơ sở)* và *numeric-tail (số đuôi)*.

Tất cả hành vi trên file system không xét hoa thường (tìm kiếm, mở, xóa, đổi tên).

Thao tác trên tên file 8.3 chỉ kiểm tra tên trong normal entry, thao tác trên tên LFN sẽ kiểm tra trên cả LFN và normal entry.

1. Giải thuật Basic-name

Giải thuật triển khai cơ bản theo các bước sau:

1. The UNICODE name passed to the file system is converted to upper case.

2. The upper cased UNICODE name is converted to OEM.  
if (the uppercased UNICODE glyph does not exist as an OEM glyph in the OEM code page)  
 or (the OEM glyph is invalid in an 8.3 name)  
{  
 Replace the glyph to an OEM '\_' (underscore) character.  
 Set a "lossy conversion" flag.  
}

3. Strip all leading and embedded spaces from the long name.

4. Strip all leading periods from the long name.

5. While (not at end of the long name)  
 and (char is not a period)  
 and (total chars copied < 8)  
{  
 Copy characters into primary portion of the basis name  
}

6. Insert a dot at the end of the primary components of the *basis-name* iff the basis name has an extension after the last period in the name.

7. Scan for the last embedded period in the long name.  
If (the last embedded period was found)  
{  
 While (not at end of the long name)  
 and (total chars copied < 3)  
 {  
 Copy characters into extension portion of the basis name  
 }  
}

1. Giải thuật number-tail

If (a "lossy conversion" was not flagged)  
 and (the long name fits within the 8.3 naming conventions)  
 and (the basis-name does not collide with any existing short name)  
{  
 The short name is only the basis-name without the numeric tail.  
}  
else  
{  
 Insert a numeric-tail "~n" to the end of the primary name such that the value of the "~n" is chosen so that the   
 name thus formed does not collide with any existing short name and that the primary name does not exceed eight characters in length.  
}

Chuỗi "~n" từ "~1" tới "~999999". Số "n" tăng dần cho các file có basic name giống nhau.

Ví dụ: thư mục đã có LETTER~1.DOC và LETTER~2.DOC, tiếp theo sẽ là LETTER~3.DOC.

Giả sử LETTER~1.DOC, LETTER~3.DOC tồn tại, tiếp theo sẽ là LETTER~2.DOC.

Giải thuật chọn tên được tối ưu tốc độ và chọn “n” dựa trên đặc tính của tên 8.3, suffix là “~n”.

1. Thao tác trên FAT
2. Tạo thư mục

Khi một thư mục được tạo, bit 4 trong byte thuộc tính (Directory) được set, trường File size sẽ là zero. Chúng ta duyệt thư mục bằng cluster chain trong bảng FAT tới khi gặp EOC (FAT entry có giá trị >= 0xFFFFFFF8).

Khi cluster được cấp cho thư mục, 2-byte cluster thấp và cao đều trỏ vào vị trí cluster này và đặt luôn EOC vào cluster entry (trên bảng FAT) tương ứng.

Tiếp theo, khởi tạo tất cả byte của cluster cấp phát về zero. Nếu là thư mục root thì không phải làm gì nữa.

Nếu là thư mục con, phải tạo thêm hai 32-bytes entry đặc biệt vào vùng data đầu tiên của cluster cấp phát:

* Đầu tiên là *dot* entry có tên “. ”, đại diện cho entry thư mục hiện tại.  
  DIR\_FstClusLO và DIR\_FstClusHI trỏ vào vị trí cluster đang chứa dot và dotdot entry, hay cũng là cluster vừa cấp phát cho thư mục mới tạo.
* Tiếp theo là *dotdot* entry có tên “.. ”, đại diện cho entry thư mục cha của thư mục hiện tại.  
  DIR\_FstClusLO và DIR\_FstClusHI trỏ tới vị trí cluster đầu tiên của thư mục cha (zero nếu thư mục cha là root).

Cả hai đều set File size về zero, các nhãn thời gian, ngày tháng có cùng giá trị với entry thư mục cha.

Trên FAT32, entry thư mục gốc hay thư mục con có kích thước không cố định, dạng cluster chain. Khác với thư mục thông thường, thư mục root trên mọi kiểu FAT đều không có nhãn thời gian, ngày tháng, tên file chỉ là “\”, không chứa “.” và “..” cho 2 direcotry record đầu tiên trong thư mục.

1. Tạo file

Find first free entry in directory and write file name

Search FAT for unallocated cluster; set to EOF (0xFFF)

Write that cluster's address into directory entry

If another cluster is needed,

find an unallocated FAT entry,

reset that value to EOF, and

reset previous FAT pointer to this new cluster

Repeat this step as necessary

1. Xóa file/thư mục

Find directory entry for file to delete

Using starting cluster value in the root directory, set all FAT entries in file's cluster chain to zero

Deallocate directory entry by overwriting first byte of the entry with 0xE5 (å)

1. Tìm file/thư mục

1. Tìm Rood directory

Find first cluster number in directory entry of interesting file or directory

2. Figure out the sector to read using cluster number and FirstSectorofCluster equation

3. Read that cluster

4. Figure out if file or directory continues past cluster by looking up FAT[current\_cluster\_number]

If EoC mark stop

Else go to 3 with cluster=FAT[current cluster number]

1. Effect of Long Directory Entries on Down Level Versions of FAT

The support of long names is most important on the hard disk, however it will be supported on removable media as well. The implementation provides support for long names without breaking compatibility with the existing FAT format. A disk can be read by a down level system without any compatibility problems. An existing disk does not go through a conversion process before it can start using long names. All of the current files remain unmodified. The long name directory entries are added when a long name is created. The addition of a long name to an existing file may require the 8.3 directory entry to be moved if the required adjacent directory entries are not available.

The long name entries are as hidden as hidden or system files are on a down level system. This is enough to keep the casual user from causing problems. The user can copy the files off using the 8.3 name, and put new files on without any side effects

The interesting part of this is what happens when the disk is taken to a down level FAT system and the directory is changed. This can affect the long name entries since the down level system ignores these long names and will not ensure they are properly associated with the 8.3 names.

A down level system will only see the long name entries when searching for a label. On a down level system, the volume label will be incorrectly reported if the true volume label does not come before all of the long name entries in the root directory. This is because the long name entries also have the volume label bit set. This is unfortunate, but is not a critical problem.

If an attempt is made to remove the volume label, one of the long name directory entries may be deleted. This would be a rare occurrence. It is easily detected on an aware system. The long name entry will no longer be a valid file entry, since one or more of the long entries is marked as deleted. If the deleted entry is reused, then the attribute byte will not have the proper value for a long name entry.

If a file is renamed on a down level system, then only the short name will be renamed. The long name will not be affected. Since the long and short names must be kept consistent across the name space, it is desirable to have the long name become invalid as a result of this rename. The checksum of the 8.3 name that is kept in the long name directory provides the ability to detect this type of change. This checksum will be checked to validate the long name before it is used. Rename will cause problems only if the renamed 8.3 file name happens to have the same checksum. The checksum algorithm chosen has a relatively flat distribution across the short name space.

This rename of the 8.3 name must also not conflict with any of the long names. Otherwise a down level system could create a short name in one file that matches a long name, when case is ignored, in a different file. To prevent this, the automatic creation of an 8.3 name from a long name, that has an 8.3 format, will directly map the long name to the 8.3 name by converting the characters to upper case.

If the file is deleted, then the long name is simply orphaned. If a new file is created, the long name may be incorrectly associated with the new file name. As in the case of a rename the checksum of the 8.3 name will help prevent this incorrect association.

1. Giải thuật phân loại Directory

These guidelines are provided so that disk maintenance utilities can verify individual directory entries for 'correctness' while maintaining compatibility with future enhancements to the directory structure.

1. *DO NOT* look at the content of directory entry fields marked 'reserved' and assume that, if they are any value other than zero, that they are 'bad'.

2. *DO NOT* reset the content of directory entry fields marked *reserved* to zero when they contain non-zero values (under the assumption that they are "bad"). Directory entry fields are designated *reserved,* rather than *must-be-zero.*  They should be ignored by your application.. These fields are intended for future extensions of the file system. By ignoring them an utility can continue to run on future versions of the operating system.

3. *DO* use the A\_LONG attribute *first* when determining whether a directory entry is a long directory entry or a short directory entry. The following algorithm is the correct algorithm for making this determination:

if (((LDIR\_attr & ATTR\_LONG\_NAME\_MASK) == ATTR\_LONG\_NAME) && (LDIR\_Ord != 0xE5))  
{  
 /\* Found an active long name sub-component. \*/  
}

4. *DO* use bits 4 and 3 of a short entry *together* when determining what type of short directory entry is being inspected. The following algorithm is the correct algorithm for making this determination:

if (((LDIR\_attr & ATTR\_LONG\_NAME\_MASK) != ATTR\_LONG\_NAME) && (LDIR\_Ord != 0xE5))  
{  
 if ((DIR\_Attr & (ATTR\_DIRECTORY | ATTR\_VOLUME\_ID)) == 0x00)  
 /\* Found a file. \*/  
 else if ((DIR\_Attr & (ATTR\_DIRECTORY | ATTR\_VOLUME\_ID)) == ATTR\_DIRECTORY)  
 /\* Found a directory. \*/  
 else if ((DIR\_Attr & (ATTR\_DIRECTORY | ATTR\_VOLUME\_ID)) == ATTR\_VOLUME\_ID)  
 /\* Found a volume label. \*/  
 else  
 /\* Found an invalid directory entry. \*/  
}

5. *DO NOT* assume that a non-zero value in the "type" field indicates a bad directory entry. Do not force the "type" field to zero.

6. Use the "checksum" field as a value to validate the directory entry. The "first cluster" field is currently being set to zero, though this might change in future.

1. Ghi chú thêm về FAT Directories

* LFN entry hỗ trợ trên mọi kiểu FAT.
* Trên FAT32 file system không được tạo cluster chain dài hơn 0x100000000 bytes, và byte cuối cùng trên cluster cuối cùng trong cluster chain không được cấp phát cho file.

Đó là vì FAT file system giới hạn file size <= 0xFFFFFFFF bytes (4,294,967,295)

* Do đó, FAT file system không cho phép thư mục lớn hơn 65,536-entry \* 32 (2,097,152) bytes.   
    
  **NOTE:** This limit does *not* apply to the number of files in the directory. This limit is on the size of the directory itself and has nothing to do with the content of the directory. There are two reasons for this limit:

1. Because FAT directories are not sorted or indexed, it is a bad idea to create huge directories; otherwise, operations like creating a new entry (which requires every allocated directory entry to be checked to verify that the name doesn’t already exist in the directory) become very slow.
2. There are many FAT file system drivers and disk utilities, including Microsoft’s, that expect to be able to count the entries in a directory using a 16-bit WORD variable. For this reason, directories cannot have more than 16-bits worth of entries.
3. NTFS layout
4. Thao tác trên đĩa cứng
5. DASD handle

Truy xuất trực tiếp tới đĩa cứng hay Volume bị cấm từ Windows Vista về sau.

Chúng ta phải dùng CreateFile để tạo handle (DASD Handle), handle truy xuất trực tiếp thiết bị lưu trữ.

DASD handle được dùng với DeviceIoControl để lấy metadata disk hay bảng partition. Điều này yêu cầu lập trình viên phải cẩn thận, vì có thể làm mất dữ liệu trên Volume khiến hệ điều hành không truy xuất được dữ liệu.

Điều kiện để phương thức gọi thành công:

* Phải có quyền Administrator.
* dwCreationDisposition : phải là **OPEN\_EXISTING**
* dwShareMode : phải là **FILE\_SHARE\_WRITE** đối với đĩa cứng hay ổ đĩa mềm
* dwDesiredAccess  : có thể là zero, khi ứng dụng chỉ lấy thuộc tính Volume mà không truy xuất tới. Ví dụ: lấy thông tin kích thước đĩa mềm và định dạng mà nó hỗ trợ, không cần phải có đĩa trong ổ.
* lpFileName  : Dùng Win32 namespace, dạng "\\.\PhysicalDriveX” , không có “\” ở cuối. X là số thứ tự đĩa cứng bắt đầu từ zero.

|  |  |
| --- | --- |
| **String** | **Meaning** |
| "\\.\PhysicalDrive0" | Ổ cứng đầu tiên |
| "\\.\PhysicalDrive2" | Ổ cứng thứ 3 |

1. Volume handle

Khi mở Handle cho Volume hay ổ di động (đĩa mềm, flash) dùng CreateFile:

* lpFileName : Dùng Win32 namespace, dạng "\\.\C:” - “\\.\HarddiskVolume1” hay “\\.\GUID” path không có “\” ở cuối

|  |  |
| --- | --- |
| **String** | **Meaning** |
| "\\.\A:" | Ổ đĩa mềm A. |
| "\\.\C:" | Truy xuất tới Volume C: |

Handle Volume độc lập với file system, dù có tùy chọn non-cached hay không.

Handle Volume nên mở với non-cached option, các điều giới hạn trên I/O non-chard cho file cũng sẽ áp dụng cho volume.

**Lưu ý:**  Để đọc hay ghi các sector cuối của volume, phải gọi DeviceIoControl với flag [**FSCTL\_ALLOW\_EXTENDED\_DASD\_IO**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa364556(v=vs.85).aspx)**, file system sẽ không kiểm ra ranh giới partition cho tiến trình I/O, thay vào đó, chỉ kiểm tra ranh giới của đĩa cứng mà thôi.**

 Một Volume có thể trải trên nhiều ổ đĩa vật lý, dùng DeviceIoControl với flag [**IOCTL\_VOLUME\_GET\_VOLUME\_DISK\_EXTENTS**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365194(v=vs.85).aspx) **để lấy số đĩa và offset trên đĩa cứng phân bổ cho Volume.**

1. File system handle

Khi mở Handle cho Volume hay ổ di động (đĩa mềm, flash) dùng CreateFile:

* lpFileName : Dùng Win32 namespace, dạng "\\.\C:\”, phải có (\) ở cuối.  
  Ví dụ:

|  |  |
| --- | --- |
| **String** | **Meaning** |
| "\\.\C:\" | Truy xuất tới file system trên C: volume. |

File system không yêu cầu bộ đệm dù data không dùng cached.

1. Disk Control code

Basic disks và volumes quản lý bởi Disk Management snap-in. Các code sau dùng bởi Disk Management snap-in:

|  |  |
| --- | --- |
| **Control Code** | **Operation** |
| IOCTL\_DISK\_CREATE\_DISK | Khởi tạo đĩa cứng và bảng partition theo thông tin cấu hình. |
| IOCTL\_DISK\_DELETE\_DRIVE\_LAYOUT | Xóa phần phần kết của MBR. |
| IOCTL\_DISK\_GET\_DRIVE\_GEOMETRY\_EX | Lấy thông tin về tổ chức của đĩa cứng. |
| IOCTL\_DISK\_GET\_DRIVE\_LAYOUT\_EX | Lấy thông tin về số partition của đĩa và thông tin mỗi partition. |
| IOCTL\_DISK\_GET\_LENGTH\_INFO | Lấy kích thước của đĩa, volume hay partition. |
| IOCTL\_DISK\_GET\_PARTITION\_INFO\_EX | Lấy thông tin về AT và EFI (Extensible Firmware Interface) partitions. |
| IOCTL\_DISK\_GROW\_PARTITION | Mở rộng partition |
| IOCTL\_DISK\_SET\_DRIVE\_LAYOUT\_EX | Partitions đĩa. |
| IOCTL\_DISK\_SET\_PARTITION\_INFO\_EX | Cấu hình kiểu partition. |

1. IOCTL\_VOLUME\_GET\_VOLUME\_DISK\_EXTENTS

Lấy vị trí vậy lý của volume trên một hay nhiều đĩa cứng.

BOOL WINAPI

DeviceIoControl( (HANDLE) hDevice, // handle to device

IOCTL\_VOLUME\_GET\_VOLUME\_DISK\_EXTENTS, // dwIoControlCode

NULL, // Không dùng để NULL

0, // Để ZERO

(LPVOID) lpOutBuffer, // output buffer

(DWORD) nOutBufferSize, // kích thước theo byte của bộ đệm

(LPDWORD) lpBytesReturned, // Kích thước dữ liệu trả về cho bộ đệm out

(LPOVERLAPPED) lpOverlapped // Không dùng để NULL

);

Hàm trả về TRUE nếu thành công, FALSE nếu thất bại. GetLastError để bug lỗi.

\_ GetLastError trả về ERROR\_INSUFFICIENT\_BUFFER nếu bộ đệm lpOutBuffer nhỏ hơn sizeof(VOLUME\_DISK\_EXTENTS) và lpBytesReturned = zero.

\_ GetLastError trả về ERROR\_MORE\_DATA báo bộ đệm không đủ chứa dữ liệu trả về. Chúng ta phải gọi IOCTL\_VOLUME\_GET\_VOLUME\_DISK\_EXTENTS để lấy nốt dữ liệu còn lại.

* lpOutBuffer[out]: Bộ đệm cấu trúc VOLUME\_DISK\_EXTENTS nhận thông tin đĩa cứng.  
  Cấu trúc VOLUME\_DISK\_EXTENTS:

typedef struct \_VOLUME\_DISK\_EXTENTS {

DWORD NumberOfDiskExtents; //Số đĩa trong Volume

DISK\_EXTENT Extents[ANYSIZE\_ARRAY]; //Mảng DISK\_EXTENT

} VOLUME\_DISK\_EXTENTS, \*PVOLUME\_DISK\_EXTENTS;

Volume có thể trải dài trên nhiều đĩa (Sector liên tiếp nhau)

Khi số NumberOfDiskExtents > 1, hàm FALSE, GetLastError trả về ERROR\_MORE\_DATA, phải gọi DeviceIoControl lần nữa và tính toán vùng nhớ bộ đệm phù hợp với số NumberOfDiskExtents.

Cấu trúc DISK\_EXTENT:

typedef struct \_DISK\_EXTENT {

DWORD DiskNumber; //Hậu tố X trong "\\?\PhysicalDriveX" hay "\\?\HarddiskX

LARGE\_INTEGER StartingOffset; //offset byte bắt đầu của đĩa mở rộng

LARGE\_INTEGER ExtentLength; //Kích thước của phần mở rộng

} DISK\_EXTENT, \*PDISK\_EXTENT;

1. IOCTL\_DISK\_GET\_DRIVE\_GEOMETRY\_EX

Lấy thông tin về đĩa cứng: loại, số cylinders, tracks per cylinder, sectors per track, và bytes per sector.

BOOL DeviceIoControl(

(HANDLE) hDevice, // handle to volume

IOCTL\_DISK\_GET\_DRIVE\_GEOMETRY\_EX, // dwIoControlCode

NULL, // Không dùng để NULL

0, // Để ZERO

(LPVOID) lpOutBuffer, // Bộ đệm kiểu cấu trúc DISK\_GEOMETRY\_EX nhận dữ liệu trả về

(DWORD) nOutBufferSize, // Kích thước bộ đệm tính theo byte

(LPDWORD) lpBytesReturned, // Kích thước dữ liệu trả về trong bộ đệm out, tính theo byte

(LPOVERLAPPED) lpOverlapped // Không dùng để NULL

);

Nếu bộ đệm lpOutBuffer không đủ cho dữ liệu trả về, hàm FALSE và GetLastError trả về ERROR\_INSUFFICIENT\_BUFFER, lpBytesReturned = zero.

Hàm trả về TRUE nếu thành công, FALSE nếu thất bại. GetLastError để bug lỗi.

* Cấu trúc DISK\_GEOMETRY\_EX:

typedef struct \_DISK\_GEOMETRY\_EX {

DISK\_GEOMETRY Geometry; //Cấu trúc DISK\_GEOMETRY

LARGE\_INTEGER DiskSize; //Kích thước đĩa cứng theo byte

BYTE Data[1]; //

} DISK\_GEOMETRY\_EX, \*PDISK\_GEOMETRY\_EX;

* Cấu trúc DISK\_GEOMETRY:

typedef struct \_DISK\_GEOMETRY {

LARGE\_INTEGER Cylinders; //Tổng số Cylinder

MEDIA\_TYPE MediaType; // Hằng số, RemovableMedia là ổ đĩa di động,FixedMedia là ổ cố định

DWORD TracksPerCylinder; //Số Track trên 1 Cylinder

DWORD SectorsPerTrack; //Số Sector trên 1 Track

DWORD BytesPerSector; //Số Byte trên 1 Sector

} DISK\_GEOMETRY;

1. Cấu trúc bộ đệm
2. CREATE\_DISK

Cấu trúc bộ đệm CREATE\_DISK được áp dụng để khởi tạo MBR hay GUID partition table GPT cho đĩa:

typedef struct {

PARTITION\_STYLE PartitionStyle;//Kiểu enum \_PARTITION\_STYLE

union {

CREATE\_DISK\_MBR Mbr;//Thông tin để khởi tạo Đĩa MBR

CREATE\_DISK\_GPT Gpt;//Thông tin để khởi tạo Đĩa GPT

};

} CREATE\_DISK, \*PCREATE\_DISK;

1. Enum \_PARTITION\_STYLE

* PartitionStyle: Kiểu partition định nghĩa bởi kiểu enum \_PARTITION\_STYLE:

|  |  |
| --- | --- |
| \_PARTITION\_STYLE | Định dạng Partition |
| **PARTITION\_STYLE\_MBR**  **0** | Master boot record (MBR) format theo chuẩn AT-style MBR partitions. |
| **PARTITION\_STYLE\_GPT**  **1** | GUID Partition Table (GPT) format. |
| **PARTITION\_STYLE\_RAW**  **2** | Partition không phải MBR hay GPT. |

1. CREATE\_DISK\_MBR

* Mbr: Cấu trúc CREATE\_DISK\_MBR

typedef struct {

DWORD Signature;// Disk signature để khởi tạo Đĩa MBR

} CREATE\_DISK\_MBR, \*PCREATE\_DISK\_MBR;

1. CREATE\_DISK\_GPT

* Gpt: Cấu trúc A CREATE\_DISK\_GPT

typedef struct {

GUID DiskId; //Mã số nhận dạng GUID để khởi tạo Đĩa GPT

DWORD MaxPartitionCount;//Số partition tối đa của Đĩa GPT.

} CREATE\_DISK\_GPT, \*PCREATE\_DISK\_GPT;

Mặc định MaxPartitionCount tối thiểu là 128, nếu để nhỏ hơn, hệ thống tự reset lên 128. Thông số này qui định bởi [**Extensible Firmware Interface**](https://msdn.microsoft.com/en-us/library/windows/hardware/dn614600(v=vs.85).aspx) (EFI)

1. DRIVE\_LAYOUT\_INFORMATION\_EX

Dùng để lấy thông tin định dạng hay thiết lập định dạng partition cho đĩa cứng.

* Cấu trúc DRIVE\_LAYOUT\_INFORMATION\_EX:

typedef struct \_DRIVE\_LAYOUT\_INFORMATION\_EX {

DWORD PartitionStyle;//Kiểu enum \_PARTITION\_STYLE

DWORD PartitionCount;//Số Partition

union {

DRIVE\_LAYOUT\_INFORMATION\_MBR Mbr;//Thông tin MBR

DRIVE\_LAYOUT\_INFORMATION\_GPT Gpt;//Thông tin GPT

};

PARTITION\_INFORMATION\_EX PartitionEntry[1];//Mảng Thông tin từng Partition, mỗi phần tử là 1 partition trên đĩa

} DRIVE\_LAYOUT\_INFORMATION\_EX, \*PDRIVE\_LAYOUT\_INFORMATION\_EX;

* PartitionCount: Nếu là đĩa MBR, luôn là bộ số của 4.

1. DRIVE\_LAYOUT\_INFORMATION\_MBR

* Mbr: Chứa thông tin Disk Signature cho MBR layout.

typedef struct \_DRIVE\_LAYOUT\_INFORMATION\_MBR {

ULONG Signature;//Disk Signature

} DRIVE\_LAYOUT\_INFORMATION\_MBR, \*PDRIVE\_LAYOUT\_INFORMATION\_MBR;

1. DRIVE\_LAYOUT\_INFORMATION\_GPT

* Gpt: Chứa thông tin của đĩa GPT layout.
* typedef struct \_DRIVE\_LAYOUT\_INFORMATION\_GPT {

GUID DiskId; //GUID của đĩa

LARGE\_INTEGER StartingUsableOffset;//

LARGE\_INTEGER UsableLength; //

ULONG MaxPartitionCount;// Số partition tối đa của đĩa

} DRIVE\_LAYOUT\_INFORMATION\_GPT, \*PDRIVE\_LAYOUT\_INFORMATION\_GPT;

StartingUsableOffset

The starting byte offset of the first usable block.

UsableLength

The size of the usable blocks on the disk, in bytes.

MaxPartitionCount

The maximum number of partitions that can be defined in the usable block.

1. PARTITION\_INFORMATION\_EX

Cấu trúc PARTITION\_INFORMATION\_EX để định dạng cho parition.

typedef struct {

PARTITION\_STYLE PartitionStyle; //Kiểu Partition

LARGE\_INTEGER StartingOffset; //Offset byte bắt đầu của partition

LARGE\_INTEGER PartitionLength; //Kích thước partition theo byte

DWORD PartitionNumber; //Số thứ tự partition tính từ 1.

BOOLEAN RewritePartition; //Nên để TRUE: paritition có thể được định dạng lại

union {

PARTITION\_INFORMATION\_MBR Mbr;//Thông tin partition MBR

PARTITION\_INFORMATION\_GPT Gpt;// Thông tin partition GPT

};

} PARTITION\_INFORMATION\_EX;

* PartitionStyle:

|  |  |
| --- | --- |
| **Constant/value** | **Description** |
| **PARTITION\_ENTRY\_UNUSED**  0x00 | Partition chưa dùng. |
| **PARTITION\_EXTENDED**  0x05 | Partition mở rộng |
| **PARTITION\_FAT\_12**  0x01 | Partition định dạng file system FAT 12 |
| **PARTITION\_FAT\_16**  0x04 | Partition định dạng file system FAT 16 |
| **PARTITION\_FAT32**  0x0B | Partition định dạng file system FAT 32 |
| **PARTITION\_IFS**  0x07 | Partition IFS |
| **PARTITION\_LDM**  0x42 | Partition logical disk manager (LDM). |
| **PARTITION\_NTFT**  0x80 | Partition định dạng file system NTFS |
| **VALID\_NTFT**  0xC0 | Partition NTFT hợp lệ.  The high bit of a partition type code indicates that a partition is part of an NTFT mirror or striped array. |

1. IOCTL\_DISK\_CREATE\_DISK

Để khởi tạo đĩa theo chuẩn MBR hay GPT

BOOL DeviceIoControl( (HANDLE) hDevice, // handle to device

IOCTL\_DISK\_CREATE\_DISK, // dwIoControlCode

(LPVOID) lpInBuffer, // Bộ đệm cấu trúc CREATE\_DISK

(DWORD) nInBufferSize, // Kích thước bộ đệm tính theo byte

(LPVOID) NULL, // Không dùng để NULL

(DWORD) 0, // Để ZERO

(LPDWORD) lpBytesReturned, // number of bytes returned

(LPOVERLAPPED) lpOverlapped ); // Để NULL

Nếu tạo đĩa GPT, ứng dụng phải đợi cho tới khi MSR partition hoàn thành trước khi dùng IOCTL\_DISK\_SET\_DRIVE\_LAYOUT\_EX để định dạng partition. Tham khảo [**RegisterDeviceNotification**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa363431(v=vs.85).aspx).

Hàm trả về TRUE nếu thành công, FALSE nếu thất bại. GetLastError để bug lỗi.

1. IOCTL\_DISK\_GET\_DRIVE\_LAYOUT\_EX

Để nhận diện loại partition dùng DeviceIoControl với flag IOCTL\_DISK\_GET\_DRIVE\_LAYOUT\_EX, kiểm tra PartitionType cho từng entry trong danh sách partition trả về

BOOL DeviceIoControl( (HANDLE) hDevice, // handle to device

IOCTL\_DISK\_GET\_DRIVE\_LAYOUT\_EX, // dwIoControlCode

NULL, // Không dùng để NULL

0, // Để ZERO

(LPVOID) lpOutBuffer, // Bộ đệm cấu trúc DRIVE\_LAYOUT\_INFORMATION\_EX

(DWORD) nOutBufferSize, // Kích thước bộ đệm tính theo byte

(LPDWORD) lpBytesReturned, // Kích thước dữ liệu trả về trong bộ đệm out

(LPOVERLAPPED) lpOverlapped ); // Không dùng để NULL

Nếu bộ đệm không đủ cho dữ liệu trả về, hàm FALSE và GetLastError trả về ERROR\_INSUFFICIENT\_BUFFER, lpBytesReturned = zero.

Hàm trả về TRUE nếu thành công, FALSE nếu thất bại. GetLastError để bug lỗi.

1. IOCTL\_DISK\_SET\_DRIVE\_LAYOUT\_EX

Định dạng Partition layout cho đĩa cứng.

BOOL DeviceIoControl(

(HANDLE) hDevice, // handle to device

IOCTL\_DISK\_SET\_DRIVE\_LAYOUT\_EX, // dwIoControlCode

(LPVOID) lpInBuffer, // Cấu trúc DRIVE\_LAYOUT\_INFORMATION\_EX chứa thông tin layout

(DWORD) nInBufferSize, // Kích thước bộ đệm tính theo byte

NULL, // Không dùng để NULL

0, // Để ZERO

(LPDWORD) lpBytesReturned, // number of bytes returned

(LPOVERLAPPED) lpOverlapped // Không dùng để NULL

);

Hàm trả về TRUE nếu thành công, FALSE nếu thất bại. GetLastError để bug lỗi.

Khi định dạng, MBR partition tối đa là 2TB. Dùng GUID partition để mở rộng hơn 2 TB.

1. IOCTL\_DISK\_SET\_PARTITION\_INFO\_EX

BOOL DeviceIoControl(

(HANDLE) hDevice, // handle to device

IOCTL\_DISK\_SET\_PARTITION\_INFO\_EX, // dwIoControlCode

(LPVOID) lpInBuffer, // input buffer

(DWORD) nInBufferSize, // size of input buffer

NULL, // lpOutBuffer

0, // nOutBufferSize

(LPDWORD) lpBytesReturned, // number of bytes returned

(LPOVERLAPPED) lpOverlapped // OVERLAPPED structure

);

1. IOCTL\_DISK\_DELETE\_DRIVE\_LAYOUT
2. IOCTL\_DISK\_UPDATE\_PROPERTIES

### Quản lý Event Device

### Liên kết và Reparse point

1. Liên kết thư mục – file

1 link là một tên thay thế cho file hay thư mục. Tạo ra Hard Link đồng nghĩa với tạo ra bản sao file.

Symbolic link lại giúp tiết kiệm vùng đĩa cứng, tất cả link chỉ tham chiếu tới cùng 1 file mà không tạo bản sao.

Unix dùng command “ln” để tạo link

Kể từ Windows 7, command “mklink” dùng để tạo link.

1. NTFS Hard link

Khi 1 hard link tới file được tạo, hệ thống sẽ tạo ra thuộc tính tên mới, làm tăng số lượng hard link và khi xóa, sẽ làm giảm số lượng hard link. Khi số hard link về zero, file sẽ bị xóa hoàn toàn, thu hồi đĩa cứng và giải phóng MFT(Master file table) record

Tất cả link độc lập. Khi xóa, di chuyển hay đổi tên không ảnh hưởng tới nhau.

* Dùng CreateHardLink để tạo hard link mới
* Dùng DeleteFile để xóa hard link : hệ thống giảm số hard link trong MFT record
* Dùng GetFileInformationByHandle, tham số nNumberOfLinks cho biết số hard link của file.

Vì tất cả hard link đều tham chiếu tới cùng MFT entry, nên không thể tạo hard link trên ổ đĩa khác. Và chỉ có file mới được tạo hard link, thư mục không được.

Một file chỉ có thể tạo tối đa 1023 hard link.

Hàm CreateHardLink chỉ làm việc trên NTFS file system.

BOOL WINAPI CreateHardLink(

\_In\_ LPCTSTR lpFileName,

\_In\_ LPCTSTR lpExistingFileName,

\_Reserved\_ LPSECURITY\_ATTRIBUTES lpSecurityAttributes

);

* lpFileName [in]: tên file liên kết để mở file gốc, không hỗ trợ thư mục
* lpExistingFileName [in]: đường dẫn file gốc
* lpSecurityAttributes: Phải là NULL. Thành viên Security descriptors thuộc về file, không phải hard link.

Nếu thiết lập thuộc tính bảo mật khác, sẽ làm thay đổi bảo mật của file (cũng như tất cả hard link), hàm sẽ thất bại nếu user không sở hữu hay không phải Administrator.

Hàm trả về true nếu thành công, ngược lại trả về false. GetLastError để bug lỗi.

1. Symbolic link

Liên kết Symbolic tạo đường dẫn tắt (shortcut) tới file hay thư mục

BOOLEAN WINAPI CreateSymbolicLink(

\_In\_ LPTSTR lpSymlinkFileName,

\_In\_ LPTSTR lpTargetFileName,

\_In\_ DWORD dwFlags

);

* lpSymlinkFileName [in]: Tên liên kết cho file gốc. Nếu file gốc là tên thiết bị, liên kết sẽ là tuyệt đối, ngược lại liên kết chỉ mang tính quan hệ.
* lpTargetFileName [in]: đường dẫn file hay thư mục cho liên kết.
* dwFlags [in]: Flag ám chỉ file gốc là file hay thư mục

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| 0x0 | Đối tượng là file. |
| **SYMBOLIC\_LINK\_FLAG\_DIRECTORY**  0x1 | Đối tượng là thư mục |

Hàm trả về true nếu thành công, ngược lại trả về false.

1. Junction point- Soft link

Một junction hay còn gọi là Soft link, Soft link tham chiếu tới thư mục , có thể liên kết nhiều thư mục khác ổ đĩa logic trên cùng một máy. Cơ chế Soft link hoạt động tương tự hard link.

junction point triển khai thông qua reparse points.

Ví dụ: Documents and Settings là một junction points

Windows Vista và Windows Server 2008, user data bị chuyển từ “%SystemDrive%\Documents and Settings“ sang “%SystemDrive%\Users”

Để người dùng dễ thích nghi, junction point được set. “C:\Documents and Settings” trỏ tới “C:\Users”

Thư mục là Junction point có các thuộc tính: FILE\_ATTRIBUTE\_REPARSE\_POINT, FILE\_ATTRIBUTE\_HIDDEN, và FILE\_ATTRIBUTE\_SYSTEM được set.

Junction point cũng áp dụng (ACLs).

Ứng dụng sao lưu, duyệt file system phải nhận biết được junction points. Có 2 lý do:

* Xử lí cùng một dữ liệu nhiều lần: nếu junction trỏ tới thư mục cùng Volume
* Vòng lặp vô hạn: 2 junction đang trỏ tới nhau.

1. Per-User Junctions và System Junctions

Ảo hóa file và registry trên Window dùng junction point chia làm 2 loại:

* per-user junction: tự động tạo ra cho mỗi user’s profile mới, như “C:\Users\username\My Documents” that trỏ tới “C:\Users\username\Documents”.  
  Ví dụ: Documents and Settings Junctions trong tất cả Users, Public, và Default User profiles
* System junctions: tạo bởi thư viện userenv.dll when it is invoked by Windows Welcome (also called the machine out-of-box-experience, or mOOBE).  
  Các junction nằm ngoài “Users\username” đều là system junction

Lưu ý: Nếu thay đổi ngôn ngữ hệ thống khác English, per-user và hệ thống junction point được tạo tương ứng với tên địa phương.

1. Liên kết thư mục - volume

Chỉ NTFS volume mới tạo được mounted folder, một dạng link giữa volume và thư mục.

Khi mounted folder được tạo, user và ứng dụng truy xuất volume qua mounted folder thay vì Drive letter.

Ví dụ tạo mounted folder C:\D\_drive tương ứng với ổ D:\. Ứng dụng dùng đường dẫn C:\D\_drive để truy xuất ổ D:\

Điểm đặc biệt là chúng ta có thể mount với đủ file system như NTFS, FAT 16, FAT 32 và ngay cả ISO-9660 CD-ROM trên NFTS volume.

1. Tạo Mounted Folder

Gồm 2 bước:

* Gọi GetVolumeNameForVolumeMountPoint : Lấy GUID path của volume
* Dùng SetVolumeMountPoint: Liên kết GUID path với thư mục được mount.

Chúng ta có thể chọn bất kì thư mục trống để mount.

Có thể liên kết một volume ảo tới nhiều mounted folder. Các mounted folder được duy trì kể cả khi hệ thống khởi động lại.

Nếu một volume bị lỗi, không thể truy xuất mounted folder.

Ví dụ: khi ổ D:\ bị lỗi, Mounted folder C:\D\_drive link tới volume D:\ không thể truy xuất được nữa

Cơ chế liên kết này được triển khai thông qua reparse point, các API như SetVolumeMountPoint tự xử lý reparse point.

Chúng ta vẫn có thể đổi tên, xóa, hay di chuyển mounted folder.

1. GetVolumeNameForVolumeMountPoint

Lấy GUID path của Volume path hay Volume Mount point.

BOOL WINAPI GetVolumeNameForVolumeMountPoint(

\_In\_ LPCTSTR lpszVolumeMountPoint,

\_Out\_ LPTSTR lpszVolumeName,

\_In\_ DWORD cchBufferLength

);

* lpszVolumeMountPoint [in]:   
  \_ Trường hợp tham số là đường dẫn Volume path như “C:\”, hàm trả về GUID path.  
  \_ Trường hợp tham số là đường dẫn mounted folder, hàm trả về Volume name trỏ tới  
  \_ Tham số phải có “\” ở cuối.
* lpszVolumeName [out]: Bộ đệm nhận GUID path .Nếu có nhiều GUID, chỉ có GUID đầu tiên trong bộ quản lý mount cache trả về.
* cchBufferLength [in]: Kích thước bộ đệm, tính theo số ký tự TCHAR, 50 là hợp lý.

Hàm trả về TRUE nếu thành công, FALSE nếu thất bại. GetLastError để bug lỗi.

GUID path trả về được dùng cho SetVolumeMountPoint hay FindFirstVolumeMountPoint.

1. SetVolumeMountPoint

Liên kết mouted folder với Volume.

BOOL WINAPI SetVolumeMountPoint(

\_In\_ LPCTSTR lpszVolumeMountPoint,

\_In\_ LPCTSTR lpszVolumeName

);

* pszVolumeMountPoint [in]: Đường dẫn mounted folder, phải có “\” ở cuối. Hoặc GUID path hay Drive letter
* lpszVolumeName [in]: GUID path của Volume được mount tới.

Hàm trả về TRUE nếu thành công, FALSE nếu thất bại. GetLastError để bug lỗi.

Nếu pszVolumeMountPoint đã là một mounted folder, hàm thất bại và GetLastError trả về ERROR\_DIR\_NOT\_EMPTY dù thư mục trống đi nữa.

Nếu thư mục có chứa file hay thư mục con, sẽ xảy ra lỗi, không thể liên kết.

1. DeleteVolumeMountPoint

Xóa Volume ảo tương ứng với lpszVolumeMountPoint.

BOOL WINAPI DeleteVolumeMountPoint(

\_In\_ LPCTSTR lpszVolumeMountPoint

);

* lpszVolumeMountPoint [in]: Đường dẫn mounted folder, ví dụ “C:\MountFolder\” hay thư mục gốc “C:\”, phải có “\” ở cuối.

Hàm trả về TRUE nếu thành công, FALSE nếu thất bại. GetLastError để bug lỗi.

Xóa mounted folder sẽ không xóa các file, thư mục con bên trong.

Nếu lpszVolumeMountPoint không phải mounted folder, hàm không làm gì cả.

1. Liệt kê mounted folder

Sử dụng các API sau:

* FindFirstVolumeMountPoint
* FindNextVolumeMountPoint
* FindVolumeMountPointClose

Cách dùng tương tự như FindFirstFile, FindNextFile và FindClose.

Nguyên tắc là kiểm tra Volume có hỗ trợ mounted folder:

1. Lấy Volume name (GUID path) trả về từ FindFirstVolume và FindNextVolume
2. Gọi GetVolumeInformation cho mỗi Volume name, kiểm tra file system có phải NTFS, lpFileSystemNameBuffer là “NTFS”
3. Nếu Volume là NTFS, bắt đầu dùng FindFirstVolumeMountPoint và FindNextVolumeMountPoint để tìm mounted folder
4. Nhận diện mounted folder

Việc kiểm tra mounted folder cần thiết khi đang dùng ứng dụng sao lưu hay tìm kiếm trên một volume. Dùng FindFirstFile và FindNextFile để duyệt thư mục.

1. Gọi GetFileAttributes: Kiểm tra đối tượng có reparse point với flag FILE\_ATTRIBUTE\_REPARSE\_POINT
2. Nếu có reparse point: Lấy reparse tag từ thành viên dwReserved0 của WIN32\_FIND\_DATA
3. Nếu reparse tag == IO\_REPARSE\_TAG\_MOUNT\_POINT : thì đối tượng là mounted folder.
4. Nếu reparse tag == IO\_REPARSE\_TAG\_SYMLINK : thì đối tượng là symbolic link.

Dùng GetVolumeNameForVolumeMountPoint cho mounted folder path để lấy Volume Name.

1. Reparse point

Là một tính năng mở rộng của NTFS, giúp file system linh hoạt và mạnh mẽ hơn.

Xuất phát từ nhu cầu của ứng dụng, muốn triển khai dữ liệu bí mật chỉ có thể xử lí bởi ứng dụng đó thành một reparse point.

Reparse point được gắn vào file hay thư mục, ứng dụng dùng reparse tag để nhận biết reparse point của nó.

Nhiều ứng dụng có thể chèn reparse point vào cùng một file hay thư mục, với các tag khác nhau. Khi người dùng truy xuất vào file, thư mục có reparse point, file system sẽ phân tích, tìm bộ lọc tương ứng với ứng dụng và truyền reparse data tới bộ lọc đó.

Bộ lọc dùng dữ liệu trong reparse point để làm mọi việc mà reparse point chỉ định. Cách reparse point làm việc tùy thuộc vào ứng dụng qui định.

Microsoft cũng tự định nghĩa các tag dành riêng và dùng triển khai các tính năng như:

* Symbolic Links: Không giống như symbolic file linking trên UNIX file systems, chỉ mô phỏng dùng reparse points. Để chuyển hướng truy xuất từ file sang file, thư mục (Windows shortcut).
* Junction Points: khá giống symbolic link, nhưng chuyển hướng truy xuất từ thư mục tới file hay thư mục khác.
* Volume Mount Points: Giống symbolic link và junction point, nhưng điều hướng truy xuất từ thư mục tới một Volume   
  Chúng ta có thể tạo volume mount points cho removable hard disks hay storage media, hay các logic partition (C:, D:, E: …).   
  Tính năng này cho phép Window 2k mở rộng hơn 26 kí tự ổ đĩa – hữu dụng cho CD-ROM server.
* Remote Storage Server (RSS): This feature of Windows 2000 uses a set of rules to determine when to move infrequently used files on an NTFS volume to archive storage (such as CD-RW or tape). When it moves a file to "offline" or "near offline" storage in this manner, RSS leaves behind reparse points that contain the instructions necessary to access the archived files, if they are needed in the future.

Một đối tượng file, thư mục chỉ được tối đa 63 reparse point.

1. Reparse tag

Reparse tag để nhận diện kiểu reparse point mà không cần phải phân tích dữ liệu người dùng bên trong.

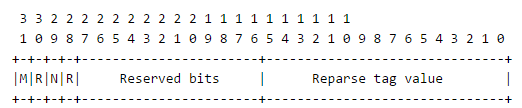
Hệ thống sử dụng một bộ tag dành riêng cho Microsoft, nếu chúng ta dùng các tag này cấu hình reparse point, tiến trình sẽ bị lỗi.

Microsoft tag định nghĩa trong WinNT.h

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| IO\_REPARSE\_TAG\_RESERVED\_ZERO  0x00000000 | Reserved reparse tag value. |
| IO\_REPARSE\_TAG\_RESERVED\_ONE  0x00000001 | Reserved reparse tag value. |
| IO\_REPARSE\_TAG\_MOUNT\_POINT  0xA0000003 | Dùng cho mount point, specified in section [2.1.2.5](https://msdn.microsoft.com/en-us/library/cc232007.aspx). |
| IO\_REPARSE\_TAG\_HSM  0xC0000004 | Obsolete. Used by legacy Hierarchical Storage Manager Product. |
| IO\_REPARSE\_TAG\_HSM2  0x80000006 | Obsolete. Used by legacy Hierarchical Storage Manager Product. |
| IO\_REPARSE\_TAG\_DRIVER\_EXTENDER  0x80000005 | Home server drive extender.[<3>](https://msdn.microsoft.com/en-us/library/cc232128.aspx#Appendix_A_3) |
| IO\_REPARSE\_TAG\_SIS  0x80000007 | Used by [single-instance storage (SIS)](https://msdn.microsoft.com/en-us/library/cc231989.aspx#gt_40c57198-06d7-4dcb-a011-d8152b64d2cb) filter driver. Server-side interpretation only, not meaningful over the wire. |
| IO\_REPARSE\_TAG\_DFS  0x8000000A | Used by the DFS filter. The DFS is described in the Distributed File System (DFS): Referral Protocol Specification [[MS-DFSC]](https://msdn.microsoft.com/en-us/library/cc226982.aspx). Server-side interpretation only, not meaningful over the wire. |
| IO\_REPARSE\_TAG\_DFSR  0x80000012 | Used by the DFS filter. The DFS is described in [MS-DFSC]. Server-side interpretation only, not meaningful over the wire. |
| IO\_REPARSE\_TAG\_FILTER\_MANAGER  0x8000000B | Used by [filter manager](https://msdn.microsoft.com/en-us/library/cc231989.aspx#gt_09f024eb-3598-47db-91e0-37c83b82bf68) test harness.[<4>](https://msdn.microsoft.com/en-us/library/cc232128.aspx#Appendix_A_4) |
| IO\_REPARSE\_TAG\_SYMLINK  0xA000000C | Dùng cho [symbolic link](https://msdn.microsoft.com/en-us/library/cc231989.aspx#gt_04f1ed93-15cb-4090-8204-c43bec8c7398) support. See section [2.1.2.4](https://msdn.microsoft.com/en-us/library/cc232006.aspx). |

Các tag không có trong danh sách, có thể dùng cho ứng dụng.

Reparse tag lưu trữ trong thành viên dữ liệu DWORD WIN32\_FIND\_DATA::dwReserved0



2-bytes thấp chỉ rõ kiểu reparse point. 2-bytes cao có 12-bits dành riêng không được dùng và 4-bits qui định thuộc tính của tag và dữ liệu trong reparse point.

Ý nghĩa 4-bits:

|  |  |
| --- | --- |
| **Bit** | **Description** |
| M | Microsoft bit. Bit được set khi tag thuộc Microsoft. Tất cả tag khác phải để zero. |
| R | Dành riêng, phải để zero cho tất cả tag không phải của Microsoft. |
| N | Bit tên đại diện. Nếu bit được set, thư mục hay file dại diện cho tên khác trong hệ thống. |

1. Nhận diện Reparse object

Gọi GetFileAttributes: Kiểm tra đối tượng reparse point với flag FILE\_ATTRIBUTE\_REPARSE\_POINT.

* Trường hợp duyệt thư mục:  
  Dùng FindFirstFile và FindNextFile nhận biết reparse tag: kiểm tra reparse tag từ thành viên dwReserved0 của WIN32\_FIND\_DATA với IO\_REPARSE\_TAG\_MOUNT\_POINT.
* Trường hợp làm việc với handle:  
  Dùng DeviceIoControl với code FSCTL\_GET\_REPARSE\_POINT, kiểm tra thông tin trả trong thành viên tham số bộ đệm REPARSE\_MOUNTPOINT\_DATA\_BUFFER.ReparseTag với flag IO\_REPARSE\_TAG\_MOUNT\_POINT.

1. Tiến trình Reparse point

Trước tiên dùng CreateFile lấy handle của file hay thư mục

* dwFlagsAndAttributes [in]: phải là FILE\_FLAG\_REPARSE\_POINT, kèm theo FILE\_FLAG\_BACKUP\_SEMANTIC nếu là thư mục.

Cấu hình bộ đệm của reparse point, tùy theo loại tag mà chọn cấu trúc bộ đệm phù hợp

* REPARSE\_DATA\_BUFFER : tag của Microsoft
* REPARSE\_GUID\_DATA\_BUFFER: tag của Microsoft hay của nhà phát triển
* REPARSE\_MOUNTPOINT\_DATA\_BUFFER: dành cho tag IO\_REPARSE\_TAG\_MOUNT\_POINT

Tùy theo tiến trình DeviceIoControl mà set thông tin bộ đệm In hay Out.

Gọi DeviceIoControl để cấu hình reparse points (Thêm, xóa, lấy, sửa). Các flag tương ứng được dùng cho reparse point:

|  |  |
| --- | --- |
| **Operation** | **Description** |
| [**FSCTL\_SET\_REPARSE\_POINT**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa364595(v=vs.85).aspx) | Thêm hay sửa reparse point. |
| [**FSCTL\_GET\_REPARSE\_POINT**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa364571(v=vs.85).aspx) | Lấy dữ liệu trong reparse point. |
| [**FSCTL\_DELETE\_REPARSE\_POINT**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa364560(v=vs.85).aspx) | Xóa reparse point. |

Khi sửa, lấy, xóa reparse point, phải dùng đúng reparse tag tương ứng trong file, thư mục.

1. Cấu trúc bộ đệm Reparse point
2. Bộ đệm chuẩn

Là cấu trúc bộ đệm nhận dữ liệu về reparse point. Chỉ được dùng với giá trị reparse tag có high bit = 1.

Bộ đệm dẫn xuất chia làm 4 loại:

* Symbolic Link Reparse Data Buffer
* Mount Point Reparse Data Buffer
* Network File System Reparse Data Buffer
* GUID Reparse Data Buffer

Dưới đây là cấu trúc tổng quát:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **1 0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **2 0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **3 0** | **1** |
| **ReparseTag (4 bytes):**Số nguyên qui định loại tag, cho biết nhà phát triển của [reparse point](https://msdn.microsoft.com/en-us/library/cc231989.aspx#gt_4fed0b53-5fc8-4818-886f-93d87f3035e1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **ReparseDataLength (2 bytes):**Kích thước theo byte của reparse data trong thành viên **DataBuffer**. | | | | | | | | | | | | | | | | **Reserved (2 bytes):**Không dùng, để 0 | | | | | | | | | | | | | | | |
| **DataBuffer (variable):**Mảng số nguyên 8-bit chứa Microsoft reparse data cho reparse point, có kích thước tùy biến  Chứa dữ liệu định dạng bởi nhà phát triển, là phía triển khai [filter](https://msdn.microsoft.com/en-us/library/cc231989.aspx#gt_ffbe7b55-8e84-4f41-a18d-fc29191a4cda) driver tương ứng với ReparseTag. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| ... | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

Cấu trúc REPARSE\_DATA\_BUFFER chỉ được dùng cho Microsoft file systems, filters, và minifilter drivers, I/O manager.

Microsoft reparse points có thể dùng REPARSE\_DATA\_BUFFER hay REPARSE\_GUID\_DATA\_BUFFER.

Cần khai báo header < Ntifs.h > trong bộ WDK

typedef struct \_REPARSE\_DATA\_BUFFER {

ULONG ReparseTag;

USHORT ReparseDataLength;

USHORT Reserved;

union {

struct {

USHORT SubstituteNameOffset;

USHORT SubstituteNameLength;

USHORT PrintNameOffset;

USHORT PrintNameLength;

ULONG Flags;

WCHAR PathBuffer[1];

} SymbolicLinkReparseBuffer;

struct {

USHORT SubstituteNameOffset;

USHORT SubstituteNameLength;

USHORT PrintNameOffset;

USHORT PrintNameLength;

WCHAR PathBuffer[1];

} MountPointReparseBuffer;

struct {

UCHAR DataBuffer[1];

} GenericReparseBuffer;

};

} REPARSE\_DATA\_BUFFER, \*PREPARSE\_DATA\_BUFFER;

1. Bộ đệm Mount Point

Bộ đệm là một dẫn xuất từ REPARSE\_DATA\_BUFFER, chỉ dùng cho Reparse tag có bit cao set = 1.

Chứa thông tin về reparse point gồm:

* Substitute name: là đường dẫn đối tượng trỏ tới. Phải có tiền tố “\??\” phía trước.
* Print name: đường dẫn dùng hiển thị cho user

Cả hai đường dẫn không dùng thư mục “.” Và “..”, phải có (\) ở cuối.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **1 0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **2 0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **3 0** | **1** |
| **ReparseTag (4 bytes):**Số nguyên qui định loại tag, cho biết nhà phát triển của reparse point. Giá trị phải là: 0xA0000003. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **ReparseDataLezngth (2 bytes):**Kích thước byte của reparse data.  Tính từ thành viên **SubstituteNameOffset** tới **PathBuffer** | | | | | | | | | | | | | | | | **Reserved (2 bytes):**Không dùng, để 0. | | | | | | | | | | | | | | | |
| **SubstituteNameOffset (2 bytes):**Offset byte của Substitute name trong mảng **PathBuffer** , offset đi từ zero.  Chia cho 2 để lấy index trong **PathBuffer** | | | | | | | | | | | | | | | | **SubstituteNameLength (2 bytes):** Độ dài theo byte của Substitute name, giá trị này sẽ không gồm kí tự Unicode ‘\0’ nếu chuỗi có. | | | | | | | | | | | | | | | |
| **PrintNameOffset (2 bytes):** Offset byte của print name trong mảng **PathBuffer,** offset đi từ zero.  Chia cho 2 để lấy index trong **PathBuffer** | | | | | | | | | | | | | | | | **PrintNameLength (2 bytes):**Độ dài theo byte của print name, giá trị này sẽ không gồm kí tự Unicode ‘\0’ nếu chuỗi có. | | | | | | | | | | | | | | | |
| **PathBuffer (variable):**Mảng kí tự Unicode chứa Substitute name và print name, 2 đường dẫn có thể xuất hiện ngẫu nhiên trong mảng. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| ... | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

typedef struct {

DWORD ReparseTag;

DWORD ReparseDataLength;

WORD Reserved;

WORD ReparseTargetLength;

WORD ReparseTargetMaximumLength;

WORD Reserved1;

WCHAR ReparseTarget[1];

} REPARSE\_MOUNTPOINT\_DATA\_BUFFER, \*PREPARSE\_MOUNTPOINT\_DATA\_BUFFER;

1. Bộ đệm Symbolic Link

Là bộ đệm dẫn xuất từ REPARSE\_DATA\_BUFFER, chứa thông tin symbolic link reparse point, chỉ dùng cho reparse tag có giá trị high bit = 1.

Chứa thông tin về reparse point gồm:

* Substitute name: là đường dẫn đối tượng trỏ tới
* Print name: đường dẫn dùng hiển thị cho user

Cả hai đường dẫn không dùng thư mục “.” Và “..”, phải có (\) ở cuối.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **1 0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **2 0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **3 0** | **1** |
| **ReparseTag (4 bytes):**Số nguyên qui định loại tag, cho biết nhà phát triển của reparse point . Giá trị phải là xA000000C. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **ReparseDataLength (2 bytes):**Kích thước byte của reparse data.  Tính từ thành viên **SubstituteNameOffset** tới **PathBuffer** | | | | | | | | | | | | | | | | **Reserved (2 bytes):**Không dùng, để 0. | | | | | | | | | | | | | | | |
| **SubstituteNameOffset (2 bytes):**Offset byte của Substitute name trong mảng **PathBuffer** , offset đi từ zero.  Chia cho 2 để lấy index trong **PathBuffer** | | | | | | | | | | | | | | | | **SubstituteNameLength (2 bytes):** Độ dài theo byte của Substitute name, giá trị này sẽ không gồm kí tự Unicode ‘\0’ nếu chuỗi có. | | | | | | | | | | | | | | | |
| **PrintNameOffset (2 bytes):** Offset byte của print name trong mảng **PathBuffer,** offset đi từ zero.  Chia cho 2 để lấy index trong **PathBuffer** | | | | | | | | | | | | | | | | **PrintNameLength (2 bytes):**Độ dài theo byte của print name, giá trị này sẽ không gồm kí tự Unicode ‘\0’ nếu chuỗi có. | | | | | | | | | | | | | | | |
| **Flags (4 bytes):**chỉ rõ substitute name là đường dẫn đầy đủ hay đường dẫn liên quan tới thư mục chứa symbolic link | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **PathBuffer (variable):**Mảng kí tự Unicode chứa Substitute name và print name, 2 đường dẫn có thể xuất hiện ngẫu nhiên trong mảng. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| ... | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

Trường flag nhận 1 trong 2 giá trị sau:

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| 0x00000000 | The substitute name là đường dẫn đầy đủ. |
| SYMLINK\_FLAG\_RELATIVE  0x00000001 | The substitute name đường dẫn liên quan tới thư mục chứa symbolic link |

1. Bộ đệm cho nhà phát triển

Là cấu trúc bộ đệm nhận dữ liệu về reparse point , có sử dụng GUID với Reparse tag tương ứng.

Chỉ được dùng với giá trị reparse tag có high bit = 0.

Reparse point GUIDs đươc đăng kí bởi nhà phát triển phần mềm (ISV). An ISV phải liên kết GUID tới mỗi reparse point tag đăng kí, và phải luôn dùng GUID đó với tag.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **1 0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **2 0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **3 0** | **1** |
| **ReparseTag (4 bytes):**Số nguyên qui định loại tag, cho biết nhà phát triển của [reparse point](https://msdn.microsoft.com/en-us/library/cc231989.aspx#gt_4fed0b53-5fc8-4818-886f-93d87f3035e1) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **ReparseDataLength (2 bytes):**Kích thước theo byte của reparse data trong thành viên **DataBuffer**. | | | | | | | | | | | | | | | | **Reserved (2 bytes):**Không dùng, để 0 | | | | | | | | | | | | | | | |
| **ReparseDataLength (2 bytes):**A 16-bit unsigned integer value containing the size, in bytes, of the reparse data in the **DataBuffer** member. | | | | | | | | | | | | | | | | **Reserved (2 bytes):**Nên để 0 cho client, Server nên bỏ qua. | | | | | | | | | | | | | | | |
| **ReparseGuid (16 bytes):**Mã số GUID của nhà phát triển đăng ký với Microsoft để dùng kèm với reparse point tag đăng ký . | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| ... | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| ... | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **DataBuffer (variable):**Dữ liệu trả về độc lập với file system . Nó được xử lí và trả về bởi nhà phát triển. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| ... | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

Cấu trúc REPARSE\_GUID\_DATA\_BUFFER được dùng cho nhà phát triển drive chứa reparse data. Mỗi reparse point đều chứa REPARSE\_GUID\_DATA\_BUFFER.

typedef struct \_REPARSE\_GUID\_DATA\_BUFFER {

DWORD ReparseTag;

WORD ReparseDataLength;

WORD Reserved;

GUID ReparseGuid;

struct {

BYTE DataBuffer[1];

} GenericReparseBuffer;

} REPARSE\_GUID\_DATA\_BUFFER, \*PREPARSE\_GUID\_DATA\_BUFFER;

1. Bộ đệm Junction Point
2. Reparse control code
3. FSCTL\_SET\_REPARSE\_POINT

Set reparse point cho thư mục hay file

BOOL DeviceIoControl( (HANDLE) hDevice, // handle to file or directory

FSCTL\_SET\_REPARSE\_POINT, // dwIoControlCode

(LPVOID) lpInBuffer, // input buffer

(DWORD) nInBufferSize, // size of input buffer

NULL, // lpOutBuffer

0, // nOutBufferSize

(LPDWORD) lpBytesReturned, // lpBytesReturned

(LPOVERLAPPED) lpOverlapped ); // OVERLAPPED structure

* lpInBuffer[in]: bộ đệm nhận dữ liệu reparse point trả về.
* nInBufferSize[in]: kích thước bộ đệm tính theo byte. Bằng kích thước header bộ đệm + kích thước người dùng định nghĩa.
* lpBytesReturned: kích thước dữ liệu trả về bộ đệm lpOutBuffer theo byte.  
  \_ Nếu lpOverlapped = NULL: lpBytesReturned không được NULL.  
  \_ Nếu lpOverlapped != NULL: lpBytesReturned có thể để NULL. Nếu lpBytesReturned!=NULL và tiến trình có trả về dữ liệu, lpBytesReturned vô nghĩa tới khi tiến trình overlapped hoàn tất. Dùng GetOverlappedResult để lấy số byte trả về.
* lpOverlapped: con trỏ OVERLAPPED

\_ Nếu **hDevice** không dùng FILE\_FLAG\_OVERLAPPED, để NULL  
\_ Nếu **hDevie** dùng FILE\_FLAG\_OVERLAPPED, phải dùng giá trị hợp lệ chứa event handle, nếu không hàm sẽ FALSE

Hàm chỉ trả về khi tiến trình hoàn tất hay xảy ra lỗi.

Với tiến trình overlapped, hàm sẽ trả về ngay lập tức, đối tượng event được báo khi tiến trình hoàn tất.

Hàm trả về TRUE nếu thành công, FALSE nếu thất bại. GetLastError để bug lỗi.

1. FSCTL\_GET\_REPARSE\_POINT

Truy vấn reparse point tương ứng với handle file, thư mục

BOOL DeviceIoControl(

(HANDLE) hDevice, // handle to file or directory

FSCTL\_GET\_REPARSE\_POINT, // dwIoControlCode

NULL, // lpInBuffer

0, // nInBufferSize

(LPVOID) lpOutBuffer, // output buffer

(DWORD) nOutBufferSize, // size of output buffer

(LPDWORD) lpBytesReturned, // number of bytes returned

(LPOVERLAPPED) lpOverlapped // OVERLAPPED structure

);

* lpOutBuffer: Bộ đệm nhận reparse point data.
* nOutBufferSize: kích thước bộ đệm tính theo byte. Tối đa MAXIMUM\_REPARSE\_DATA\_BUFFER\_SIZE.
* lpBytesReturned: kích thước dữ liệu trả về bộ đệm lpOutBuffer theo byte.  
  \_ Nếu lpOverlapped = NULL: lpBytesReturned không được NULL.  
  \_ Nếu lpOverlapped != NULL: lpBytesReturned có thể để NULL. Nếu lpBytesReturned!=NULL và tiến trình có trả về dữ liệu, lpBytesReturned vô nghĩa tới khi tiến trình overlapped hoàn tất. Dùng GetOverlappedResult để lấy số byte trả về.
* lpOverlapped: con trỏ OVERLAPPED

\_ Nếu **hDevice** không dùng FILE\_FLAG\_OVERLAPPED, để NULL  
\_ Nếu **hDevie** dùng FILE\_FLAG\_OVERLAPPED, phải dùng giá trị hợp lệ chứa event handle, nếu không hàm sẽ FALSE

Hàm chỉ trả về khi tiến trình hoàn tất hay xảy ra lỗi.

Với tiến trình overlapped, hàm sẽ trả về ngay lập tức, đối tượng event được báo khi tiến trình hoàn tất.

Hàm trả về TRUE nếu thành công, FALSE nếu thất bại hay đang đợi. GetLastError để bug lỗi.

1. FSCTL\_DELETE\_REPARSE\_POINT

Gỡ reparse point khỏi handle file, directory.

BOOL DeviceIoControl(

(HANDLE) hDevice, // handle to file or directory

FSCTL\_DELETE\_REPARSE\_POINT, // dwIoControlCode

(LPVOID) lpInBuffer, // input buffer

(DWORD) nInBufferSize, // size of input buffer

NULL, // lpOutBuffer

0, // nOutBufferSize

(LPDWORD) lpBytesReturned, // number of bytes returned

(LPOVERLAPPED) lpOverlapped // OVERLAPPED structure

);

* lpInBuffer: Bộ đệm phải có tag, hay ReparseGuid hợp lệ nếu là tag của nhà phát triển, ReparseDataLength phải bằng 0.
* nInBufferSize: Kích thước bộ đệm, tính theo byte. Tối đa MAXIMUM\_REPARSE\_DATA\_BUFFER\_SIZE.
* lpBytesReturned: kích thước dữ liệu trả về bộ đệm lpOutBuffer theo byte.  
  \_ Nếu lpOverlapped = NULL: lpBytesReturned không được NULL.  
  \_ Nếu lpOverlapped != NULL: lpBytesReturned có thể để NULL. Nếu lpBytesReturned!=NULL và tiến trình có trả về dữ liệu, lpBytesReturned vô nghĩa tới khi tiến trình overlapped hoàn tất. Dùng GetOverlappedResult để lấy số byte trả về.
* lpOverlapped: con trỏ OVERLAPPED

\_ Nếu **hDevice** không dùng FILE\_FLAG\_OVERLAPPED, để NULL  
\_ Nếu **hDevie** dùng FILE\_FLAG\_OVERLAPPED, phải dùng giá trị hợp lệ chứa event handle, nếu không hàm sẽ FALSE

Hàm chỉ trả về khi tiến trình hoàn tất hay xảy ra lỗi.

Với tiến trình overlapped, hàm sẽ trả về ngay lập tức, đối tượng event được báo khi tiến trình hoàn tất.

Hàm trả về TRUE nếu thành công, FALSE nếu thất bại hay đang đợi. GetLastError để bug lỗi.

### Quản lý Registry

1. Cơ bản Registry

Registry là cơ sở dữ liệu trung tâm, phân cấp chứa thông tin cấu hình của các services và ứng dụng. Chúng ta truy xuất thông qua các khóa registry, được sắp xếp giống như cây thư mục phân cấp.

Chúng ta không nên tùy chỉnh dữ liệu registry không thuộc ứng dụng đang quản lý, nếu xảy ra lỗi trên registry, hệ thống sẽ hoạt động không ổn định.

Trước khi tùy chỉnh, nên backup và phục hồi registry ở trạng thái tốt nhất khi xảy ra sự cố.

1. Cấu trúc Registry

Dữ liệu registry tổ chức theo dạng cây. Mỗi node gọi là khóa, mỗi khóa chứa khóa con hoặc dữ liệu.

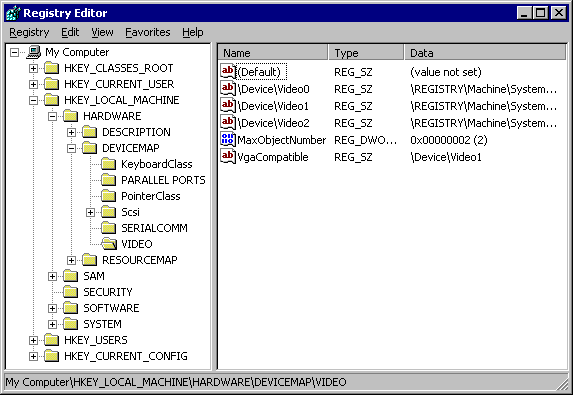
Đôi khi, ứng dụng chỉ cần sự tồn tại của khóa là được, hoặc sử dụng giá trị của khóa.

Giá trị của khóa có thể là bất kỳ dạng nào, tham khảo  [Registry Value Types](https://msdn.microsoft.com/en-us/library/windows/desktop/ms724884(v=vs.85).aspx) và [Registry Element Size Limits](https://msdn.microsoft.com/en-us/library/windows/desktop/ms724872(v=vs.85).aspx)

Tên của khóa không phân biệt hoa thường, không chứa ký tự (\), và là duy nhất.

Giá trị của khóa có tên và dữ liệu, tên và dữ liệu của giá trị có thể chứa ký tự (\).

Các giá trị đều có tên và dữ liệu mặc định



Các cây dưới My Computer đều là khóa. Khóa HKEY\_LOCAL\_MACHINE gồm các khóa con: HARDWARE, SAM, SECURITY, SOFTWARE, và SYSTEM. Các khóa con này đều chứa khóa con khác.

Khóa HARDWARE chứa các khóa con: DESCRIPTION, DEVICEMAP, và RESOURCEMAP

Mỗi giá trị chứa tên và dữ liệu của nó. MaxObjectNumber và VgaCompatible là các giá trị chứa dữ liệu nằm trong khóa VIDEO.

Cây registry có thể sâu tới 512 bậc. Chúng ta có thể tạo ra 32 bậc trong một lần gọi registry API.

Cây registry chứa các thông tin sau:

* Windows version, build number, và registered user. Tuy nhiên, ứng dụng thường dùng WinAPI để truy xuất thông tin này.
* Thông tin ứng dụng cài trên windows: version, build number…
* Thông tin loại chíp xử lí, số CPU, bộ nhớ,…
* Thông tin cấu hình của người dùng như home directory và cấu hình ứng dụng
* Thông tin bảo mật
* Các dịch vụ hệ thống và ứng dụng
* Ánh xạ từ phần mở rộng file tới ứng dụng thực thi: những ánh xạ này dùng bởi user interface shell khi người dùng click vào biểu tượng file, ví dụ file .doc, .docx ánh xạ tới MS Word.

1. Lưu trữ Registry

Theo khuyến cáo, các dữ liệu lớn hơn 1KB nên chứa thành file ở đâu đó và ứng dụng sử dụng registry của tham chiếu tới các file này.

Dữ liệu nhị phân file thực thi cũng không được chứa trong registry.

Các dữ liệu nên được nhóm lại theo cấu trúc và chứa trong giá trị theo dạng nhị phân, hơn là phân chia cấu trúc thành nhiều khóa riêng biệt.

Registry files được ánh xạ từ vùng nhớ hệ thống (paged pool memory). Đo đó, 1 dữ liệu registry không được vượt quá 4-MB

Windows Server 2008 for 32-bit, Windows Vista SP1 32-bit, Windows Vista, Windows Server 2003, Windows XP: registry files được ánh xạ từ cache address space.

Kích thước tối đa 1 registry hive là 2-GB, ngoại trừ system hive. Giới hạn kích thước registry hive 2-GB bắt đầu từ Windows Server 2003 (SP2).

Windows Server 2003 - SP1, Windows Server 2003 và Windows XP: không giới hạn vùng nhớ của registry hives trong paged pool memory và đĩa cứng, dù hệ thống có phân ngạch vùng nhớ tối đa .

|  |  |
| --- | --- |
| **System** | **Kích thước tối đa của system hive** |
| Hệ x86 | 50 % physical memory, tối đa 400 MB.  **Windows Server 2003 with SP2, Windows Server 2003 with SP1, Windows Server 2003 and Windows XP:**25 percent of physical memory, up to 200 MB. |
| Hệ x64 | 50 % physical memory, tối đa 1.5 GB.  **Windows Server 2003 with SP2:**25 % system memory, tối đa 200 MB.  **Windows Server 2003 with SP1, Windows Server 2003 và Windows XP 64-Bit Edition:**32 MB. |
| Intel Itanium-based systems | 50 % physical memory, tối đa 1 GB.  **Windows Server 2008, Windows Vista, Windows Server 2003 with SP2, Windows Server 2003 SP1, Windows Server 2003 và Windows XP 64-Bit Edition:**32 MB. |

1. Các khóa gốc

Ứng dụng phải mở khóa trước khi truy xuất dữ liệu vào registry. Để mở khóa, phải có handle tới khóa.

Hệ thống định nghĩa sẵn các khóa gốc, các khóa này giúp ứng dụng dò đường, hay giúp nhà quản trị hệ thống quản lý registry.

Các handle registry khá đang dạng, tùy theo platform hệ điều hành mà có hay không có các handle riêng biệt

Chúng ta truy xuất registry thông qua khóa và tên của giá trị. Có một vài khóa định nghĩa sẵn :

|  |  |
| --- | --- |
| **Handle** | **Mô tả** |
| HKEY\_LOCAL\_MACHINE | * Thông tin về trạng thái phần cứng như bus, system memory, các phần ứng và phần mềm cài đặt trên máy. * Dữ liệu cấu hình, thông tin Plug and Play (danh sách tất cả phần cứng đã cài). * Thông tin đăng nhập network, bảo mật network. * Thông tin hệ thống |
| HKEY\_USERS | * Thông tin cấu hình mặc định cho user mới trên local computer * Thông tin cấu hình của user trên máy |
| HKEY\_CURRENT\_CONFIG | * Thông tin về profile phần cứng hiện hành trên local computer. * Các thông tin này mô tả sự khác biệt cấu hình phần cứng hiện hành so với cấu hình chuẩn. * Thông tin cấu hình chuẩn chứa dưới khóa hệ thống và phần mềm HKEY\_LOCAL\_MACHINE. * HKEY\_CURRENT\_CONFIG chỉ là bí danh cho khóa HKEY\_LOCAL\_MACHINE\System\CurrentControlSet\HardwareProfiles\Current |
| HKEY\_CLASSES\_ROOT | * Thông tin ánh xạ phần mở rộng file tới ứng dụng. Tất cả khóa đều dùng bởi Microsoft’s Component Object Model (COM). * Không nên dùng trong service hay ứng dụng chạy độc lập với user |
| HKEY\_CURRENT\_USER | * Thông tin cấu hình của user hiện tại, bao gồm các biến môi trường, dữ liệu về program groups, màu, máy in, kết nối mạng, cấu hình ứng dụng. * Khóa này là bí danh một nhánh từ khóa HKEY\_USERS, giúp cấu hình dễ dàng thông tin user hiện hành. * Trong HKEY\_CURRENT\_USER chứa thông tin nhà phát triển ứng dụng và cấu hình dùng trong ứng dụng. Ví dụ đối với Microsoft, liên quan tới khóa HKEY\_CURRENT\_USER\Software\Microsoft. Với mỗi ứng dụng của Microsoft đều có khóa con nằm dưới * Việc ánh xạ tới HKEY\_USERS xảy ra khi có tham chiếu bởi process tới HKEY\_CURRENT\_USER.  Quá trình ánh xạ lệ thuộc vào ngữ cảnh bảo mật từ thread đầu tiên tham chiếu tới. Nếu trạng thái bảo mật không yêu cầu nạp registry hive từ HKEY\_USER, ánh xạ sẽ được thực hiện với HKEY\_USERS\.Default * Tất cả entry registry trong HKEY\_CURRENT\_USER, ngoại trừ HKEY\_CURRENT\_USER\Software\Classes, được nạp vào registry cho từng roaming user profile tương ứng. Các entry từ roaming user profile khác, được chứa trong HKEY\_CURRENT\_USER\_LOCAL\_SETTINGS * Không nên dùng trong service hay ứng dụng chạy độc lập với user. Thay vào đó, dùng phương thức RegOpenCurrentUser |

RegOverridePredefKey cho phép ánh xạ khóa gốc tới khóa chi tiết trong registry, ví dụ một ứng dụng cài đặt phải ánh xạ lại khóa gốc trước khi cài đặt thành phần DLL. Điều này cho phép trình cài đặt dễ dàng kiểm tra trạng thái thủ tục cài DLL tới đâu.

RegDisablePredefinedCache và RegDisablePredefinedCacheEx để vô hiệu hóa caching cho handle khóa gốc. Các services impersonation nên dùng RegDisablePredefinedCacheEx trước khi sử dụng handle khóa gốc.

Handle khóa gốc không đảm bảo thread safe, khi thread đóng handle khóa gốc sẽ ảnh hưởng đến thread khác đang dùng handle.

1. Registry hive

Registry hive: là một nhóm khóa, khóa con, và giá trị, được sao lưu trong 1 nhóm file gọi là hive file

Windows lưu các phần registry trong các hive file khác nhau. Khi hiển thị, các hives được gộp lại thành một unit logic.

Hive file có thể backups, export và import sang máy khác như file REG, nhưng chỉ có thể xem và tùy chỉnh thông qua Regedit mà thôi.

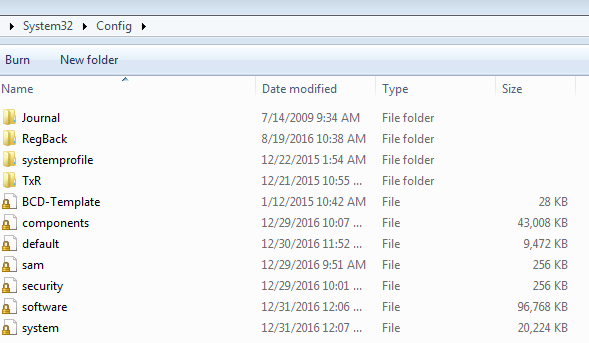
User profile hive nằm dưới khóa HKEY\_USER.

Các registry file REG và hive có 2 định dạng:

* Định dạng chuẩn: chỉ có trên Windows 2000. Vẫn được hỗ trợ trên phiên bản Window mới hơn
* Định dạng mới: có trên Window XP về sau.

Trên phiên bản Window mới, các hive sau vẫn sử dụng định dạng chuẩn: HKEY\_CURRENT\_USER, HKEY\_LOCAL\_MACHINE\SAM, HKEY\_LOCAL\_MACHINE\Security, và HKEY\_USERS\.DEFAULT

Các hive còn lại dùng định dạng mới

Hầu hết các user profile hive file nằm trong %SystemRoot%\System32\Config

Các file được cập nhật mỗi lần user đăng nhập, và một số không có phần mở rộng.

Bảng sau liệt kê các phần mở rộng cùng với mô tả dữ liệu của file.

|  |  |
| --- | --- |
| **Phần mở rộng** | **Mô tả** |
| Không có | Là bản sao hoàn chỉnh 1 hive. |
| .alt | Bản sao lưu của hive **HKEY\_LOCAL\_MACHINE\System** . Chỉ có khóa hệ thống mới có phần mở rộng .alt |
| .log | Bản ghi tiểu sử các phiên làm việc của khóa và giá trị trong hive. |
| .sav | Bản sao lưu của hive  **Windows Server 2003 và Windows XP/2000:**Bản sao của hive files as they looked at the end of the text-mode stage in Setup.  Setup has two stages: text mode and graphics mode. The hive is copied to a .sav file after the text-mode stage of setup to protect it from errors that might occur if the graphics-mode stage of setup fails.  If setup fails during the graphics-mode stage, only the graphics-mode stage is repeated when the computer is restarted; the .sav file is used to restore the hive data. |

Bảng liệt kê hive file và registry tương ứng

|  |  |
| --- | --- |
| **Registry hive** | **Hive Files** |
| **HKEY\_CURRENT\_CONFIG** | System, System.alt, System.log, System.sav |
| **HKEY\_CURRENT\_USER** | Ntuser.dat, Ntuser.dat.log |
| **HKEY\_LOCAL\_MACHINE\SAM** | Sam, Sam.log, Sam.sav |
| **HKEY\_LOCAL\_MACHINE\Security** | Security, Security.log, Security.sav |
| **HKEY\_LOCAL\_MACHINE\Software** | Software, Software.log, Software.sav |
| **HKEY\_LOCAL\_MACHINE\System** | System, System.alt, System.log, System.sav |
| **HKEY\_USERS\.DEFAULT** | Default, Default.log, Default.sav |

1. Thông tin ứng dụng

Trước khi đưa thông tin vào registry, ứng dụng chia thông tin ra làm 2 loại: máy và người dùng.

Căn cứ vào 2 đặc điểm trên, ứng dụng có thể có cấu hình riêng biệt cho nhiều user, nhận biết được user qua mạng và dùng dữ liệu ở nhiều địa điểm khác nhau, cho phép dữ liệu user profile không lệ thuộc bởi vị trí địa lý.

Khi ứng dụng được cài đặt, nên để lại thông tin liên quan trong khóa HKEY\_LOCAL\_MACHINE như công ty phát triển, tên sản phẩm, số phiên bản. Ví dụ:

HKEY\_LOCAL\_MACHINE\Software\MyCompany\MyProduct\1.0

Nếu ứng dụng hỗ trợ COM, nên để lại thông tin trong khóa HKEY\_LOCAL\_MACHINE\Software\Classes.

Cuối cùng là ghi lại thông tin cấu hình của user trong khóa HKEY\_CURRENT\_USER, ví dụ:

HKEY\_CURRENT\_USER\Software\MyCompany\MyProduct\1.0

1. Độ dài trong registry

|  |  |
| --- | --- |
| **Yếu tố Registry** | **Kích thước tối đa** |
| Tên khóa | 255 kí tự. Là đường dẫn đầy đủ trong registry.  Luôn bắt đầu từ khóa cơ bản, ví dụ: HKEY\_LOCAL\_MACHINE. |
| Tên giá trị | 16,383 kí tự  **Windows 2000:**260 kí tự ANSI hay 16,383 kí tự Unicode. |
| Giá trị | Tùy theo bộ nhớ còn trống (định dạng mới)  1 MB (định dạng chuẩn) |
| Cây registry | Tối đa 512 bậc. Các API chỉ tạo tối đa 32 bậc cho 1 lần gọi. |

 Các giá trị có kích thước lớn hơn 2-MB nên lưu sang file, để tối ưu hiệu suất.

Đường dẫn file có thể đặt làm tên của giá trị hoặc dữ liệu của giá trị. Phải double backslash “\\” cho chuỗi, ví dụ: C:\\mydir\\myfile"

1. Kiểu giá trị

Khi tạo giá trị bằng RegSetValueEx, chúng ta phải chỉ rõ kiểu dữ liệu của giá trị. Cũng như khi lấy giá trị registry bằng RegQueryValueEx, phải dùng các marco để thông báo kiểu dữ liệu

Kiểu dữ liệu định nghĩa trong thư viện Winnt.h.

|  |  |
| --- | --- |
| **Kiểu** | **Mô tả** |
| REG\_BINARY | Giá trị nhị phân. |
| REG\_DWORD | Số nguyên 32-bit. |
| REG\_DWORD\_LITTLE\_ENDIAN | Số nguyên 32-bit dạng little-endian.  Mặc định Windows thiết kế trên little-endian nên kiểu này cũng là REG\_DWORD |
| REG\_DWORD\_BIG\_ENDIAN | Số nguyên 32-bit dạng big-endian. Một số hệ UNIX dùng dạng này |
| REG\_EXPAND\_SZ | Chuỗi có byte kết zero.  Chứa biến môi trường rút gọn (ví dụ “%PATH%”).  Mã hóa Unicode hay ANSI tùy vào hàm bạn dùng.  Dùng [**ExpandEnvironmentStrings**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms724265(v=vs.85).aspx) để lấy đường dẫn đầy đủ. |
| REG\_LINK | Chuỗi Unicode có byte kết zero.  Chứa đường dẫn của liên kết symbolic  Giá trị này được tạo bởi [**RegCreateKeyEx**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms724844(v=vs.85).aspx)  với REG\_OPTION\_CREATE\_LINK |
| REG\_MULTI\_SZ | Gồm nhiều chuỗi kết thúc bởi ‘\0’.  Ví dụ: *String1*\0*String2*\0*String3*\0*LastString*\0\0  (\0) được đặt ở cuối, báo hiệu kết thúc. |
| REG\_NONE | Không có kiểu xác định |
| REG\_QWORD | Số nguyên 64-bits |
| REG\_QWORD\_LITTLE\_ENDIAN | Số nguyên 64-bit dạng little-endian.  Mặc định Windows thiết kế trên little-endian nên kiểu này cũng là REG\_ QWORD |
| REG\_SZ | Chuỗi có byte kết zero.  Mã hóa Unicode hay ANSI, tùy vào hàm sử dụng |

Đối với các kiểu chuỗi như REG\_SZ, REG\_MULTI\_SZ, hay REG\_EXPAND\_SZ, dữ liệu sẽ không có ‘\0’, khi đọc chuỗi từ registry phải chèn vào cuối trước khi dùng. (Lưu ý: REG\_MULTI\_SZ khi chèn vào sẽ có double ‘\0’).

Khi ghi chuỗi vào registry, phải chắc chắn số ký tự ghi vào, bao gồm cả ‘\0’. Các lỗi phổ biến nhất là dùng lstrlen để xác định độ dài chuỗi, nó không tính đến ký tự ‘\0’. Do đó công thức tính nên là: lstrlen() + 1.

A REG\_MULTI\_SZ kết thúc với chuỗi có độ dài zero. Do đó, không thể chèn chuỗi như vậy vào cuối, chuỗi như vậy định nghĩa là \0

1. Truy xuất khóa

Các API liên quan registry nằm trong thư viện <Winreg.h>, được inlcude sẵn trong <Windows.h>

1. Nguyên tắc

Ứng dụng cần handle khóa để truy xuất dữ liệu registry, RegOpenKeyEx để mở khóa và RegCreateKeyEx để tạo khóa. Khóa mới tạo ra được coi như khóa con của khóa các khóa đang mở.

Các khóa gốc sau luôn luôn mở: HKEY\_LOCAL\_MACHINE, HKEY\_CLASSES\_ROOT, HKEY\_USERS, và HKEY\_CURRENT\_USER

Lứu ý: Ứng dụng không được phép tạo khóa con trực tiếp của khóa HKEY\_USERS hay HKEY\_LOCAL\_MACHINE, chỉ được phép tạo khóa con cho các cấp thấp hơn mà thôi

Không có cơ chế lock một khóa trong khi có tiến trình ghi dữ liệu vào khóa. Tuy nhiên, có thể kiểm soát việc truy xuất khóa bằng cách dùng thuộc tính bảo mật của khóa

Để kết hợp khóa với một phiên làm việc, có thể dùng RegCreateKeyTransacted hay RegOpenKeyTransacted. Tham khảo thêm Kernel Transaction Manager

1. Tham số bộ đệm

Trong nhiều API Registry sẽ sử dụng nhiều tham số bộ đệm để nhận thông tin trả về như: tên khóa, tên giá trị, dữ liệu của giá trị hay chuỗi mô tả phân loại khóa do người dùng tự định nghĩa.

Mỗi tham số bộ đệm luôn theo sau là tham số chỉ rõ kích thước của bộ đệm, tùy theo thông tin trả về mà tham số kích thước tính theo số ký tự (tên khóa, tên giá trị, dữ liệu chuỗi) hay tính theo số byte (dữ liệu số, nhị phân).

Đối với bộ đệm nhận dữ liệu kiểu chuỗi, phải set ‘\0’ vào cuối trước khi dùng.

Có hai cách để xác định kích thước bộ đệm:

1. Dùng RegQueryInfoKey để biết kích thước tối đa cho tên giá trị và dữ liệu.
2. Để NULL cho tham số bộ đệm, nhưng tham số kích thước phải hợp lệ. Hàm sẽ trả về ERROR\_MORE\_DATA, đồng thời hàm cho biết kích thước bộ đệm cần thiết trong lpcchValueName và lpcbData.
3. RegOpenKeyEx

RegOpenKeyEx chỉ mở khóa, không tạo khóa nếu không tồn tại.

LONG WINAPI RegOpenKeyEx(

\_In\_ HKEY hKey,

\_In\_opt\_ LPCTSTR lpSubKey,

\_In\_ DWORD ulOptions,

\_In\_ REGSAM samDesired,

\_Out\_ PHKEY phkResult

);

* hKey [in]: Handle khóa đang mở. hKey trả về từ RegCreateKeyEx hay RegOpenKeyEx, hoặc 1 trong những khóa gốc: HKEY\_CLASSES\_ROOT, HKEY\_CURRENT\_CONFIG, HKEY\_CURRENT\_USER, HKEY\_LOCAL\_MACHINE, HKEY\_USERS
* lpSubKey [in, optional]: Tên khóa cần mở, tên không phân biệt hoa thường.

Trường hợp lpSubKey rỗng và hKey = HKEY\_CLASSES\_ROOT: phkResult = HKEY\_CLASSES\_ROOT.

Trường hợp lpSubKey rỗng và hKey khác HKEY\_CLASSES\_ROOT: phkResult là Handle khóa mới tham chiếu tới hKey

lpSubKey chỉ để NULL nếu hKey là khóa gốc.

Trường hợp lpSubKey = NULL và hKey = HKEY\_CLASSES\_ROOT: phkResult là Handle khóa mới tham chiếu tới HKEY\_CLASSES\_ROOT

Trường hợp lpSubKey = NULL và hKey khác HKEY\_CLASSES\_ROOT: phkResult = hKey

* ulOptions [in]: Set là zero. Nếu set REG\_OPTION\_OPEN\_LINK, thì khóa là liên kết symbolic. Liên kết symbolic trong registry chỉ nên dùng trong trường hợp cần thiết.
* samDesired [in]: Khai báo quyền truy xuất của khóa được mở. Một số Macro hay dùng:

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| KEY\_ALL\_ACCESS: | Kết hợp của STANDARD\_RIGHTS\_REQUIRED, KEY\_QUERY\_VALUE, KEY\_SET\_VALUE, KEY\_CREATE\_SUB\_KEY, KEY\_ENUMERATE\_SUB\_KEYS, KEY\_NOTIFY, và KEY\_CREATE\_LINK |
| KEY\_WRITE | Kết hợp của STANDARD\_RIGHTS\_WRITE, KEY\_SET\_VALUE, and KEY\_CREATE\_SUB\_KEY |
| KEY\_READ | Kết hợp của STANDARD\_RIGHTS\_READ, KEY\_QUERY\_VALUE, KEY\_ENUMERATE\_SUB\_KEYS, và KEY\_NOTIFY |
| KEY\_QUERY\_VALUE | Quyền truy vấn dữ liệu của giá trị |
| KEY\_ENUMERATE\_SUBKEYS | Quyền truy vấn khóa con |

* phkResult [out]: Con trỏ tới Handle khóa nhận kết quả trả về.

Hàm trả về ERROR\_SUCCESS nếu thành công, ngược lại trả về giá trị khác zero (giá trị được định nghĩa trong Winerror.h)

Hàm cũng sẽ thất bại nếu security descriptor của khóa không cho phép truy xuất đối với process hiện tại.

Các truy xuất khóa registry chịu ảnh hưởng bởi security descriptor. Dù khóa được mở với chế độ samDesired là KEY\_READ, nó vẫn có thể tạo khóa nếu security descriptor của khóa cho phép.

Nếu ứng dụng hay services phục vụ nhiều user (impersonation), không được dùng RegOpenKeyEx với HKEY\_CURRENT\_USER. Thay vào đó, dùng RegOpenCurrentUser.

1. RegCreateKeyEx

RegCreateKeyEx tạo khóa registry nếu không tồn tại, nếu khóa đã có thì chỉ mở.

LONG WINAPI RegCreateKeyEx(

\_In\_ HKEY hKey,

\_In\_ LPCTSTR lpSubKey,

\_Reserved\_ DWORD Reserved,

\_In\_opt\_ LPTSTR lpClass,

\_In\_ DWORD dwOptions,

\_In\_ REGSAM samDesired,

\_In\_opt\_ LPSECURITY\_ATTRIBUTES lpSecurityAttributes,

\_Out\_ PHKEY phkResult,

\_Out\_opt\_ LPDWORD lpdwDisposition

);

* hKey [in]: Handle khóa đang mở, phải có quyền KEY\_CREATE\_SUB\_KEY. Tuy nhiên nếu security descriptor của hKey cho phép, vẫn có thể tạo khóa dù không có quyền KEY\_CREATE\_SUB\_KEY.  
  hKey trả về từ RegCreateKeyEx hay RegOpenKeyEx hoặc: HKEY\_CLASSES\_ROOT, HKEY\_CURRENT\_CONFIG, HKEY\_CURRENT\_USER, HKEY\_LOCAL\_MACHINE, HKEY\_USERS
* lpSubKey [in]: Không được NULL. Tên khóa con để mở hay tạo, hệ điều hành sẽ dò khóa trong vòng 32 bậc registry cho bạn.

Nếu lpSubKey rỗng, phkResult sẽ là handle khóa mới tham chiếu tới hKey.

* Reserved: Để zero.
* lpClass [in, optional]: Người dùng tự phân loại cho khóa, nên để NULL.
* dwOptions [in]: gồm các tùy chọn sau:

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| **REG\_OPTION\_BACKUP\_RESTORE**  0x00000004L | If this flag is set, the function ignores the samDesired parameter and attempts to open the key with the access required to backup or restore the key. If the calling thread has the SE\_BACKUP\_NAME privilege enabled, the key is opened with the ACCESS\_SYSTEM\_SECURITY and KEY\_READ access rights. If the calling thread has the SE\_RESTORE\_NAME privilege enabled, beginning with Windows Vista, the key is opened with the ACCESS\_SYSTEM\_SECURITY, DELETE and KEY\_WRITE access rights. If both privileges are enabled, the key has the combined access rights for both privileges. For more information, see [Running with Special Privileges](https://msdn.microsoft.com/en-us/library/windows/desktop/ms717802(v=vs.85).aspx). |
| **REG\_OPTION\_CREATE\_LINK**  0x00000002L | **Note**  Registry symbolic links should only be used for for application compatibility when absolutely necessary.    This key is a symbolic link. The target path is assigned to the L"SymbolicLinkValue" value of the key. The target path must be an absolute registry path. |
| **REG\_OPTION\_NON\_VOLATILE**  0x00000000L | Giá trị mặc định, khóa cố định. Thông tin khóa lưu trữ trong file và được duy trì khi hệ thống khởi động lại.  [**RegSaveKey**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms724917(v=vs.85).aspx) chỉ làm việc với khóa non-volatile. |
| **REG\_OPTION\_VOLATILE**  0x00000001L | Khóa không cố định. Thông tin khóa chỉ nằm trên vùng nhớ RAM và không được duy trì khi registry hive tương ứng không được nạp.  Đối với **HKEY\_LOCAL\_MACHINE**, this occurs only when the system initiates a full shutdown. For registry keys loaded by the [**RegLoadKey**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms724889(v=vs.85).aspx) function, this occurs when the corresponding[**RegUnLoadKey**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms724924(v=vs.85).aspx) is performed. The [**RegSaveKey**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms724917(v=vs.85).aspx) function does not save volatile keys. This flag is ignored for keys that already exist.  **Note**  On a user selected shutdown, a fast startup shutdown is the default behavior for the system. |

* samDesired [in]: qui định quyền truy xuất của khóa được tạo hay mở
* lpSecurityAttributes [in, optional]: con trỏ SECURITY\_ATTRIBUTES. Nếu là NULL, handle khóa sẽ không thừa kế thuộc tính bảo mật từ khóa cha và có security descriptor mặc định.

security descriptor khóa mới qui định bởi thành viên lpSecurityDescriptor. ACLs trong security descriptor mặc định của khóa được thừa kế từ khóa cha.

* phkResult [out]: con trỏ Handle khóa nhận kết quả trả về cho khóa mới tạo hay mở.
* lpdwDisposition [out, optional]: Cho biết thông tin về khóa , nhận 2 giá trị sau  
  REG\_CREATED\_NEW\_KEY (0x00000001L) : Khóa không tồn tại và được tạo mới.  
  REG\_OPENED\_EXISTING\_KEY (0x00000002L) : Khóa tồn tại và được mở.   
  Nếu lpdwDisposition = NULL, đơn giản là không có thông tin trả về, không có lỗi gì cả.

Hàm trả về ERROR\_SUCCESS nếu thành công, ngược lại trả về giá trị khác zero (giá trị được định nghĩa trong Winerror.h)

Lưu ý: RegCreateKeyEx sẽ tạo tất cả khóa trên đường dẫn lpSubKey nếu chúng không tồn tại. Ứng dụng có thể tạo nhiều khóa cho một lần gọi hàm

Ví dụ: subkey1\subkey2\subkey3\subkey4

Tuy nhiên, đường dẫn khóa phải chính xác để tránh tạo ra khóa rác trong registry

Nếu ứng dụng hay services phục vụ nhiều user (impersonation), không được dùng RegOpenKeyEx với HKEY\_CURRENT\_USER. Thay vào đó, dùng RegOpenCurrentUser.

1. RegQueryInfoKey

Truy vấn thông tin về khóa registry

LONG WINAPI RegQueryInfoKey(

\_In\_ HKEY hKey,

\_Out\_opt\_ LPTSTR lpClass,

\_Inout\_opt\_ LPDWORD lpcClass,

\_Reserved\_ LPDWORD lpReserved,

\_Out\_opt\_ LPDWORD lpcSubKeys,

\_Out\_opt\_ LPDWORD lpcMaxSubKeyLen,

\_Out\_opt\_ LPDWORD lpcMaxClassLen,

\_Out\_opt\_ LPDWORD lpcValues,

\_Out\_opt\_ LPDWORD lpcMaxValueNameLen,

\_Out\_opt\_ LPDWORD lpcMaxValueLen,

\_Out\_opt\_ LPDWORD lpcbSecurityDescriptor,

\_Out\_opt\_ PFILETIME lpftLastWriteTime

);

* hKey [in]: Handle khóa đang mở có quyền KEY\_QUERY\_VALUE. hKey trả về từ RegCreateKeyEx hay RegOpenKeyEx hoặc: HKEY\_CLASSES\_ROOT, HKEY\_CURRENT\_CONFIG, HKEY\_CURRENT\_USER, HKEY\_LOCAL\_MACHINE, HKEY\_USERS
* lpClass [in, out]: bộ đệm nhận thông tin người dùng phân loại cho hKey, có thể NULL
* lpcClass [in, out, optional]: kích thước tính theo ký tự của lpClass. Khi hàm trả về thành công, lpcName sẽ là số ký tự ghi vào bộ đệm lpClass, không bao gồm ‘\0’. Để NULL nếu lpClass là NULL.
* lpReserved: Để NULL
* lpcSubKeys [out, optional]: Nhận thông tin số khóa con. Để NULL nếu bỏ qua
* lpcMaxSubKeyLen [out, optional]: Nhận thông tin số ký tự Unicode dài nhất của tên khóa con, không tính ‘\0’. Để NULL nếu bỏ qua
* lpcMaxClassLen [out, optional]: Nhận thông tin số ký tự Unicode dài nhất của chuỗi phân loại do người dùng tự định nghĩa cho khóa con, không tính ‘\0’. Để NULL nếu bỏ qua
* lpcValues [out, optional]: Nhận thông tin số giá trị có trong khóa. Để NULL nếu bỏ qua
* lpcMaxValueNameLen [out, optional]: Nhận thông tin số ký tự Unicode dài nhất của tên giá trị, không tính ‘\0’. Để NULL nếu bỏ qua
* lpcMaxValueLen [out, optional]: Nhận thông tin số byte của dữ liệu lớn nhất. Để NULL nếu bỏ qua
* lpcbSecurityDescriptor [out, optional]: kích thước security descriptor của khóa, tính theo byte. Để NULL nếu bỏ qua
* lpftLastWriteTime [out, optional]: Con trỏ FILETIME chỉ thời gian lần ghi khóa cuối cùng. Để NULL nếu bỏ qua

Hàm trả về:

ERROR\_SUCCESS : nếu thành công, hoặc error code nếu bị lỗi.

ERROR\_MORE\_DATA: khi bộ đệm không đủ lớn cho thông tin trả về

ERROR\_INVALID\_PARAMETER: khi bộ đệm hợp lệ nhưng tham số kích thước bộ đệm không hợp lệ

1. RegDeleteKey

RegDeleteKey xóa khóa và các giá trị của nó

LONG WINAPI RegDeleteKey(

\_In\_ HKEY hKey,

\_In\_ LPCTSTR lpSubKey

);

* hKey [in]: Handle khóa đang mở, quyền truy xuất hKey không ảnh hưởng tới thao tác xóa

hKey trả về từ RegCreateKeyEx hay RegOpenKeyEx hoặc:

HKEY\_CLASSES\_ROOT, HKEY\_CURRENT\_CONFIG, HKEY\_CURRENT\_USER, HKEY\_LOCAL\_MACHINE, HKEY\_USERS

* lpSubKey [in]: Tên khóa bị xóa, phải là khóa con của hKey. Tham số này không được để NULL

RegDeleteKey mở lpSubKey với quyền DELETE.

Hàm trả về ERROR\_SUCCESS nếu thành công, ngược lại trả về error code

RegDeleteKey sẽ xóa luôn khóa sau khi tất cả handle liên quan đóng. Không thể tạo khóa con và giá trị cho khóa đã xóa

Khóa bị xóa không được có khóa con. Để xóa toàn bộ khóa và khóa con đệ qui dùng RegDeleteTree hay SHDeleteKey

1. RegEnumKeyEx

Liệt kê khóa con, sau mỗi lần gọi hàm lấy thông tin về của một khóa con.

LONG WINAPI RegEnumKeyEx(

\_In\_ HKEY hKey,

\_In\_ DWORD dwIndex,

\_Out\_ LPTSTR lpName,

\_Inout\_ LPDWORD lpcName,

\_Reserved\_ LPDWORD lpReserved,

\_Inout\_ LPTSTR lpClass,

\_Inout\_opt\_ LPDWORD lpcClass,

\_Out\_opt\_ PFILETIME lpftLastWriteTime

);

* hKey [in]: Handle khóa đang mở, phải có quyền KEY\_ENUMERATE\_SUB\_KEYS. Handle trả về từ RegCreateKeyEx hay RegOpenKeyEx hoặc: HKEY\_CLASSES\_ROOT, HKEY\_CURRENT\_CONFIG, HKEY\_CURRENT\_USER, HKEY\_LOCAL\_MACHINE, HKEY\_USERS
* dwIndex [in]: index của khóa con trả về. Các khóa trả về không theo thứ tự nào.
* lpName [out]: Bộ đệm nhận tên của khóa con, có ‘\0’. Không phải đường dẫn đầy đủ.
* lpcName [in, out]: Con trỏ kích thước của bộ đệm lpName, tính theo số ký tự. Khi hàm trả về thành công, lpcName sẽ là số ký tự ghi vào bộ đệm lpName, không bao gồm ‘\0’

Dùng RegQueryInfoKey để biết kích thước bộ đệm tối đa

* lpReserved: Để NULL
* lpClass [in, out]: bộ đệm nhận thông tin người dùng phân loại cho khóa con, nên để NULL
* lpcClass [in, out, optional]: kích thước tính theo ký tự của lpClass. Khi hàm trả về thành công, lpcName sẽ là số ký tự ghi vào bộ đệm lpClass, không bao gồm ‘\0’. Để NULL nếu lpClass là NULL.
* lpftLastWriteTime [out, optional]: Con trỏ FILETIME, nhận thông tin thời gian lần cuối khóa con bị ghi, có thể để NULL.

Giá trị trả về của hàm:

ERROR\_SUCCESS :nếu thành công, ngược lại trả về error code.

ERROR\_NO\_MORE\_ITEMS: Nếu không còn khóa con để trả về

ERROR\_MORE\_DATA: Nếu kích thước bộ đệm không đủ

Nguyên tắc sử dụng là dùng RegQueryInfoKey để lấy tổng số khóa con. Tăng dần hay giảm dần dwIndex cho mỗi lần gọi RegEnumKeyEx tới khi hàm trả về ERROR\_MORE\_DATA.

Khi đang dùng RegEnumKeyEx, không nên dùng các hàm registry khác, tránh ảnh hưởng đến khóa đang liệt kê.

1. RegCloseKey

Chúng ta chỉ gọi RegCloseKey cho các khóa không phải mặc định

LONG WINAPI RegCloseKey(

\_In\_ HKEY hKey

);

* hKey [in]: Handle khóa đang mở, trả về từ RegCreateKeyEx, RegOpenKeyEx, RegOpenKeyTransacted hay RegConnectRegistry

Hàm trả về ERROR\_SUCCESS nếu thành công, ngược lại trả về error code

Sau khi đóng, handle không nên dùng nữa. Handle khóa không nên để lâu hơn cần thiết.

RegCloseKey sẽ đóng khóa và ghi dữ liệu vào registry, RegCloseKey sẽ tốn thời gian sau khi nó trả về để flush dữ liệu vào đĩa cứng.

1. RegFlushKey

Ghi tất cả thông tin đã tùy chỉnh của khóa vào registry.

LONG WINAPI RegFlushKey(

\_In\_ HKEY hKey

);

* hKey [in]: Handle khóa đang mở, phải có quyền KEY\_QUERY\_VALUE

hKey trả về từ RegCreateKeyEx hay RegOpenKeyEx hoặc: HKEY\_CLASSES\_ROOT, HKEY\_CURRENT\_CONFIG, HKEY\_CURRENT\_USER, HKEY\_LOCAL\_MACHINE, HKEY\_USERS

Hàm trả về ERROR\_SUCCESS nếu thành công, ngược lại trả về error code

RegFlushKey sẽ ép hệ thống ghi dữ liệu registry vào đĩa cứng. Tuy nhiên, không nên lạm dụng RegFlushKey vì nó tốn tài nguyên hệ thống, nó còn ghi tất cả dữ liệu của khóa khác trong cùng 1 hive

RegFlushKey sẽ block tất cả thao tác điều chỉnh mọi process trong registry hive đang flush cho tới khi hoàn tất và trả về.

Tất cả điều chỉnh tới khóa registry đều thấy được tới mọi process mà không cần phải flush.

Windows dùng cơ chế “lazy flush”, chỉ flush registry vào đĩa cứng theo chu kỳ nhất định. Khi hệ thống tắt, registry cũng được flush. Cơ chế lazy flush hiện là cách quản lý hiệu quả.

Dùng RegCloseKey đóng handle sau khi RegFlushKey trả về .

1. Truy xuất giá trị registry

Dùng RegSetValueEx để set giá trị và dữ liệu cho khóa, nếu tên giá trị tham chiếu trong RegSetValueEx không tồn tại, hàm sẽ tạo ra giá trị mới.

Khóa registry không có tên mặc định và chỉ được phép có 1 giá trị có tên-dữ liệu mặc định.

Kích thước của giá trị bị giới hạn bởi bộ nhớ còn trống. Tuy nhiên, việc lưu giá trị lớn trong registry có thể ảnh hưởng tới hiệu suất.

Các yếu tố như biểu tượng, bitmaps, và file thực thi nên lưu trữ thành file, chứ không nên lưu vào registry.

1. RegSetValueEx

Set dữ liệu, tên và kiểu của giá trị trong khóa registry.

LONG WINAPI RegSetValueEx(

\_In\_ HKEY hKey,

\_In\_opt\_ LPCTSTR lpValueName,

\_Reserved\_ DWORD Reserved,

\_In\_ DWORD dwType,

\_In\_ const BYTE \*lpData,

\_In\_ DWORD cbData

);

* hKey [in]: Handle khóa đang mở, phải có quyền KEY\_SET\_VALUE. Handle trả về từ RegCreateKeyEx hay RegOpenKeyEx hoặc các macro: HKEY\_CLASSES\_ROOT, HKEY\_CURRENT\_CONFIG, HKEY\_CURRENT\_USER, HKEY\_LOCAL\_MACHINE, HKEY\_USERS

Bản Unicode của hàm còn hỗ trợ thêm các khóa gốc: HKEY\_PERFORMANCE\_TEXT và HKEY\_PERFORMANCE\_NLSTEXT

* lpValueName [in, optional]: Tên giá trị, nếu tên chưa có sẽ tạo ra giá trị với tên mới.  
  Nếu để NULL hay chuỗi rỗng, hàm sẽ tạo giá trị có tên và dữ liệu mặc định, có kiểu không xác định.
* Reserved: Để zero
* dwType [in]: Macro kiểu dữ liệu
* lpData [in]: bộ đệm chứa dữ liệu cho giá trị. Đối với dữ liệu chuỗi phải có ‘\0’.  
  Riêng kiểu REG\_MULTI\_SZ, phải có double kí tự ‘\0’, các chuỗi con bên trong phải dùng “\\” cho đường dẫn, ví dụ: "C:\\mydir\\myfile" cho "C:\mydir\myfile".

lpData có thể để NULL, trong trường hợp này, cbData phải là zero.

* cbData [in]: kích thước bộ đệm lpData, tính theo bytes. Nếu kiểu dữ liệu là dạng chuỗi như REG\_SZ, REG\_EXPAND\_SZ, hay REG\_MULTI\_SZ, phải tính luôn kích thước của kí tự ‘\0’.

Hàm trả về ERROR\_SUCCESS nếu thành công, ngược lại trả về error code

Đối với các kiểu chuỗi như REG\_SZ, REG\_MULTI\_SZ, và REG\_EXPAND\_SZ : Nếu dùng hàm ANSI RegSetValueExA hay không định nghĩa UNICODE trước Windows.h thì bộ đệm phải chứa chuỗi ANSI.

1. RegEnumValue

Liệt kê giá trị cho khóa đang mở, hàm trả về tên giá trị và dữ liệu sau mỗi lần gọi.

LONG WINAPI RegEnumValue(

\_In\_ HKEY hKey,

\_In\_ DWORD dwIndex,

\_Out\_ LPTSTR lpValueName,

\_Inout\_ LPDWORD lpcchValueName,

\_Reserved\_ LPDWORD lpReserved,

\_Out\_opt\_ LPDWORD lpType,

\_Out\_opt\_ LPBYTE lpData,

\_Inout\_opt\_ LPDWORD lpcbData

);

* hKey [in]: Handle khóa đang mở, phải có quyền KEY\_QUERY\_VALUE. Handle trả về từ RegCreateKeyEx hay RegOpenKeyEx hoặc các macro: HKEY\_CLASSES\_ROOT, HKEY\_CURRENT\_CONFIG, HKEY\_CURRENT\_USER, HKEY\_LOCAL\_MACHINE, HKEY\_USERS
* dwIndex [in]: index của giá trị trả về. Các giá trị trả về không theo thứ tự nào.
* lpValueName [out]: Bộ đệm nhận tên giá trị trả về, có ‘\0’. Phải đủ lớn để chứa kết quả.
* lpcchValueName [in, out]: Số ký tự trả về của lpValueName, không tính ‘\0’. Khi hàm trả về, lpcchValueName cho biết số ký tự ghi vào bộ đệm.

Tên giá trị tối đa 32,767 byte. Bản ANSI RegEnumValueA, tham số này kiểu SHORT.

* lpReserved: Để NULL
* lpType [out, optional]: Macro trả về cho biết kiểu của giá trị. Để NULL nếu bỏ qua.
* lpData [out, optional]: Bộ đệm nhận dữ liệu. Để NULL nếu bỏ qua
* lpcbData [in, out, optional]: Kích thước byte của bộ đệm lpData. Khi hàm trả về, lpcbData là số byte đã ghi vào bộ đệm. Để NULL nếu lpData là NULL.

Nếu dữ liệu là kiểu chuỗi REG\_SZ, REG\_MULTI\_SZ và REG\_EXPAND\_SZ, kích thước sẽ tính luôn ký tự ‘\0’.

Các giá trị trả về của hàm:

ERROR\_SUCCESS : Hàm gọi thành công, ngược lại trả về error code.

ERROR\_NO\_MORE\_ITEMS : Không còn giá trị để trả về

ERROR\_MORE\_DATA : Khi bộ đệm không đạt kích thước tối thiểu

Có thể dùng RegQueryInfoKey để lấy tổng số index, dùng vòng lặp gọi RegEnumValue tới khi trả về ERROR\_NO\_MORE\_ITEMS.

Không nên dùng API registry khác trong khi đang duyệt giá trị bằng RegEnumValue.

Có hai cách để xác định kích thước bộ đệm:

1. Dùng RegQueryInfoKey để biết kích thước tối đa cho tên giá trị và dữ liệu.
2. Để NULL cho bộ đệm, hàm sẽ trả về ERROR\_MORE\_DATA, đồng thời hàm cho biết kích thước bộ đệm cần thiết trong lpcchValueName và lpcbData.
3. RegDeleteValue

Xóa khóa giá trị trong khóa

LONG WINAPI RegDeleteValue(

\_In\_ HKEY hKey,

\_In\_opt\_ LPCTSTR lpValueName

);

* hKey [in]: Handle khóa đang mở, phải có quyền KEY\_SET\_VALUE. hKey trả về từ RegCreateKeyEx hay RegOpenKeyEx hoặc:

HKEY\_CLASSES\_ROOT, HKEY\_CURRENT\_CONFIG, HKEY\_CURRENT\_USER, HKEY\_LOCAL\_MACHINE, HKEY\_USERS

* lpValueName [in, optional]: Tên giá trị cần xóa. Nếu để NULL hay chuỗi rỗng, thì các giá trị set bởi RegSetValue sẽ bị xóa.

Hàm trả về ERROR\_SUCCESS nếu thành công, ngược lại trả về error code.

1. Registry nâng cao
2. HKEY\_CLASSES\_ROOT

Khóa gốc HKEY\_CLASSES\_ROOT (HKCR) gồm thông tin ánh xạ phần mở rộng file và đăng ký COM class như ProgIDs, CLSIDs, và IIDs.

Mục đích của HKCR là để tương thích ngược với registry 16-bit Windows.

Thông tin đăng ký của class và phần mở rộng file chứa trong HKEY\_LOCAL\_MACHINE và HKEY\_CURRENT\_USER.

* HKEY\_LOCAL\_MACHINE\Software\Classes : cấu hình mặc định để cài đặt cho tất cả user trên local computer.
* HKEY\_CURRENT\_USER\Software\Classes : cấu hình riêng của user hiện hành.
* HKEY\_CLASSES\_ROOT : là tổng hợp thông tin cấu hình của 2 khóa gốc trên và đồng thời là tổng hợp thông tin về ứng dụng thiết kế cho phiên bản Windows trước đó.

Cấu hình riêng của user có quyền ưu tiên cao hơn cấu hình mặc định, ví dụ user có thể chỉ định ứng dụng mặc định cho phần mở rộng .pdf

Khi chúng ta gọi các Registry API như RegOpenKeyEx hay RegQueryValueEx tham chiếu tới các subkey trong HKEY\_CLASSES\_ROOT, hệ thống sẽ tổng hợp thông tin HKEY\_LOCAL\_MACHINE\Software\Classes với HKEY\_CURRENT\_USER\Software\Classes

Một số điều lưu ý khi tùy chỉnh khóa:

* Truy xuất cấu hình user hiện hành: tham chiếu tới HKEY\_CURRENT\_USER\Software\Classes
* Truy xuất cấu hình mặc định: tham chiếu tới HKEY\_LOCAL\_MACHINE\Software\Classes.
* Trường hợp ghi khóa vào HKEY\_CLASSES\_ROOT: hệ thống cập nhật trong HKEY\_LOCAL\_MACHINE\Software\Classes
* Trường hợp ghi giá trị vào khóa trong HKEY\_CLASSES\_ROOT và khóa này đã tồn tại trên HKEY\_CURRENT\_USER\Software\Classes: hệ thống cập nhật trong HKEY\_CURRENT\_USER\Software\Classes

Đối với các process thực thi độc lập với user:

* Không nên truy xuất HKEY\_CLASSES\_ROOT, dùng tường minh HKEY\_LOCAL\_MACHINE\Software\Classes để cập nhật cấu hình mặc định.
* Dùng RegOpenUserClassesRoot: có chức năng tổng hợp thông tin cấu hình mặc định và user riêng biệt cho các ứng dụng kiểu services, làm việc với các clients hơn là user hiện hành. Tuy nhiên, RegOpenUserClassesRoot sẽ thất bại nếu user profile chưa nạp. Hệ thống tự động nạp profile cho user khi user đăng nhập. Để nạp profile cho user khác, dùng LoadUserProfile.

Ví dụ: Một thread chạy độc lập với client, có thể dùng RegOpenUserClassesRoot để lấy thông tin tổng hợp của client riêng biệt nào đó.

Nếu ứng dụng có quyền Administrator và User Account Control bị vô hiệu, COM runtime sẽ bỏ qua cấu hình COM từng user và chỉ truy xuất được cấu hình COM từng computer.

Ứng dụng yêu cầu quyền Administrator nên đăng kí COM object vào HKEY\_LOCAL\_MACHINE\Software\Classes trong quá trình cài đặt từng máy.

1. Cơ chế tổng hợp của HKCR

Luật tổng hợp thông tin của HKEY\_CLASSES\_ROOT như sau:

* Nạp tất cả subkey của HKEY\_CURRENT\_USER\Software\Classes
* Nạp tất cả subkey của HKEY\_LOCAL\_MACHINE\Software\Classes, nhưng không nạp các key đã có trên HKEY\_CURRENT\_USER\Software\Classes

1. Import và export Registry

Ứng dụng có thể lưu một phần của registry thành file và nạp nội dung file vào lại registry.

Registry file hữu dụng để xử lí lượng lớn dữ liệu, như sao lưu và phục hồi registry.

Để lưu key, subkey và giá trị, chúng ta dùng API RegSaveKey hay RegSaveKeyEx : 2 API tạo ra file với thuộc tính nén.

File được tạo ra trong thư mục hiện hành của process (key nội bộ) hay %systemroot%\system32 đối với remote key.

* RegSaveKey chỉ tạo file theo định dạng chuẩn, chỉ hỗ trợ Windows 2000, nhưng tương thích trên Window XP về sau.
* RegSaveKeyEx tạo file theo cả định dạng chuẩn và mới, từ Window XP về sau: tùy theo flag REG\_STANDARD\_FORMAT hay REG\_LATEST\_FORMAT

Có thể dùng RegSaveKeyEx để chuyển registry file từ định dạng chuẩn sang định dạng mới.

Để nạp registry file, chúng ta dùng RegLoadKey, RegReplaceKey, hay RegRestoreKey

* RegLoadKey : nạp dữ liệu file vào subkey của HKEY\_USERS hay HKEY\_LOCAL\_MACHINE trên máy local hay remote. Phương thức sẽ tạo subkey nếu chưa có.
* RegUnLoadKey : để phục hồi trạng thái bạn đầu registry sau khi nạp bằng RegLoadKey.
* RegReplaceKey : thay thế key , toàn bộ subkey và giá trị với dữ liệu trong file. Dữ liệu mới có hiệu lực khi hệ thống khởi động lại.
* RegRestoreKey : nạp dữ liệu file vào registry trên máy local hay remote. Phương thức sẽ thay thế các subkey và giá trị bởi subkey và giá trị của top-key nằm trong file.

Để truy xuất thông tin registry trên máy remote trong môi trường mạng: chúng ta có thể dùng trình Regedit hay phương thức RegConnectRegistry với đủ quyền truy xuất trên máy remote.

Ứng dụng muốn sao lưu registry (gồm system file và hive) nên dùng Volume Shadow Copy services.

1. RegSaveKeyEx

LONG WINAPI RegSaveKeyEx(

\_In\_ HKEY hKey,

\_In\_ LPCTSTR lpFile,

\_In\_opt\_ LPSECURITY\_ATTRIBUTES lpSecurityAttributes,

\_In\_ DWORD Flags

);

* hKey [in]: Handke khóa đang mở, phương thức không dùng cho HKEY\_CLASSES\_ROOT
* lpFile [in]: Tên file để lưu, file không được tồn tại. File sau khi tạo có thuộc tính nén.   
  Nếu tên file không có đường dẫn, file mặc định tạo trong thư mục hiện thành với máy local hoặc %systemroot%\system32 với máy remote.
* lpSecurityAttributes [in, optional]: Con trỏ SECURITY\_ATTRIBUTES cho file. File nhận security descriptor mặc định nếu để NULL, ACLs thừa kế từ thư mục cha  
  Flags [in]: Định dạng khóa chuẩn hay mới. Flags chỉ nhận 1 trong 3 giá trị dưới đây:

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| **REG\_STANDARD\_FORMAT**  1 | Định dạng chuẩn |
| **REG\_LATEST\_FORMAT**  2 | Định dạng mới |
| **REG\_NO\_COMPRESSION**  4 | file lưu không nén, ghi sẽ nhanh hơn. Tham số hKey phải chỉ rõ root hive trong **HKEY\_LOCAL\_MACHINE**hay **HKEY\_USERS**. Ví dụ: **HKLM\SOFTWARE** |

Hàm trả về:

ERROR\_SUCCESS : nếu thành công, ngược lại trả về error code nếu thất bại.

ERROR\_INVALID\_PARAMETER: Khi kết hợp nhiều giá trị cho tham số Flags, hoặc Flags = REG\_NO\_COMPRESSION nhưng hKey không phải root hive

RegSaveKeyEx chỉ làm việc với khóa cố định (non-volatile).

File tạo bởi RegSaveKeyEx được dùng cho RegLoadKey, RegReplaceKey, hay RegRestoreKey

Nếu quá trình tạo file bị lỗi, thì RegLoadKey, RegReplaceKey, hay RegRestoreKey sẽ thất bại.

Không nên dùng RegSaveKeyEx với RegRestoreKey để sao chép cây con registry.

Process phải có quyền SE\_BACKUP\_NAME.

1. RegRestoreKey

RegRestoreKey ghi đè vào subkey của khóa, tức thay thế toàn bộ subkey và giá trị của hKey bởi subkey và giá trị của top-key trong file.

LONG WINAPI RegRestoreKey(

\_In\_ HKEY hKey,

\_In\_ LPCTSTR lpFile,

\_In\_ DWORD dwFlags

);

* hKey [in]: Handle khóa đang mở. hKey trả về từ RegCreateKeyEx hay RegOpenKeyEx, hoặc khóa gốc: HKEY\_CLASSES\_ROOT, HKEY\_CURRENT\_CONFIG, HKEY\_CURRENT\_USER, HKEY\_LOCAL\_MACHINE, HKEY\_USERS
* lpFile [in]: Tên file registry tạo bởi RegSaveKey
* dwFlags [in]: định nghĩa cách ghi. Flags chỉ dùng 1 trong 2 Macro sau:

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| **REG\_FORCE\_RESTORE**  0x00000008L | If specified, the restore operation is executed even if open handles exist at or beneath the location in the registry hierarchy to which the *hKey* parameter points. |
| **REG\_WHOLE\_HIVE\_VOLATILE**  0x00000001L | If specified, a new, volatile (memory only) set of registry information, or hive, is created. If REG\_WHOLE\_HIVE\_VOLATILE is specified, the key identified by the *hKey* parameter must be either the **HKEY\_USERS** or **HKEY\_LOCAL\_MACHINE** value. |

Hàm trả về ERROR\_SUCCESS nếu thành công, ngược lại trả về error code.

Nếu có bất kỳ subkeys Handle nào của hKey đang mở, RegRestoreKey sẽ thất bại.

Process phải có quyền SE\_RESTORE\_NAME và SE\_BACKUP\_NAME.

Nếu hKey là remote handle key, đường dẫn trong lpFile cũng phải tương ứng với máy remote.

1. RegLoadKey

Phương thức sử dụng file tạo ta từ RegSaveKey, để tạo subkey trong HKEY\_USERS hay HKEY\_LOCAL\_MACHINE và nạp registry hive.

LONG WINAPI RegLoadKey(

\_In\_ HKEY hKey,

\_In\_opt\_ LPCTSTR lpSubKey,

\_In\_ LPCTSTR lpFile

);

* hKey [in]: Handle key trả về từ RegConnectRegistry hay HKEY\_LOCAL\_MACHINE, HKEY\_USERS.

Các subkey trong lpFile sẽ được tạo dưới hKey.

HKEY\_CLASSES\_ROOT và HKEY\_CURRENT\_USER chỉ là một phần từ HKEY\_LOCAL\_MACHINE, HKEY\_USERS nên không được tham chiếu vào hKey.

* lpSubKey [in, optional]: Tên subkey sẽ tạo ra dưới hKey. Subkey không nhất thiết là top-key trong file, phương thức sẽ tự phân tích và nạp đúng subkey vào registry.
* lpFile [in]: Đường dẫn file registry, đây phải là local file tạo ra từ RegSaveKey. Nếu file không tồn tại, hàm cũng tự sinh ra file.

Hàm tả về ERROR\_SUCCESS nếu thành công, ngược lại tả về error code nếu thất bại.

Process phải có quyền SE\_RESTORE\_NAME và SE\_BACKUP\_NAME

1. RegUnLoadKey

Unload dữ liệu registry từ thao tác RegLoadKey.

Phương thức tác động lên registry hive nhưng làm thay đổi file hive

LONG WINAPI RegUnLoadKey(

\_In\_ HKEY hKey,

\_In\_opt\_ LPCTSTR lpSubKey

);

* hKey [in]: Handle key cần unload, trả về từ RegConnectRegistry hay khóa gốc: HKEY\_LOCAL\_MACHINE, HKEY\_USERS
* lpSubKey [in, optional]: Tên subkey cần unload. Đây là subkey được load bởi RegLoadKey

Hàm tả về ERROR\_SUCCESS nếu thành công, ngược lại tả về error code nếu thất bại.

Process phải có quyền SE\_RESTORE\_NAME và SE\_BACKUP\_NAME

1. Dữ liệu 32-bit và 64-bit

On 64-bit Windows, portions of the registry entries are stored separately for 32-bit application and 64-bit applications and mapped into separate logical registry views using the registry redirector and registry reflection, because the 64-bit version of an application may use different registry keys and values than the 32-bit version. There are also shared registry keys that are not redirected or reflected.

The parent of each 64-bit registry node is the Image-Specific Node or ISN. The registry redirector transparently directs an application's registry access to the appropriate ISN subnode. Redirection subnodes in the registry tree are created automatically by the WOW64 component using the name Wow6432Node. As a result, it is essential not to name any registry key you create Wow6432Node.

The KEY\_WOW64\_64KEY and KEY\_WOW64\_32KEY flags enable explicit access to the 64-bit registry view and the 32-bit view, respectively. For more information, see Accessing an Alternate Registry View.

To disable and enable registry reflection for a particular key, use the RegDisableReflectionKey and RegEnableReflectionKey functions. Applications should disable reflection only for the registry keys that they create and not attempt to disable reflection for the predefined keys such as HKEY\_LOCAL\_MACHINE or HKEY\_CURRENT\_USER. To determine which keys are on the reflection list, use the RegQueryReflectionKey function.

1. User-profile

Một user-profile gồm 2 thành phần:

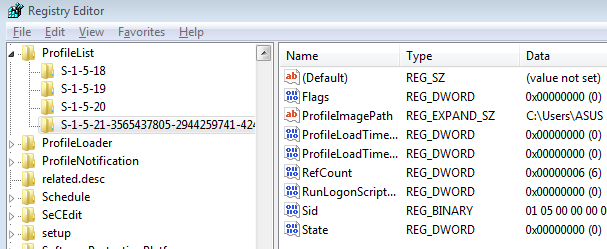
* Registry hive : nằm trong file NTuser.dat, được nạp vào khóa gốc HKEY\_CURRENT\_USER khi user đăng nhập, chứa thông tin cấu hình của ứng dụng, desktop, môi trường, kết nối mạng và máy in…
* Thư mục profile : mỗi user sẽ có thư mục profile riêng.

Thuận lợi của user-profile:

* Dễ quản lý thông tin cấu hình cho từng user
* User có thể đem theo cấu hình khi đăng nhập vào bất kì máy nào

1. Key ProfileList

Thông tin user-profile trong registry nằm trong khóa HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\ProfileList\SID

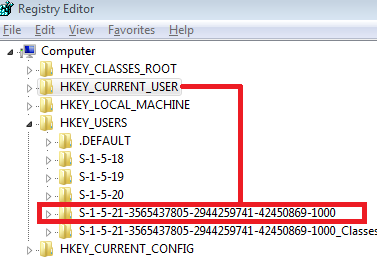
Mỗi SID tương ứng với một user-profile, tên subkey chính là mã số nhận dạng bảo mật SID (Security identifier).

Mỗi subkey SID đều có giá trị Sid chứa mã số SID của user

ProfileList còn cho biết các thông tin sau:

|  |  |
| --- | --- |
| **Directory** | **Description** |
| ProgramData (Windows Vista về sau)/All Users | Cấu hình ứng dụng mặc định cho tất cả user. Đường dẫn nằm trong giá trị ***ProgramData*** của subkey: HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList |
| *\Users\Default* | Cấu hình Local use-profile mặc định. Đường dẫn nằm trong giá trị ***Default*** của subkey: HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList |
| \User\... | Mỗi user riêng biệt sẽ có thư mục con chứa cấu hình trong *\User\*  Đường dẫn nằm trong giá trị ***ProfileImagePath*** của subkey:  HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList\SID |

1. User-profile hive file

User-profile hive nằm trong file ntuser.dat, mỗi thư mục user-profile đều có file này.

Khi user đăng nhập, Windows nạp ntuser.dat vào đúng tên SID (tên subkey) của HKEY\_USERS\SID

Sau đó ánh xạ tới HKEY\_CURRENT\_USER

Khóa HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\hivelist: chứa thông tin đường dẫn tất cả hive file của hệ thống.

1. Local profile

Local Profile được tạo khi user lần đầu đăng nhập, không thể đem theo trên máy khác.

Khi user đăng nhập, Windows kiểm tra danh sách user-profile trong key ProfileList.

* Nếu tìm thấy entry: hệ thống sẽ nạp thông tin.
* Nếu không tìm thấy entry : Window kiểm tra NETLOGON\Default User\ trên domain controller cho user-profile mặc định và sao chép vào %SystemDrive%\User\user\_name nếu tìm thấy Default User.
* Nếu không tìm thấy entry: Window không phải thành viên domain hoặc không tìm thấy Default User trên NETLOGON, nó sẽ dùng user-proflie mặc định trên máy local và sao chép vào %SystemDrive%\User\user\_name

Sau đó ntuser.dat được nạp vào HKEY\_USER và ánh xạ thông tin tới khóa gốc HKEY\_CURRENT\_USER.

Khi user thoát, profile hive sẽ bị loại bỏ khỏi registry.

1. Roaming Profiles

Roaming profile áp dụng cho toàn user trên domain, user có thể đem profile trên bất kì máy nào trong domain.

Khi user đăng nhập, Windows kiểm tra danh sách user-profile trong key ProfileList.

* Nếu tìm thấy entry: Windows sẽ nạp chồng profile trên server lên local profile tìm thấy.
* Nếu không tìm thấy entry : Window kiểm tra NETLOGON\ Default User\ trên domain controller cho user-profile mặc định và sao chép vào %SystemDrive%\User\user\_name nếu tìm thấy Default User
* Nếu không tìm thấy entry: Window không phải thành viên domain hoặc không tìm thấy Default User trên NETLOGON, nó sẽ dùng user-proflie mặc định trên máy local và sao chép vào %SystemDrive%\User\user\_name

Sau đó ntuser.dat được nạp vào HKEY\_USER và ánh xạ thông tin tới khóa gốc HKEY\_CURRENT\_USER.

Khi user thoát, profile hive sẽ bị loại bỏ khỏi registry và Windows cập nhật thư mục profile lên server.

1. Ứng dụng Registry
2. Liệt kê danh sách registry

Duyệt subkey – value của khóa theo 2 cách:

* Chỉ duyệt key : theo cú pháp *- regQuery Subkey*
* Dùng đệ qui duyệt toàn bộ subkey – value của key : theo cú pháp *-regQuery -R Subkey*

Đối số command-line nhận tên key, có kèm theo đường dẫn, ví dụ: *HKEY\_LOCAL\_MACHINE\HARDWARE\DESCRIPTION*

Tên key luôn bắt đầu từ khóa gốc. Ứng dụng phải nhận biết được khóa gốc qua tên key.

### Quản lý vùng nhớ, Memory-map file

1. Quản lý vùng nhớ

Windows quản lý việc ánh xạ qua lại giữa vùng nhớ ảo và RAM thông qua vài cơ chế như page swapping, demand paging…

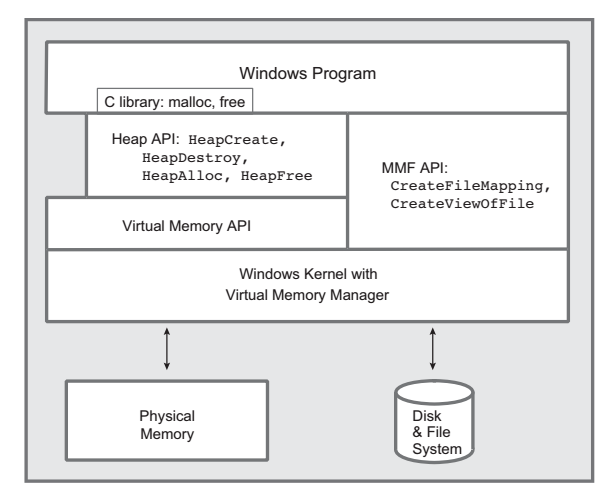
Mỗi process sở hữu một không gian vùng nhớ ảo riêng, có thể lớn hơn cả vùng nhớ vật lý.

Hầu hết vùng nhớ ảo triển khai trên đĩa cứng, dưới dạng file hệ thống hay file thông thường. Dù vậy, việc dùng quá nhiều vùng nhớ ảo sẽ ảnh hưởng tới hiệu suất, ứng dụng chỉ nên dùng vừa đủ.

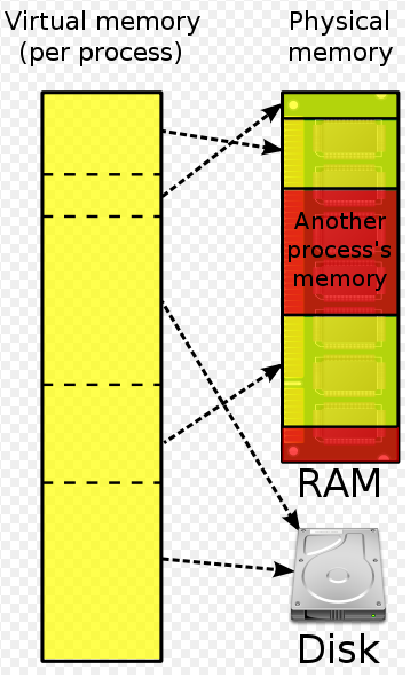
Window dùng khái niệm về Heap như một bộ quản lý vùng nhớ riêng cho từng process

Bên dưới là kiến trúc mô hình quản lý vùng nhớ của Windows: Các API quản lý vùng nhớ ảo (VirtualAlloc ,VirtualFree, VirtualLock, VirtualUnlock…) làm việc với các papes.

Các Heap API quản lý vùng nhớ ở cấp người dùng.



1. Không gian địa chỉ ảo

Thuật ngữ vùng nhớ ảo ám chỉ kỹ thuật quản lý bộ nhớ, được triển khai trên cả phần cứng và phần mềm.

Nó ánh xạ địa chỉ vùng nhớ dùng bởi chương trình, gọi là địa chỉ ảo, với địa chỉ vật lý. Vùng nhớ chính của process là một dải địa chỉ liên tục. OS quản lý vùng địa chỉ ảo và việc ánh xạ vùng nhớ thật tới vùng nhớ ảo.

Còn phần cứng phiên dịch địa chỉ nằm trên CPU, hay còn gọi là đơn vị quản lý vùng nhớ (Memory Manegement Unit), tự động phiên dịch vùng nhớ ảo sang vùng nhớ vật lý.

Phần mềm trong OS có thể được mở rộng vùng nhớ ảo vượt quá khả năng vùng nhớ thật.

Điểm thuận lợi chính của vùng nhớ ảo là giúp trình ứng dụng không cần phải quản lý vùng nhớ chia sẻ, tằng tính bảo mật qua việc cách ly vùng nhớ, và có thể dùng bộ nhớ nhiều hơn mà vùng nhớ vật lý cung cấp, kỹ thuật bộ nhớ đệm.

Vùng nhớ ảo giúp lập trình ứng dụng trở nên dễ hơn, các đoạn vùng nhớ vật lý bị giấu đi, ủy quyền cho phần lõi gánh vác việc quản lý vùng nhớ thứ cấp và mỗi process sẽ chỉ chạy trong vùng địa chỉ cấp phát riêng cho nó thôi (cơ chế ngăn không cho chương trình mở rộng hay truy xuất vùng nhớ vượt quá phạm vi cấp phát).

Ảo hóa vùng nhớ được xem là khái niệm mở đầu cho lý thuyết vùng nhớ ảo.

1. GetSystemInfo

Lấy thông tin hệ thống hiện hành

void WINAPI GetSystemInfo(

\_Out\_ LPSYSTEM\_INFO lpSystemInfo

);

* lpSystemInfo [out]: con trỏ cấu trúc SYSTEM\_INFO nhận thông tin trả về.

Ví dụ:

SYSTEM\_INFO siSysInfo;

// Lay thong tin phan cung vao SYSTEM\_INFO.

GetSystemInfo(&siSysInfo);

// In thong tin phan cung.

\_tprintf(\_T("Hardware information: \n"));

\_tprintf(\_T(" OEM ID: %u\n"), siSysInfo.dwOemId);

\_tprintf(\_T(" Number of processors: %u\n"),siSysInfo.dwNumberOfProcessors);

\_tprintf(\_T(" Page size: %u\n"), siSysInfo.dwPageSize);

\_tprintf(\_T(" Processor type: %u\n"), siSysInfo.dwProcessorType);

\_tprintf(\_T(" Minimum application address: %lx\n"),siSysInfo.lpMinimumApplicationAddress);

\_tprintf(\_T(" Maximum application address: %lx\n"),siSysInfo.lpMaximumApplicationAddress);

\_tprintf(\_T(" Active processor mask: %u\n"),siSysInfo.dwActiveProcessorMask);

1. GetProcessHeap

Mỗi process sở hữu head mặc định, để lấy head handle của process hiện hành, ta dùng GetProcessHeap:

HANDLE WINAPI GetProcessHeap(void);

Hàm trả về Handle heap nếu thành công. Ngược lại, hàm trả về NULL nếu thất bại. Dùng GetLastError để bug lỗi.

Process dùng Heap handle này để cấp phát vùng nhớ từ process heap mà không cần phải tạo head riêng từ HeapCreate, chúng ta dùng C library API (malloc, free, calloc, realloc) để cấp phát vùng nhớ từ process heap

Windows duy trì heap memory pools, một process có thể dùng nhiều heap.

Heap cũng là một object và có Handle, nhưng không phải kernel Handle

1. Lợi ích từ heap riêng

Việc tạo một head riêng đem lại những thuận lợi sau:

* Tăng hiệu suất
* Cấp phát hiệu quả hơn: cấp phát theo một kích thước cố định với heap qui mô nhỏ hiệu quả hơn là một heap lớn cấp phát đủ kích thước. Đồng thời làm giảm phân mảnh vùng nhớ
* Chống rò rỉ: chỉ cần một lần gọi hàm, tất cả vùng nhớ sẽ bị thu hồi.

1. HeapCreate

Tạo Handle heap riêng cho process hiện hành. Hệ thống sẽ dành riêng 1 vùng nhớ ảo của process và cấp phát một lượng vùng nhớ vậy lý cho heap mới tạo.

HANDLE WINAPI HeapCreate(

\_In\_ DWORD flOptions,

\_In\_ SIZE\_T dwInitialSize,

\_In\_ SIZE\_T dwMaximumSize

);

* flOptions [in]: Tùy chọn cấp phát. Có thể là zero hoặc chỉ chấp nhận 1 trong các giá trị sau:

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| **HEAP\_CREATE\_ENABLE\_EXECUTE**  0x00040000 | Tất cả memory block cấp phát từ heap này cho phép được thực thi code, nếu phần cứng kích hoạt chế độ bảo vệ dữ liệu.  Sử dụng flag này trên ứng dụng chạy code từ heap.  Nếu flag này không kích hoạt và ứng dụng cố gắng chạy code từ page được bảo vệ, thì ứng dụng sẽ nhận được ngoại lệ với trạng thái code **STATUS\_ACCESS\_VIOLATION** |
| **HEAP\_GENERATE\_EXCEPTIONS**  0x00000004 | [**HeapAlloc**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa366597(v=vs.85).aspx) và [**HeapReAlloc**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa366704(v=vs.85).aspx)  ném ngoại lệ, thu hồi các vùng nhớ đã cấp phát thay vì trả về **NULL** khi thất bại. |
| **HEAP\_NO\_SERIALIZE**  0x00000001 | Mặc định được set. Không kích hoạt mutex cho Heap.  Tuy nhiên chế độ chống phân mảnh sẽ không làm việc được. |

* dwInitialSize [in]: Kích thước khởi tạo cho heap, tính theo bội số kích thước page, phải nhỏ hơn **dwMaximumSize**.  
  dwInitialSize = Zero tức khởi tạo 1 page. Dùng **GetSystemInfo** để kiểm tra kích thước 1 page.
* dwMaximumSize [in]: Quy định kích thước tối đa của head, tính theo bytes. Hàm sẽ tự làm tròn theo bội số page size, khi dùng **HeapAlloc** hay **HeapReAlloc** cấp phát, hệ thống sẽ tự mở rộng vùng nhớ nếu không đủ, tối đa là **dwMaximumSize**.

Nếu dwMaximumSize > 0 : Head có định, không thể mở rộng hơn **dwMaximumSize**. Block memory lớn nhất hệ thống hỗ trợ là 512 KB cho 32-bit process và 1024 KB cho 64-bit process.

Nếu **dwMaximumSize** = 0: Head mở rộng không giới hạn. Nếu yêu cầu cấp phát memory block lớn hơn kích thước block tối đa sẽ không sao cả, hệ thống sẽ tự gọi **VirtualAlloc** để lấy thêm vùng nhớ cho đủ block.

Ứng dụng cần nhiều vùng nhớ nên để **dwMaximumSize** = 0.

Hàm trả về Handle heap nếu thành công. Ngược lại trả về **NULL** nếu thất bại. **GetLastError** để bug lỗi.

Vùng nhớ của head chỉ có thể truy xuất bởi process tạo ra nó. Nếu Head tạo bởi thư viện liên kết động (DLL), xem như heap nằm trong không gian địa chỉ của process và chỉ được truy xuất bởi process đó.

1. HeadDestroy

Để hủy Handle heap, đồng thời giải phóng tất cả pages của handle heap.

BOOL WINAPI HeapDestroy(

\_In\_ HANDLE hHeap

);

* hHeap [in]: Handle heap cần hủy. Đây là handle tạo bởi **HeapCreate**, không phải Handle heap mặc định của process.

Hàm trả về TRUE nếu thành công, ngược lại FALSE, GestLastError để bug lỗi.

1. Quản lý HEAP
2. Multi-thread và ngoại lệ

Các phương thức vùng nhớ **HeapCreate**, **HeapAlloc**, **HeapFree**, **HeapReAlloc** và **HeapSize** có sử dụng 2 giá trị **HEAP\_GENERATE\_EXCEPTIONS** và **HEAP\_NO\_SERIALIZE** cho tham số **dwFlags**

Nếu flag **HEAP\_NO\_SERIALIZE** được set, sẽ vô hiệu mutex, nhiều thread có thể truy xuất đồng thời process heap, giúp cải thiện hiệu suất 1 chút nhưng sẽ dễ bị xung đột.

Để an toàn, chỉ nên dùng **HEAP\_NO\_SERIALIZE** trong vài trường hợp:

* Ứng dụng không triển khai thread
* Mỗi thread đều có ít nhất 1 Heap, thread khác không được truy xuất.
* Ứng dụng có cơ chế mutex riêng tránh xung đột khi truy xuất HeapAlloc và HeapReAlloc

Lưu ý: Không set **HEAP\_NO\_SERIALIZE** khi dùng **HeapAlloc** với Handle Heap mặc định của process, hệ thống có thể tạo ra các thread bên trong process của ứng dụng, như handler CTRL + C, cùng truy xuất tới heap mặc định process

**HEAP\_GENERATE\_EXCEPTIONS** khiến ngoại lệ ném ra khi quá trình cấp phát thất bại, bộ xử lí ngoại lệ sẽ dọn dẹp vùng nhớ đã cấp phát.

Có 2 error code trả về:

* **STATUS\_NO\_MEMORY**: Cạn vùng nhớ do phân mảnh, Heap đạt ngưỡng tối đa…
* **STATUS\_ACCESS\_VIOLATION**: Heap bị lỗi có thể do ứng dụng ghi tràn bộ đệm vùng nhớ cấp phát.

1. Phân mảnh Heap

Trên NT5 (Windows XP , Server 2k3), low-framentation heap được kích hoạt thông qua **GetProcessHeap** và **HeapSetInfomation**.

Kể từ NT6 (Vista về sau), cơ chế chống phân mảnh được mặc định cho Heap.

Khi LFH kích hoạt, Windows sẽ cấp phát theo block memory cố định, hệ thống chọn kích thước block memory có kích thước phù hợp nhất với kích thước yêu cầu.

Đối với các yêu cầu kích thước lớn hơn 16 KB, thuật toán LF sẽ không áp dụng

LF bị vô hiệu hóa khi Heap set **HEAP\_NO\_SERIALIZE** hoặc Head không mở rộng. Một vấn đề kỹ thuật nữa là LF sẽ không hoạt động trong môi trường debug heap.

Không thể vô hiệu LF khi nó đã kích hoạt.

Ứng dụng multithread rất có lợi khi tận dụng LF, điều kiện Head không set flag **HEAP\_NO\_SERIALIZE**, và các memory request phải dưới 16 KB.

1. HeapAlloc

Cấp phát block memory từ Heap.

LPVOID WINAPI HeapAlloc(

\_In\_ HANDLE hHeap,

\_In\_ DWORD dwFlags,

\_In\_ SIZE\_T dwBytes

);

* hHeap [in]: Handle heap trả về từ **HeapCreate** hay **GetProcessHeap**.
* dwFlags [in]: Tùy chọn cấp phát, nhận 1 trong các giá trị sau:

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| **HEAP\_GENERATE\_EXCEPTIONS**  0x00000004 | [**HeapAlloc**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa366597(v=vs.85).aspx) và [**HeapReAlloc**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa366704(v=vs.85).aspx)  ném ngoại lệ, thu hồi các vùng nhớ đã cấp phát thay vì trả về **NULL** khi thất bại. |
| **HEAP\_NO\_SERIALIZE**  0x00000001 | Serialized access will not be used for this allocation. For more information, see Remarks.  To ensure that serialized access is disabled for all calls to this function, specify **HEAP\_NO\_SERIALIZE** in the call to [**HeapCreate**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa366599(v=vs.85).aspx). In this case, it is not necessary to additionally specify **HEAP\_NO\_SERIALIZE** in this function call.  This value should not be specified when accessing the process's default heap. The system may create additional threads within the application's process, such as a CTRL+C handler, that simultaneously access the process's default heap. |
| **HEAP\_ZERO\_MEMORY**  0x00000008 | The allocated memory will be initialized to zero. Otherwise, the memory is not initialized to zero. |

* dwBytes [in]: Kích thước cấp phát tính theo bytes. Nếu Heap không thể mở rộng, dwBytes phải nhỏ hơn **0x7FFF8**

Hàm trả về địa chỉ memory block nếu thành công.

Nếu hàm thất bại và dwFlags không set **HEAP\_GENERATE\_EXCEPTIONS**: trả về NULL

Nếu hàm thất bại và dwFlags set **HEAP\_GENERATE\_EXCEPTIONS**: trả về **STATUS\_NO\_MEMORY** do thiếu hụt vùng nhớ hoặc **STATUS\_ACCESS\_VIOLATION** do lỗi heap , tham số không hợp lệ…

Phương thức không **SetLastError**.

1. HeapFree

Thu hồi vùng nhớ cấp phát bởi **HeapAlloc** hay **HeapReAlloc**

BOOL WINAPI HeapFree(

\_In\_ HANDLE hHeap,

\_In\_ DWORD dwFlags,

\_In\_ LPVOID lpMem

);

* hHeap [in]: Handle heap
* dwFlags [in]: tùy chọn thu hồi.

**HEAP\_NO\_SERIALIZE** (0x00000001): Truy xuất kiểu Serialized không được chấp nhận.

Khi tạo Handle Heap, flag **HEAP\_NO\_SERIALIZE** cũng có tác dụng tương tự, nhưng nếu đã set thì không cần phải khai báo **HEAP\_NO\_SERIALIZE** khi cấp phát.

Không set **HEAP\_NO\_SERIALIZE** khi truy xuất process heap.

* lpMem [in]: Địa chỉ cấp phát bởi **HeapAlloc** hay **HeapReAlloc**.

Hàm trả về TRUE nếu thành công, ngược lại FALSE. GetLastError để bug lỗi.

Gọi **HeapFree** hai lần cho cùng địa chỉ sẽ gây lỗi, **HeapAlloc** có thể sẽ trả về cùng một địa chỉ 2 lần.

1. HeapReAlloc

Tái phân bổ vùng nhớ, cho phép mở rộng hay thu nhỏ kích thước vùng nhớ đã cấp phát.

HeapReAlloc đảm bảo nội dung trong vùng nhớ được toàn vẹn sau khi tái phân bổ. Địa chỉ sau khi tái phân bổ có thể là địa chỉ mới

LPVOID WINAPI HeapReAlloc(

\_In\_ HANDLE hHeap,

\_In\_ DWORD dwFlags,

\_In\_ LPVOID lpMem,

\_In\_ SIZE\_T dwBytes

);

* hHeap [in]:Handle Heap trả về từ **HeapCreate** hay **GetProcessHeap**
* dwFlags [in]: qui định cách tái phân bổ. Nhận 1 trong các Macro sau:

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| **HEAP\_GENERATE\_EXCEPTIONS**  0x00000004 | The operating-system raises an exception to indicate a function failure, such as an out-of-memory condition, instead of returning **NULL**.  To ensure that exceptions are generated for all calls to this function, specify **HEAP\_GENERATE\_EXCEPTIONS** in the call to [**HeapCreate**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa366599(v=vs.85).aspx). In this case, it is not necessary to additionally specify **HEAP\_GENERATE\_EXCEPTIONS** in this function call. |
| **HEAP\_NO\_SERIALIZE**  0x00000001 | Serialized access will not be used. For more information, see Remarks.  To ensure that serialized access is disabled for all calls to this function, specify **HEAP\_NO\_SERIALIZE** in the call to [**HeapCreate**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa366599(v=vs.85).aspx). In this case, it is not necessary to additionally specify **HEAP\_NO\_SERIALIZE** in this function call.  This value should not be specified when accessing the process heap. The system may create additional threads within the application's process, such as a CTRL+C handler, that simultaneously access the process heap. |
| **HEAP\_REALLOC\_IN\_PLACE\_ONLY**  0x00000010 | There can be no movement when reallocating a memory block. If this value is not specified, the function may move the block to a new location. If this value is specified and the block cannot be resized without moving, the function fails, leaving the original memory block unchanged. |
| **HEAP\_ZERO\_MEMORY**  0x00000008 | If the reallocation request is for a larger size, the additional region of memory beyond the original size be initialized to zero. The contents of the memory block up to its original size are unaffected. |

* lpMem [in]: Địa chỉ vùng nhớ cần tái phân bổ
* dwBytes [in]: Kích thước mới, tính theo bytes. Nếu Heap không mỏ rộng được, dwBytes phải nhỏ hơn 0x7FFF8.

Hàm trả về địa chỉ vùng nhớ tái phân bổ nếu thành công.

Nếu hàm thất bại và dwFlags không set HEAP\_GENERATE\_EXCEPTIONS: trả về NULL

Nếu hàm thất bại và dwFlags set HEAP\_GENERATE\_EXCEPTIONS: trả về STATUS\_NO\_MEMORY do thiếu hụt vùng nhớ hoặc STATUS\_ACCESS\_VIOLATION do lỗi heap , tham số không hợp lệ…

Trường hợp thất bại, vùng nhớ vẫn chưa bị thu hồi.

Phương thức không SetLastError.

1. HeapSize

Do Heap cấp phát theo block memory, nên giá trị trả về luôn là bội số của block memory size. Có thể lớn hơn hay bằng kích thước yêu cầu.

SIZE\_T WINAPI HeapSize(

\_In\_ HANDLE hHeap,

\_In\_ DWORD dwFlags,

\_In\_ LPCVOID lpMem

);

* hHeap [in]: Handle Heap trả về từ **HeapCreate** hay **GetProcessHeap**
* dwFlags [in]: **HEAP\_NO\_SERIALIZE** (0x00000001): Truy xuất kiểu Serialized không được chấp nhận.

Khi tạo Handle Heap, flag **HEAP\_NO\_SERIALIZE** cũng có tác dụng tương tự, nhưng nếu đã set thì không cần phải khai báo **HEAP\_NO\_SERIALIZE** khi cấp phát.

Không set **HEAP\_NO\_SERIALIZE** khi truy xuất process heap.

* pMem [in]: Địa chỉ vùng nhớ

Hàm trả về kích thước vùng nhớ theo bytes nếu thành công. Ngược lại, hàm trả về -1, kiểu SIZE\_T.

Phương thức không **SetLastError**.

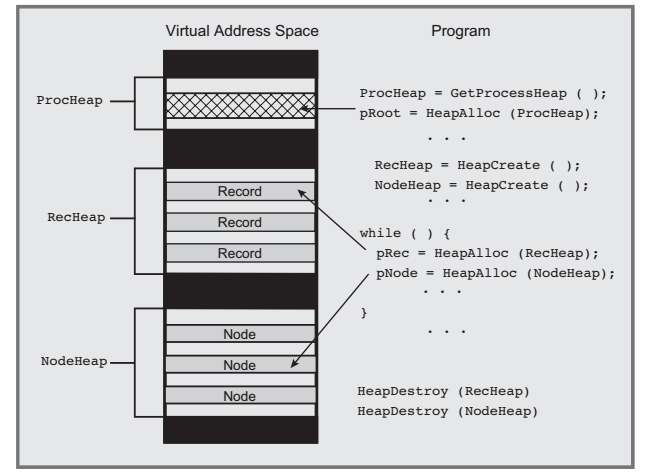
1. Sử dụng Heap
2. Sắp xếp file với cây nhị phân

Ứng dụng sẽ dùng hai Heap cho cây nhị phân:

* Node Heap: Cấu trúc có kích thước cố định, gồm khóa và con trỏ tới data record trong Data Heap
* Data Heap: Chứa chuỗi kí tự tên file, kích thước không cố định

Cấu trúc cây dùng để sắp xếp tên file.

Khóa sẽ là 8-byte đầu tiên tên file (số byte có thể tùy biến).

Hình bên dưới minh họa quy trình tạo và cấp phát của Heap.

Node Root sẽ nằm trong Heap mặc định của process

Nếu dùng C,C++ library , chúng ta phải triển khai bộ quản lý vùng nhớ riêng để lưu trữ tách biệt Node và Data.

Ứng dụng chỉ cần hủy Heap khi chuẩn bị thao tác sắp xếp mới, không cần phải hủy từng Node.

Lỗi cấp phát được xử lí exception, nên không cần phải kiểm tra con trỏ với NULL.

Lưu ý:

* Sau khi mở rộng, dải địa chỉ của Heap quản lý có thể không liên tiếp.
* Không gian địa chỉ chỉ xử lí 3GB cho Win32, với Win64 thì không giới hạn kích thước file.

Ứng dụng triển khai 4 hàm: FillTree, InsertTree, Scan, KeyCompare.

Xử lí ngoại lệ khi mở file và heap

### Process và thread:

1. Process:

Trong máy tính, một tiến trình là một bản thể chương trình đang được thực thi. Nó chứa mã lệnh và các hoạt động hiện thời. Tùy thuộc vào Os, process có thể gồm nhiều thread để thực thi các chỉ thị cùng lúc.

Một vài process cùng chạy một chương trình giống nhau; chúng là nhiều bản thể của chương trình.

Đa tác vụ là phương pháp cho phép nhiều process cùng chia sẻ nhiều CPUs và tài nguyên hệ thống khác. Mỗi CPU xử lí một tác vụ một lần. Tuy nhiên, đa tác vụ còn cho phép mỗi bộ xử lí hoán chuyển giữa các tác vụ mà không cần đợi tác vụ hoàn thành. Tùy theo Os, việc hoán chuyển thường diễn ra cho các tiến trình I/O, khi tác vụ chỉ ra là nó có thể được hoán chuyển, hay qua ngắt phần cứng.

Dạng thức phổ biến của đa tác vụ là chia sẻ thời gian (Time-sharing). Time-sharing là phương thức cho phép khả năng phản hồi nhanh khi có tương tác người dùng trên ứng dụng. Trong hệ thống time-sharing, việc hoán chuyển giữa các tác vụ diễn ra rất nhanh, khiến chúng ta cảm thấy nhiều process đang thực thi cùng lúc trên cùng bộ xử lí. Việc thực thi nhiều process đồng thời như vậy gọi là “concurrency”

Vì lý do bảo mật và tin cậy, hầu hết mọi Os hiện nay đều không cho các process chạy độc lập giao tiếp trực tiếp với nhau, nhưng cung cấp một cơ chế giới hạn cho giao tiếp nội bộ giữa các process.

Nói chung, process gồm các thành phần sau:

* Bản sao mã lệnh của chương trình
* Vùng nhớ (thường là một số vùng của vùng nhớ ảo) chứa mã thực thi, dữ liệu xử lí (xuất , nhập), call stack (kiểm soát các thủ tục con và sự kiện), và vùng nhớ heap cho dữ liệu tính toán phát sinh trong quá trình chạy
* Mã số descriptors , của tài nguyên được cấp phát cho process, như file descriptors (Unix) hay handles (Window), dữ liệu nguồn và sinks.
* Thuộc tính bảo mật, như người dùng sở hữu process và phân quyền của process (cho phép hoạt động)
* Trạng thái CPU, như nội dung thanh ghi register, địa chỉ vùng nhớ vật lý. Trạng thái thường được chứa trong thanh ghi máy tính khi process đang thực thi, và trong vùng nhớ khi không chạy.

Os giữ hầu hết các thông tin process hoạt động trong cấu trúc dữ liệu gọi là bộ điều khiển process. Bất kì bộ tài nguyên con, đều tương ứng với mỗi threads trong Os, hỗ trợ threads hay process con.

Os giữ các process riêng biệt và cấp phát tài nguyên mà chúng cần, nên giữa các process không gây trở ngại cho nhau và gây lỗi hệ thống. Os cũng cung cấp cơ chế giao tiếp nội bộ, cho phép các process tương tác với nhau một cách an toàn và có thể đoán trước được.

1. Đa tác vụ và quản lý process

Os đa nhiệm hoán chuyển xử lí giữa các process, giúp chúng thực thi một cách đồng thời (kiểu song song), dù thực tế chỉ có 1 process thực thi tại một thời điểm trên một CPU một nhân.

Process thường được gọi là tác vụ trong Os.

Nếu một process đang yêu cầu gì đó mà chúng phải đợi, nó sẽ bị khóa. Khi process trong trạng thái khóa,

If a process requests something for which it must wait, it will be blocked. When the process is in the blocked state, it is eligible for swapping to disk, but this is transparent in a virtual memory system, where regions of a process's memory may be really on disk and not in main memory at any time. Note that even unused portions of active processes/tasks (executing programs) are eligible for swapping to disk. All parts of an executing program and its data do not have to be in physical memory for the associated process to be active.

1. Trạng thái process

The various process states, displayed in a state diagram, with arrows indicating possible transitions between states.

An operating system kernel that allows multitasking needs processes to have certain states. Names for these states are not standardised, but they have similar functionality.[1]

First, the process is "created" by being loaded from a secondary storage device (hard disk drive, CD-ROM, etc.) into main memory. After that the process scheduler assigns it the "waiting" state.

While the process is "waiting", it waits for the scheduler to do a so-called context switch and load the process into the processor. The process state then becomes "running", and the processor executes the process instructions.

If a process needs to wait for a resource (wait for user input or file to open, for example), it is assigned the "blocked" state. The process state is changed back to "waiting" when the process no longer needs to wait.

Once the process finishes execution, or is terminated by the operating system, it is no longer needed. The process is removed instantly or is moved to the "terminated" state. When removed, it just waits to be removed from main memory.

1. Thread

Thread là tiến trình thực thi một đoạn nhỏ của chương trình, các thread được quản lý bởi trình lập biểu của Os. Thread trên mỗi loại Os sẽ khác nhau, nhưng hầu hết thì thread được xem là một phần của process.

Nhiều thread có thể tồn tại trong một process, thực thi đồng thời và chia sẻ tài nguyên (như vùng nhớ), trong khi giữa các process không chia sẻ với nhau.

Trên một CPU đơn, đa luồng được triển khai bởi cơ chế time-slicing, và CPU hoán chuyển giữa các thread.

Trên nhiều CPU hay CPU đa nhân, nhiều luồng được thực thi song song (cùng một thời điểm), mỗi CPU hay nhân thực thi một thread của process.

Lý thuyết về thread được phát triển từ 1967, chúng cũng được gọi là tác vụ trong lập trình đa nhiệm. Bộ lập biểu process của nhiều Os hiện nay hộ trợ trực tiếp cả time-cliced và CPU đa luồng, hạt nhân Os cho phép lập trình viên kiểm soát thread qua các API hệ thống.

Một số loại thread được gọi là kernel threads ( thread lõi), whereas lightweight processes (LWP) are a specific type of kernel thread that share the same state and information. Thêm vào đó, chương trình có thể có các thread người dùng khi dùng với bộ đếm thời gian, đánh tín hiệu hay các phương thức khác để ngắt tiến trình thực thi của chúng , performing a sort of ad hoc time-slicing.

1. Phân biệt Threads và processes

Thread có vài điểm khác với process trong Os đa nhiệm:

* Các process thường chạy độc lập, trong khi thread tồn tại như một phần của process
* Process chứa nhiều thông tin trạng thái hơn thread, trong khi nhiều thread trong một process cùng chia sẻ trạng thái như vùng nhớ và tài nguyên khác.
* Các process có vùng địa chỉ riêng, các thread trong cùng process cùng chia sẻ địa chỉ vùng nhớ.
* Các process chỉ tương tác với nhau thông qua cơ chế giao tiếp nội bộ

Phương thức xử lí luân phiên giữa các thread trong cùng process thường nhanh hơn giữa các process.  
Các hệ thống như Windows NT và OS/2 ít hao phí tài nguyên khi tạo threads và tạo thêm process thì tốn chi phí nhiều hơn.

1. Xử lí đa luồng

Xử lí đa luồng thường thấy trên các Os đa nhiệm. Đa luồng rất phổ biến trong lập trình và phương thức thực thi cho phép nhiều thread xử lí đồng thời trên một CPU. Nhưng thread này cùng chia sẻ tài nguyên của process, nhưng lại có thể chạy độc lập.

Phương pháp lập trình thread rất hữu dụng cho việc xử lý đồng thời. Áp dụng xử lí đa luồng trên một CPU cho phép xử lí song song trên hệ đa nhiệm.

Trinh ứng dụng xử lí đa luồng có các thuận lợi sau:

* Phản ứng nhanh : cho phép tiến trình input tiếp tục. Ứng dụng đơn thread khi tiến trình bị khóa, toàn bộ chương trình bị đóng băng. Bằng cách chuyển tiến trình có thời gian xử lí lâu sang một thread mới, người dùng vẫn tương tác được với ứng dụng, trong khi tiến trình đang được xử lí ngầm.  
  Thực ra có nhiều cách giúp ứng dụng không bị đóng băng (như dùng non-blocking I/O với kỹ thuật đánh tín hiệu trong Unix)
* Xử lí nhanh: Đặc biệt với nhiều CPU hay CPU đa nhân
* Ít tiêu hao tài nguyên: Ứng dụng tạo ra nhiều thread để đáp ứng cho nhiều clients một lúc, nhưng tốn ít tài nguyên hơn so với nhiều bản sao process của chương trình. Ví dụ: Apache HTTP server dùng một thread pools chờ kết nối, và một thread pool để xử lí kết nối.
* Khai thác tốt tài nguyên hệ thống: Như hệ thống file, ta dùng thread để xử lí dữ liệu nằm trong các thiết bị tốc độ truy xuất cao (như bộ nhớ cache của CPU), trong khi thread khác xử lí dữ liệu từ thiết bị chậm hơn như ổ đĩa cứng.
* Giao tiếp và chia sẻ tài nguyên đơn giản: không như process là phải có cơ chế IPC để truyền tin nhắn hay chia sẻ vùng nhớ , nhiều thread có thể giao tiếp qua dữ liệu, mã lệnh, và file mà chúng sẵn sàng chia sẻ
* Tính song song: ứng dụng khai thác hệ thống CPU đa nhân hay nhiều CPU để chia nhỏ dữ liệu, tác vụ thành các tác vụ con, sau đó để các kiến trúc bên dưới quản lý các threads chạy. (Hoặc luân phiền trên đơn nhân, hoặc song song trên đa nhân/ nhiều CPU). Môi trường tính toán trên GPU như CUDA hay OpenCL dùng xử lí đa luồng khi có hàng tá thread chạy song song trên nhiều nhân.

Và một số khó khăn :

* Tính đồng bộ: Vì các thread chia sẻ cùng địa chỉ vùng nhớ, nên lập trình viên cần phải tránh **“race condition”** và các hành vi không nằm ngoài dự đoán. Để kiểm soát, thread cần được lên kế hoạch để xử lí dữ liệu theo tuần tự.  
  Thread cần tiến trình ngăn ngừa xử lí chồng lấn ([**mutually exclusive**](https://en.wikipedia.org/wiki/Mutual_exclusion)**)**  thường dùng với [**semaphores**](https://en.wikipedia.org/wiki/Semaphore_(programming)) để ngăn trường hợp dữ liệu bị thay đổi bởi nhiều thread cùng lúc, việc dùng thread không cẩn thận có thể dẫn tới ứng dụng bị khóa chết.
* Trường hợp thread khiến process bị crash: Một thread như vậy là không hợp lệ, nó có thể phá vỡ các thread cùng process đang trong quá trình xử lí.

1. Định thời cho thread

OS định thời các thread để chúng tiếm quyền hay hợp tác với nhau. Tiếm quyền trong xử lí đa luồng thường được xem như một phương thức cấp cao, nó cho phép Os biết được các trạng thái hoán chuyển thread sẽ xảy ra.

Điểm bất lợi là khi phương thức tiếm quyền xảy ra vào thời điểm không thích hợp, gây ra lock convoy, đảo quyền ưu tiên hay hiệu ứng tiêu cực, mà chúng ta có thể tránh được bằng cách dùng phương pháp hợp tác giữa các thread.

Phương thức hợp tác yêu cầu chính các thread từ bỏ quyền kiểm soát khi chúng tới điểm dừng. Điều này có thể tạo ra lỗi nếu xảy ra trường hợp một thread phải chờ cho tài nguyên sẵn sàng.

Đầu năm 2000, hầu hết các máy tính để bàn chỉ có CPU đơn nhân, không hỗ trợ nhiều về thread. Trong năm 2002, Intel hỗ trợ xử lí đa luồng cho Pentium 4, dưới tên siêu phân luồng (Hyper-threading). Tới 2005, Intel giới thiệu Pentium D - 2 nhân và AMD cũng ra mắt chíp 2 nhân Athlon 64x2.

Chíp xử lí trong các hệ thống khép khín, nơi có yêu cầu cao cho hành vi thời gian thực, được hỗ trợ xử lí đa luồng bằng cách giảm thời gian hoán chuyển thread, tức là cấp phát các file thanh ghi chuyên dụng cho mỗi thread thay vì tiết kiệm

1. Processes, threads lõi, threads người dùng, và fibers

Thỉnh thoảng thread được tạo trong thư viện người dùng, được gọi là thread người dùng. Phần lõi không thể nhận biết chúng, nên thread người dùng được quản lý và định thời bởi người dùng. Một số thực thi dựa trên thread người dùng hoạt động trên thread lõi, để có thể hưởng lợi từ các bộ xử lí nhiều nhân hay nhiều chip (mô hình M:N).

Thuật ngữ thread trong phần này mặc định là đề cập tới thread lõi. Thread người dùng được khởi tạo bởi máy ảo, được gọi là green thread. Thread người dùng có thể tạo và quản lý khá nhanh, nhưng không thể có được sự thuận lợi của xử lí đa luồng hay xử lí đa nhiệm, và sẽ bị khóa nếu tất cả thread lõi tương ứng bị khóa thậm chí dù đang chạy.

Fiber là các đơn vị nhỏ hơn của cơ chế định thời hợp tác. Một fiber có thể tự quản lý cơ chế định thời trong bất kì thread nào, giúp tăng hiệu suất cho trình ứng dụng, thay vì phụ thuộc vào cơ chế định thời lõi (cơ chế này độc lập với trình ứng dụng).

Môi trường Lập trình song song như OpenMP thường khởi tạo các tác vụ qua fibers. Quan hệ mật thiết với fiber là co-routines, hoạt động ở tầng ngôn ngữ. Còn fiber hoạt động ở tầng hệ thống.

[**OpenMP (Open Multi-Processing)**](https://en.wikipedia.org/wiki/OpenMP) là một bộ API hỗ trợ nhiều platform có chia sẻ vùng nhớ để lập trình xử lí đa nhiệm trong C,C++ và Fortran, trên hầu hết kiến trúc CPU và Os, gồm Solaris, AIX, HP-UX, Linus, OS X và Win dows. Nó bao gồm một bộ biên dịch directives, thư viện thủ tục và biến môi trường có ảnh hưởng tới hành vi khi chạy trình ứng dụng.

Định thời cho thread thực hiện ở tầng lõi hay người dùng, và xử lí đa nhiệm hoạt động theo hai phương thức là tiếm quyền hay hợp tác.

Ở tầng lõi, một process chứa một hay nhiều thread lõi, chúng chia sẻ nhiều tài nguyên process như vùng nhớ, file handle. Một process được xem là một đơn vị tài nguyên, trong khi thread được xem là một đơn vị thực thi.

Định thời ở tầng lõi thường là tiếm quyền, phương thức hợp tác ít dùng hơn. Ở tầng người dùng , một process như hệ thống chạy runtime, tự process đó phải định thời cho các thread.

Nếu không có dữ liệu chia sẻ, như trong ngôn ngữ Erlang, chúng thường gọi là process, trong khi chúng chia sẻ tài nguyên vẫn gọi là thread, đặc biệt khi trong định thời tiếm quyền.

Trong cơ chế định thời hợp tác, thread ở tầng người dùng được gọi là fiber, mỗi process sẽ định thời thread theo cách khác nhau. Thread người dùng thực thi bởi thread lõi theo nhiều cách (một-một, nhiều-một, nhiều-nhiều). Thuật ngữ “light-weigh process” ám chỉ thread người dùng hay cơ chế định thời hợp tác tầng lõi cho thead người dùng .

Một process liên quan tới việc như tạo, hủy và hoán chuyển các process gọi là “heavy weight process” trong cơ chế định thời tầng lõi. Việc tạo hay hủy process mới tốn nhiều chi phí hệ thống, tài nguyên được cấp phát và phải được hủy. Process thường dùng phương thức tiếm quyền, và hoán chuyển giữa các process còn phát sinh thêm chi phí khác như làm cache flushing.

Tiến trình cách ly process giúp các tài nguyên của process không bị truy xuất bởi các process khác, trừ khi ta dùng phương thức đặc biệt như thừa kế file handle hay chia sẻ phân vùng nhớ, ánh xạ cùng file theo cách chia sẻ. ([**Tham khảo IPC**)](https://en.wikipedia.org/wiki/Interprocess_communication)

Thread lõi là đơn vị “light weight” của đinh thời lõi, một process có ít nhất 1 thread lõi. Các thread lõi không sở hữu tài nguyên ngoại trừ vùng nhớ stack ( tức bản sao của thanh ghi bao gồm bộ đếm chương trình) và bộ lưu trữ thread nội bộ nếu có, và chúng tốn ít chi phí để duy trì.

Hoán chuyển trạng thái giữa các thread ít tốn chi phí : thread chỉ yêu cầu trạng thái hoán chuyển (lưu và khôi phục thanh ghi và con trỏ stack), nhưng không thay đổi vùng nhớ ảo và do đó . Phần lõi có thể gán một thread cho mỗi nhân logic , và có thể trao đổi các thread bị khóa. Tuy nhiên, thread lõi mất thời gian hơn thread người dùng để trao đỏi.

1. Hybrid implementation examples

Scheduler activations used by the NetBSD native POSIX threads library implementation (an M:N model as opposed to a 1:1 kernel or userspace implementation model)

Marcel from the PM2 project.

The OS for the Tera/Cray MTA

Microsoft Windows 7[citation needed]

The Haskell compiler GHC uses lightweight threads which are scheduled on operating system threads.

Fiber implementation examples

Fibers can be implemented without operating system support, although some operating systems or libraries provide explicit support for them.

Win32 supplies a fiber API[7] (Windows NT 3.51 SP3 and later)

Ruby as Green threads

Netscape Portable Runtime (includes a user-space fibers implementation)

ribs2

1. InterProcess Communication (IPC)

In computer science, inter-process communication (IPC) is the activity of sharing data across multiple and commonly specialized processes using communication protocols. Typically, applications using IPC are categorized as clients and servers, where the client requests data and the server responds to client requests.[1] Many applications are both clients and servers, as commonly seen in distributed computing. Methods for achieving IPC are divided into categories which vary based on software requirements, such as performance and modularity requirements, and system circumstances, such as network bandwidth and latency.[1]

There are several reasons for implementing inter-process communication systems:

Sharing information; for example, web servers use IPC to share web documents and media with users through a web browser.

Distributing labor across systems; for example, Wikipedia uses multiple servers that communicate with one another using IPC to process user requests.[2]

Privilege separation; for example, HMI software systems are separated into layers based on privileges to minimize the risk of attacks. These layers communicate with one another using encrypted IPC. Để tạo ra một kênh để các process giao tiếp được với nhau, Microsoft đưa ra khái niệm về Pipe.

Pipe có chức năng tương tự như FILE stream, cũng sử dụng hàm ReadFile và WriteFile. Nhưng đối tượng làm việc của FILE stream là file, còn đối với Pipe stream là các process.

Kể từ .Net 3.5, bạn có thể dễ dàng thực hiện được việc truyền dữ liệu giữa các process (tiến trình) bằng các kênh Pipe, nhờ vào **namespace** mới **System.IO.Pipes**. Tùy theo mục đích, bạn có thể sử dụng loại Anonymous pipe hay Named pipe.

Trong lập trình, khái niệm Pipe có thể được hiểu một luồng có chức năng truyền tải dữ liệu giữa các processs. Pipe được chia thành hai loại:

1. Anonymous Pipe
2. Giới thiệu

Anonymous Pipe: dùng cho giao tiếp process trong máy tính cục bộ. Loại pipe này sử dụng ít thời gian xử lý hơn Named Pipe nhưng ít chức năng hơn.

Anonymous Pipe là kiểu byte-based IPC và chỉ truyền tài được một chiều, nó rất thích hợp để chuyển output của process này sang input của process khác

Anonymous Pipe không sử dụng được thông qua mạng và thường dùng để tạo kênh giao tiếp giữa các thread hay giữa các process cha/con. Anonymous Pipe hay dùng trong Linux và Unix.

1. Hàm CreatePipe và closeHandle

Mỗi Windows anonymous pipes gồm read handle và wirte handle. Các handles này có tính kế thừa nếu được khai báo.

Để tạo anonymous pipe và handle, ta dùng hàm **createpipe**()

BOOL createPipe(HANDLE readHnle, HANDLE writeHnle, SECURITY\_ATTRIBUTES \*pipeAtt, DWORD size)

* **readHnle*:*** Khai báo con trỏ read handle cho pipe
* **writeHnle*:*** Khai báo con trỏ write handle cho pipe
* **pipeAtt *:***  Con trỏ cấu trúc **SECURITY\_ATTRIBUTES** , dùng để xác định handle trả về có tính kế thừa (cho process con) hay không
* **size *:*** Kích thước theo byte của **pipeAtt.** Ta dùng công thức **sizeof pipeAtt.** Nếu bạn để = 0, hệ thống dùng giá trị mặc định.

Hàm trả về 0 nếu thành công, hoặc trả về error code nếu bị lỗi.

Đây là cấu trúc **SECURITY\_ATTRIBUTES**

typedef struct \_SECURITY\_ATTRIBUTES {

DWORD  nLength;

VOID lpSecurityDescriptor;

BOOL   bInheritHandle;

} SECURITY\_ATTRIBUTES, \*PSECURITY\_ATTRIBUTES, \*LPSECURITY\_ATTRIBUTES;

Thành viên **lpSecurityDescriptor chứa thông tin bảo mật của anonymous pipe được tạo ra từ hàm createPipe()**

Nếu **pipeAtt = NULL** , handle không có tính kế thừa và **lpSecurityDescriptor được gán** thông tin bảo mật mặc định.

**Sau khi kết thúc các tác vụ, bạn dùng hàm Closehandle() để giải phóng tài nguyên**

**Cú pháp:**

BOOL CloseHandle(HANDLE objHandle);

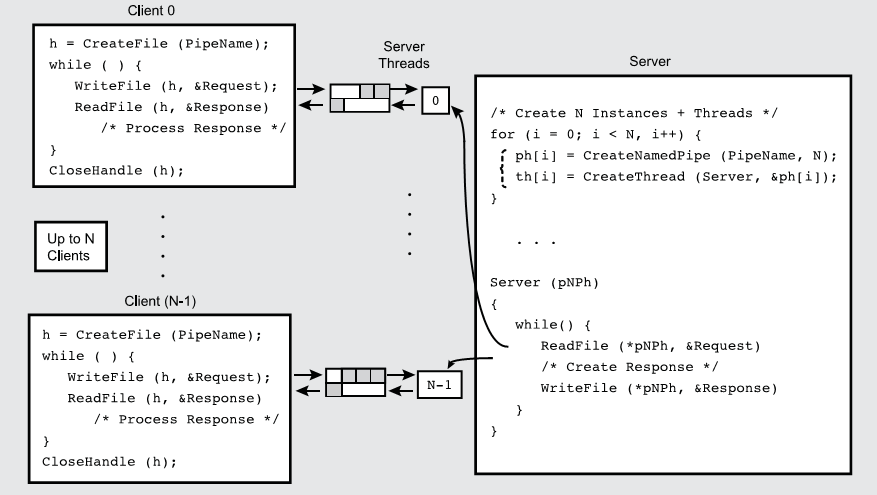
**Hàm trả về số != 0 nếu thành công, = 0 nếu bị lỗi.**

**Một pipe chỉ bị xóa khi handle cuối cùng của nó bị đóng.**

1. Name Pipe
2. Giới thiệu

Named Pipe: hỗ trợ giao tiếp hai chiều theo hướng truyền tin, message-oriented IPC. Có thể hoạt động thông qua mạng theo mô hình server/client. Named Pipe cho phép các process đang kết nối dùng quyền hạn của nó trên remote servers.

Named Pipe có thể mở kết nối từ server tới một hay nhiều pipe client. Tương ứng ta có **NamedPipeServerStream** và **NamedPipeClientStream**



1. Hàm CreateNamedPipe

Creates an instance of a named pipe and returns a handle for subsequent pipe operations. A named pipe server process uses this function either to create the first instance of a specific named pipe and establish its basic attributes or to create a new instance of an existing named pipe.

Cú pháp:

HANDLE CreateNamedPipe(

LPCTSTR  name,

DWORD  openMode,

DWORD  pipeMode,

DWORD  maxInstances,

DWORD outBufferSize,

DWORD   inBufferSize,

DWORD  defaultTimeOut,

SECURITY\_ATTRIBUTES \*secAtt

);

* **name :**

The unique pipe name. This string must have the following form:

\\.\pipe\pipename

The pipename part of the name can include any character other than a backslash, including numbers and special characters. The entire pipe name string can be up to 256 characters long. Pipe names are not case sensitive.

* **dwOpenMode*[in]***

The open mode. The function fails if dwOpenMode specifies anything other than 0 or the flags listed in the following tables.

This parameter must specify one of the following pipe access modes. The same mode must be specified for each instance of the pipe.

* **nMaxInstances*[in]***

The maximum number of instances that can be created for this pipe. The first instance of the pipe can specify this value; the same number must be specified for other instances of the pipe. Acceptable values are in the range 1 through PIPE\_UNLIMITED\_INSTANCES (255).

If this parameter is PIPE\_UNLIMITED\_INSTANCES, the number of pipe instances that can be created is limited only by the availability of system resources. If nMaxInstances is greater than PIPE\_UNLIMITED\_INSTANCES, the return value is INVALID\_HANDLE\_VALUE and [**GetLastError**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms679360(v=vs.85).aspx) returns ERROR\_INVALID\_PARAMETER.

* **nOutBufferSize*[in]***

The number of bytes to reserve for the output buffer. For a discussion on sizing named pipe buffers, see the following Remarks section.

* **nInBufferSize*[in]***

The number of bytes to reserve for the input buffer. For a discussion on sizing named pipe buffers, see the following Remarks section.

* **nDefaultTimeOut*[in]***

The default time-out value, in milliseconds, if the [**WaitNamedPipe**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365800(v=vs.85).aspx) function specifies NMPWAIT\_USE\_DEFAULT\_WAIT. Each instance of a named pipe must specify the same value.

A value of zero will result in a default time-out of 50 milliseconds.

* **lpSecurityAttributes*[in, optional]***

A pointer to a [**SECURITY\_ATTRIBUTES**](https://msdn.microsoft.com/en-us/library/windows/desktop/aa379560(v=vs.85).aspx) structure that specifies a security descriptor for the new named pipe and determines whether child processes can inherit the returned handle. If lpSecurityAttributes is**NULL**, the named pipe gets a default security descriptor and the handle cannot be inherited. The ACLs in the default security descriptor for a named pipe grant full control to the LocalSystem account, administrators, and the creator owner. They also grant read access to members of the Everyone group and the anonymous account.

If the function succeeds, the return value is a handle to the server end of a named pipe instance.

If the function fails, the return value is INVALID\_HANDLE\_VALUE. To get extended error information, call[**GetLastError**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms679360(v=vs.85).aspx).

### Xử lí ngoại lệ

1. Windows Structured Exception Handling

Nhìn chung, ngoại lệ chia làm 2 loại:

* Ngoại lệ phần cứng: phát sinh bởi CPU như lỗi chia zero, nỗ lực truy xuất địa chỉ không hợp lệ.
* Ngoại lệ phần mềm: phát sinh bởi ứng dụng hay hệ điều hành như tham số không hợp lệ.

Windows Structured Exception Handling (SEH) là cơ chế dùng cho xử lí ngoại lệ phần cứng và phần mềm.

Vectored exception handling (VEH) là cơ chế mở rộng của SEH.

SHE có thể chỉ định 1 termination code được thực thi khi đoạn code được bảo vệ thực thi xong. Termination code được thực thi bất chấp ngoại lệ xảy ra hay không.

Cơ chế này cho phép lập trình viên xử lí ngoại lệ, phục hồi trạng thái hệ thống, đảm bảo tài nguyên được giải phóng,… dù có hay không xảy ra ngoại lệ.

Thông thường ngoại lệ xảy ra khi có lỗi và nó không tự phục hồi trạng thái hệ thống. Thread hay process sẽ ngắt đột ngột sau khi handler báo xảy ra ngoại lệ. SEH sẽ đảm bảo ứng dụng tiếp tục thực thi khi xảy ra ngoại lệ.

SHE được hỗ trợ bởi nhiều Window API, trình biên dịch và run-time. Ngôn ngữ hỗ trợ là rất nhiều, trong đó có Microsoft C.

Cơ chế SHE và termination handler được tích hợp vào hệ thống, ứng dụng có thể dùng để nâng cao độ tin cậy.

SHE giúp tối ưu, đơn giản code.

1. Khối try, except

Trước tiên chúng khai báo thư viện: <excpt.h>

Trình biên dịch Microsoft C/C++ hỗ trợ từ khóa :

* \_\_try : cho code block bảo vệ
* \_\_except : cho code block xử lí ngoại lệ.

Cú pháp:

\_\_try

{

// guarded code

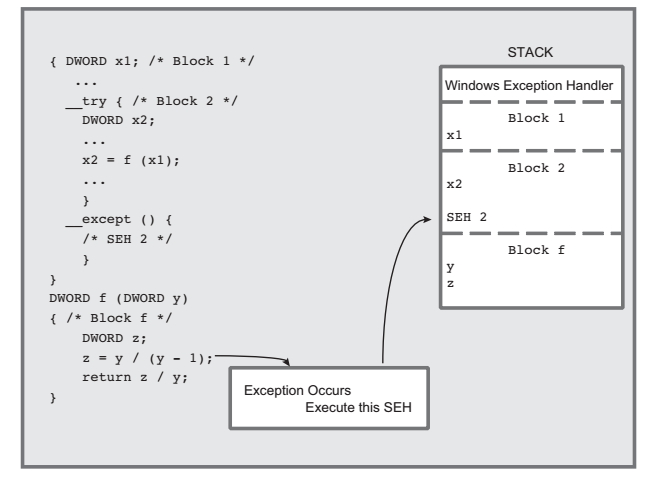
}

\_\_except (expression)

{

// exception handler code

}

Ngoại lệ xảy ra theo cơ chế unwinding stack tương tự như C++.

Trong ví dụ bên dưới, ngoại lệ chia zero xảy ra trong hàm f() và được trả về cho Block 2.

Block 2 có exception handler, quyền thực thi sẽ nhảy xuống SEH 2.

1. Khối try, finally

Termination handler đảm bảo termination code phải thực thi khi tiến trình đi ra khỏi guard code dù có xảy ra ngoại lệ.

Cơ chế termination handler được khai báo với 2 từ khóa:

* \_\_try : khối code bảo vệ
* \_\_finally: khối code thực thi dù xảy ra ngoại lệ hay không

Cú pháp:

\_\_try

{

// guarded body of code

}

\_\_finally

{

// \_\_finally block

}

Tuy nhiên, finally{} sẽ không được thực thi nếu try{} gọi ExitProcess, ExitThread hay abort.

Finally{} có thể truy xuất biến cục bộ.

Trường hợp xảy ra ngoại lệ trong code bảo vệ, finally{} sẽ được thực thi trước khi nhảy tới \_except{}.

Tương tự, việc gọi longjmp từ khối try{} cũng sẽ thực thi finally{} trước khi nhảy tới \_except{}.

Nếu khối try{} bị ngắt bởi lệnh điều khiển thì finally{} vẫn thực thi.

Phương thức AbnormalTermination có thể gọi trong finally{} để xác định try{} có thoát bình thường hay không bình thường khi xảy ra ngoại lệ, gọi longjmp, lệnh điều khiển (Khá tốn hiệu suất vì phải duyệt stack)

1. Tham số \_\_except(expression)

Tham số của mệnh đề except() thường là hằng, dùng để xác định kiểu ngoại lệ nào xảy ra, là một trong 3 giá trị sau:

* EXCEPTION\_CONTINUE\_EXECUTION (–1) : Ngoại lệ bị bỏ qua, tiếp tục thực thi tại điểm xảy ra ngoại lệ.
* EXCEPTION\_CONTINUE\_SEARCH (0) : Kiểu ngoại lệ không nhận diện được, tiếp tục duyệt stack, để tìm bộ xử lí phù hợp.
* EXCEPTION\_EXECUTE\_HANDLER (1) : Ngoại lệ được nhận diện. Chuyển quyền kiểm soát tới xử lí ngoại lệ, tiếp tục thực thi sau khối code except{}

## Lập trình nâng cao

### Hooking Windows API

##### Cấu trúc file thực thi

1. Định dạng File thực thi
2. MS-DOS COM Files

File COM là dạng thực thi đơn giản từ thập niên 1970s, .com dùng như phần mở rộng cho file văn bản chứa các dòng lệnh gọi bởi hệ thống (tương tự như file .bat).

COM file được nạp vào RAM theo đúng cách chúng nằm trên đĩa, đây là cách map file theo mô hình phân đoạn vùng nhớ từ trước đời x86.

2 thanh ghi 16-bits xác định địa chỉ thực sự được dùng để truy xuất vùng nhớ, một segment thanh ghi qui định một ô 64KB thành vùng 1MB + 64KB (tăng dần 16-byte) và một offset để map với offset của ô đó.

Offset thanh ghi dùng cho file COM, chỉ có 16-bits, do đó file COM không thể thay đổi segment thanh ghi, giới hạn chỉ dùng được 64KB RAM.

Ưu điểm là tốn ít chi phí khi DOS nạp và chạy file COM: load file COM – set segment thanh ghi – jump tới nó.

File COM nạp vào offset $100 của RAM, vùng nhớ phía trước được dùng để truyền dữ liệu đi vào hay đi ra từ DOS (ví dụ như nội dung của command line để gọi ứng dụng).

File COM không thể là 32-bit.

1. MS-DOS EXE Files

Trình biên dịch MS-DOS chỉ có 64K bộ nhớ tối đa, lý thuyết cơ bản là thiết lập các đoạn thanh ghi trong x86 CPU (CS,DS,ES,SS) để trỏ tới cùng hay các đoạn khác nhau, cho phép nhiều cấp bậc khi truy xuất vùng nhớ.

Mô hình chuẩn vùng nhớ sẽ gồm:

* Tiny : Dùng địa chỉ 16-bits (phân đoạn thanh ghi không đổi). Dùng thực thi file .COM.
* Small: Dùng địa chỉ 16-bit (phân đoạn thanh ghi không đổi).
* Compact: Địa chỉ dữ liệu gồm cả phân đoạn thanh ghi và offset, tái nạp thanh ghi DS hay ES khi truy xuất và cho phép tới 1MB dữ liệu. Truy xuất Code không làm thay đổi thanh ghi CS, cho phép 64KB Code.
* Medium: Địa chỉ Code bao gồm địa chỉ phân đoạn thanh ghi, tái nạp CS khi truy xuất và cho phép tới 1MB Code. Truy xuất dữ liệu không làm thay đổi thanh ghi DS và ES, cho phép 64K dữ liệu.
* Large: Cả địa chỉ Code và Data được cặp nhau (phân đoạn thanh ghi, offset). 1MB sẽ dành cho cả Code và Data.
* Huge: Tương tự Large, nhưng cho phép truy xuất mảng lớn hơn 64KB.

Khi nhìn vào file EXE, điều quan trọng là cần biết mô hình vùng nhớ nào dùng để tạo file này.

1. PE Files

File thực thi PE (Portable Executable) là chuẩn file nhị phân của Window.

* Thư viện Win32 SDK <winnt.h>, chứa các định nghĩa cấu trúc của file PE.
* Một vài các hàm dùng để xử lí file PE được include trong imagehlp.dll

1. Địa chỉ RVA

Trên môi trường Window, các module thực thi (DLL) được nạp tại bất kì điểm nào trong vùng nhớ.

RVA sẽ xác định địa chỉ cơ sở của module được nạp vào vùng nhớ, vốn không biết tại thời điểm biên dịch. Nên file PE mô tả vị trí của Data trong vùng nhớ như một offset từ địa chỉ cơ sở.

Một vài chỉ thị xử lí yêu cầu Code phải nhận dạng vị trí vùng nhớ mà Data nằm trên đó. Điều này là không thể khi vị trí của module trong vùng nhớ chưa xác định được khi biên dịch.

Rất nhiều trường trên file PE phải định vị dùng RVA. RVA là offset-byte vị trí trên file. Ví dụ, bộ nạp file của Windows map file PE vào vùng địa chỉ ảo bắt đầu từ địa chỉ 0x1000. Nếu có một bảng entry trên image file bắt đầu tại địa chỉ 0x10464 trong bộ nhớ, thì địa chỉ RVA của bảng entry này sẽ là 0x464.

**(Virtual address 0x10464)-(base address 0x10000) = RVA 0x00464**

Để chuyển RVA sang con trỏ dùng được, đơn giản là cộng RVA với địa chỉ cơ sở của module.

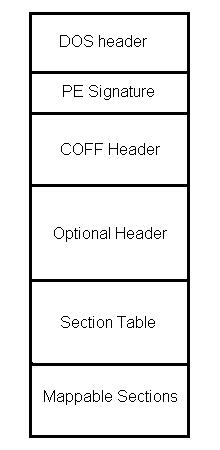
Địa chỉ cơ sở là đia chỉ bắt đầu của vùng nhớ đã map file thực thi EXE hay DLL, rất quan trọng trong Win32.

Window NT sẽ tính địa chỉ cơ sở dưới dạng instance handle (**HINSTANCE**), chúng ta có thể dùng **GetModuleHandle**() cho bất kì file DLL nào mà process có sử dụng để lấy con trỏ truy xuất tới các thành phần của module.

Lưu ý: Địa chỉ lấy từ disassembly của module sẽ không luôn khớp với địa chỉ nhìn thấy khi debug ứng dụng.

1. Định dạng file PE

Định dạng file thực thi PE bao gồm các header sắp xếp như bên dưới:

[](https://commons.wikimedia.org/wiki/File:RevEngPEFile.JPG)

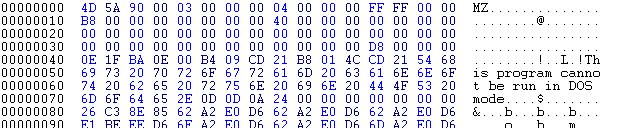
**Định dạng cơ bản của Microsoft PE file**

1. MS-DOS header

Trên bất kì file thực thi nhị phân Win32 nào, bạn sẽ thấy 2 kí tự đầu tiên là “MZ”, đặt theo tên của Mark Zbikowski, người tạo ra liên kết đầu tiên cho DOS.

Vài người xem đây là magic number, chúng ta có thể xem đó là “ID Tag” của file hay nhận dạng của file.

Sau file ID, sẽ là các kí tự ngẫu nhiên hay khoảng trắng, cho tới chuỗi kí tự "This program cannot be run in DOS mode".

[](https://commons.wikimedia.org/wiki/File:RevEngDosHead.JPG)

**MS-DOS file header**

Để đảm bảo tính tương thích về trước, hay các chuẩn file mới, Microsoft đã viết ra chuỗi mã máy vào đầu mỗi file PE.

Khi Window file 32-bit chạy trên môi trường DOS 16-bit, ứng dụng sẽ hiển thị "This program cannot be run in DOS mode." và ngắt.

DOS header còn được xem là EXE header. Cấu trúc trên C:

struct DOS\_Header

{

// short is 2 bytes, long is 4 bytes

char signature[2] = "MZ";

short lastsize;

short nblocks;

short nreloc;

short hdrsize;

short minalloc;

short maxalloc;

void \*ss;

void \*sp;

short checksum;

void \*ip;

void \*cs;

short relocpos;

short noverlay;

short reserved1[4];

short oem\_id;

short oem\_info;

short reserved2[10];

long e\_lfanew;

}

Sau DOS header là chương trình nhỏ, mã thực thi khi biên dịch trên GCC:

;# Using NASM with Intel syntax

push cs ;# Push CS onto the stack

pop ds ;# Set DS to CS

mov dx,message ; point to our message "This program cannot be run in DOS mode.", 0x0d, 0x0d, 0x0a, '$'

mov ah, 09

int 0x21 ;# when AH = 9, DOS interrupt to write a string

;# terminate the program

mov ax,0x4c01

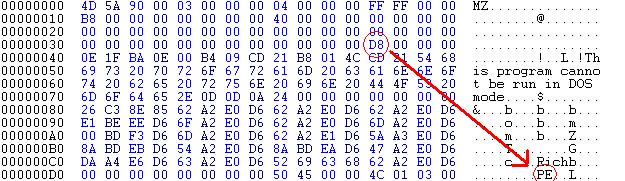
int 0x21

message db "This program cannot be run in DOS mode.", 0x0d, 0x0d, 0x0a, '$'

1. PE header

Offset thứ 60 (0x3C) trên DOS header sẽ là con trỏ trỏ tới PE header, thành viên e\_lfanew trong cấu trúc DOS\_header.

Khi DOS in tin nhắn lỗi môi trường thực thi và ngắt, Window vẫn theo con trỏ này tới thông tin kế tiếp.

[](https://commons.wikimedia.org/wiki/File:RevEngPeSig.JPG)

**PE signature, và con trỏ tới nó**

PE hader chỉ chứa ID nhận dạng file, với giá trị là “PE\0\0”, PE và 2 kí tự ASCII NULL. Ký hiệu này ám chỉ rằng đây là file PE và byte-order của file, mặc định là “little-Endian”.

Chuỗi thông tin lớn phía sau thuộc về COFF header.

1. COFF Header

COFF header hiện diện trên cả COFF object file (trước khi liên kết) và trên PE file. COFF header cung cấp thông tin liên quan cho việc thực thi và liên quan tới file object.

Cấu trúc COFF header trên C:

struct COFFHeader

{

short Machine;

short NumberOfSections;

long TimeDateStamp;

long PointerToSymbolTable;

long NumberOfSymbols;

short SizeOfOptionalHeader;

short Characteristics;

}

Chúng ta sẽ đi tìm hiểu từng trường của COFFHeader

* Machine: Xác định hệ máy mà file có thể chạy. Giá trị hex 0x14C (322d) là code của Intel 80386.

Danh sách các giá trị và ý nghĩa:

|  |  |
| --- | --- |
| **Value** | **Description** |
| 0x14c | Intel 386 |
| 0x8664 | x64 |
| 0x162 | MIPS R3000 |
| 0x168 | MIPS R10000 |
| 0x169 | MIPS little endian WCI v2 |
| 0x183 | old Alpha AXP |
| 0x184 | Alpha AXP |
| 0x1a2 | Hitachi SH3 |
| 0x1a3 | Hitachi SH3 DSP |
| 0x1a6 | Hitachi SH4 |
| 0x1a8 | Hitachi SH5 |
| 0x1c0 | ARM little endian |
| 0x1c2 | Thumb |
| 0x1d3 | Matsushita AM33 |
| 0x1f0 | PowerPC little endian |
| 0x1f1 | PowerPC with floating point support |
| 0x200 | Intel IA64 |
| 0x266 | MIPS16 |
| 0x268 | Motorola 68000 series |
| 0x284 | Alpha AXP 64-bit |
| 0x366 | MIPS with FPU |
| 0x466 | MIPS16 with FPU |
| 0xebc | EFI Byte Code |
| 0x8664 | AMD AMD64 |
| 0x9041 | Mitsubishi M32R little endian |
| 0xc0ee | clr pure MSIL |

* NumberOfSections : Tổng số section mô tả tại phần kết của PE header
* TimeDateStamp : số thời gian 32-bit khi header được tạo, dùng cho xử lí Binding
* SizeOfOptionalHeader : kích thước của PE Optional Header, nằm sau COFF header.
* Characteristics : kiểu bit flag, mô tả đặc tính của file:

|  |  |
| --- | --- |
| **Value** | **Description** |
| 0x02 | File thực thi |
| 0x200 | File dạng non-relocatable (Địa chỉ tuyệt đối, không phải RVA) |
| 0x2000 | Là file DLL , không phải EXE |

1. PE Optional Header

PE Optional Header nằm ngay sau COFF header, là Header phụ quan trọng cho file thực thi, hơn là cho COFF object file.

PE Optional Header chứa nhiều thông tin dùng để lấy cấu trúc file và cách dùng chúng.

Chúng ta sẽ tách header này làm 2 phần cho tiện, cấu trúc PE Optional Header trên C:

#define NUMBEROFRVAANDSIZES 16

struct data\_directory

{

long VirtualAddress;

long Size;

};

struct PEOptHeader

{

short signature; // 267d cho hệ 32 bit, 523d cho 64 bit, và 263d cho ROM image.

char MajorLinkerVersion;

char MinorLinkerVersion;

long SizeOfCode;

long SizeOfInitializedData;

long SizeOfUninitializedData;

long AddressOfEntryPoint; // RVA của entry point CODE

long BaseOfCode;

long BaseOfData;

/\* 21 trường mở rộng của COFF optional header\*/

long ImageBase;

long SectionAlignment;

long FileAlignment;

short MajorOSVersion;

short MinorOSVersion;

short MajorImageVersion;

short MinorImageVersion;

short MajorSubsystemVersion;

short MinorSubsystemVersion;

long Win32VersionValue;

long SizeOfImage;

long SizeOfHeaders;

long Checksum;

short Subsystem;

short DLLCharacteristics;

long SizeOfStackReserve;

long SizeOfStackCommit;

long SizeOfHeapReserve;

long SizeOfHeapCommit;

long LoaderFlags;

long NumberOfRvaAndSizes; // = NUMBEROFRVAANDSIZES

data\_directory DataDirectory[NUMBEROFRVAANDSIZES];//Số phần tử luôn = trường NumberOfRvaAndSizes.

}; //luôn là 16 trên PE files.

* Signature : Chứa ID định dạng file

|  |  |  |
| --- | --- | --- |
| **Constant Name** | **Value** | **Description** |
| IMAGE\_NT\_OPTIONAL\_HDR32\_MAGIC | 0x10b | 32 bit executable image. |
| IMAGE\_NT\_OPTIONAL\_HDR64\_MAGIC | 0x20b | 64 bit executable image |
| IMAGE\_ROM\_OPTIONAL\_HDR\_MAGIC | 0x107 | ROM image |

* MajorLinkerVersion : Số phiên bản chính của liên kết
* MinorLinkerVersion : Số phiên bản phụ của liên kết
* SizeOfCode : kích thước vùng CODE, theo byte,hoặc tổng số bytes nếu chia ra nhiều vùng.
* SizeOfInitializedData : kích thước vùng khởi tạo Data theo bytes, hoặc tổng số bytes nếu chia ra nhiều vùng
* SizeOfUninitializedData : kích thước vùng Data chưa khởi tạo, ,hoặc tổng số bytes nếu chia ra nhiều vùng.
* AddressOfEntryPoint : Chứa địa chỉ offset trỏ tới entry point con trỏ hàm, liên quan tới địa chỉ cơ sở.  
  Với file thực thi, đây là địa chỉ bắt đầu.  
  Với file driver, đây là địa chỉ hàm khởi tạo.  
  Entry point của hàm dành cho file DLLs, nếu không có entry point, trường này là zero.
* BaseOfCode : Offset trỏ tới phần đầu của vùng CODE, tính theo RVA
* BaseOfData : Offset trỏ tới phần đầu của vùng DATA, tính theo RVA
* ImageBase : Địa chỉ RVA khi file được nạp vào vùng nhớ. Luôn là bộ số của 64KB. Mặc định cho file thực thi Window NT là 0x10000, với file DLLS và ứng dụng mặc định là 0x00400000. Ứng dụng trên Window CE là 0x10000
* SectionAlignment : Giá trị căn cho các section file thực thi khi nạp vào vùng nhớ, tính theo bytes. Giá trị này phải lớn hơn hay bằng trường FileAlignment. Giá trị mặc định là kích thước page cho hệ thống.
* FileAlignment : Giá trị căn cho dữ liệu gốc của các section trong image file, tính theo bytes. Giá trị này luôn là lũy thừa của 2, nằm giữa 512 và 64KB. Mặc định là 512, nếu **SectionAlignment** nhỏ hơn kích thước page hệ thống, trường này phải cùng giá trị với **SectionAlignment**
* MajorOSVersion :
* MinorOSVersion :

2 trường qui định số phiên bản thấp nhất của hệ điều hành mà file có thể thực thi. Mặc định là 1.0

* MajorImageVersion :
* MinorImageVersion :

2 trường do người dùng định nghĩa, số phiên bản của file EXE hay DLL. Có thể set khi dùng Visual Studio: linker-> Version.

* MajorSubsystemVersion :
* MinorSubsystemVersion :

2 trường qui định phiên bản hệ thống phụ thấp nhất mà file có thể thực thi. Mặc định là 3.10 (Window NT 3.1)

* Win32VersionValue : Trường để trống, zero
* SizeOfImage : Kích thước file theo byte, bao gồm tất cả các header, phải là bộ số của **SectionAlignment**
* SizeOfHeaders : kích thước byte của PE header và bảng Sections, gồm các trường sau:
* e\_lfanew (thuộc DOS\_Header)
* 4 byte file ID (PE\0\0)
* COFFHeader
* PE optional header
* Tất cả Section headers

Luôn là bộ số của **FileAlignment**

* CheckSum : Checksum của file thực thi, các loại file sau sẽ phải kiểm tra khi nạp: driver, DLLs tại boot time hay load bởi process hệ thống
* Subsystem : hệ thống phụ có thể chạy file thực thi.

|  |  |  |
| --- | --- | --- |
| **Constant Name** | **Value** | **Description** |
| **IMAGE\_SUBSYSTEM\_UNKNOWN** | 0 | Unknown subsystem |
| **IMAGE\_SUBSYSTEM\_NATIVE** | 1 | No subsystem required (device drivers and native system processes) |
| **IMAGE\_SUBSYSTEM\_WINDOWS\_GUI** | 2 | Windows graphical user interface (GUI) subsystem |
| **IMAGE\_SUBSYSTEM\_WINDOWS\_CUI** | 3 | Windows character-mode user interface (CUI) subsystem |
| **IMAGE\_SUBSYSTEM\_OS2\_CUI** | 5 | OS/2 CUI subsystem |
| **IMAGE\_SUBSYSTEM\_POSIX\_CUI** | 7 | POSIX CUI subsystem |
| **IMAGE\_SUBSYSTEM\_WINDOWS\_CE\_GUI** | 9 | Windows CE system |
| **IMAGE\_SUBSYSTEM\_EFI\_APPLICATION** | 10 | Extensible Firmware Interface (EFI) application |
| **IMAGE\_SUBSYSTEM\_EFI\_BOOT\_SERVICE\_DRIVER** | 11 | EFI driver with boot services |
| **IMAGE\_SUBSYSTEM\_EFI\_RUNTIME\_DRIVER** | 12 | EFI driver with run-time services |
| **IMAGE\_SUBSYSTEM\_EFI\_ROM** | 13 | EFI ROM image |
| **IMAGE\_SUBSYSTEM\_XBOX** | 14 | Xbox system |
| **IMAGE\_SUBSYSTEM\_WINDOWS\_BOOT\_APPLICATION** | 16 | Boot application |

* DLLCharacteristics : Đặc tính file DLL

|  |  |  |
| --- | --- | --- |
| **Constant Name** | **Value** | **Description** |
| No constant name | 0x0001 | Call when DLL is first loaded into a process's address space |
| No constant name | 0x0002 | Call when a thread terminates |
| No constant name | 0x0004 | Call when a thread starts up |
| No constant name | 0x0008 | Call when DLL exits |
| IMAGE\_DLLCHARACTERISTICS\_DYNAMIC\_BASE | 0x0040 | DLL có thể relocated khi nạp |
| IMAGE\_DLLCHARACTERISTICS\_FORCE\_INTEGRITY | 0x0080 | Bắt buộc phải kiểm tra checksum |
| IMAGE\_DLLCHARACTERISTICS\_NX\_COMPAT | 0x0100 | The image is compatible with data execution prevention (DEP) |
| IMAGE\_DLLCHARACTERISTICS\_NO\_ISOLATION | 0x0200 | The image is isolation aware, but should not be isolated |
| IMAGE\_DLLCHARACTERISTICS\_NO\_SEH | 0x0400 | The image does not use structured exception handling (SEH). No handlers can be called in this image |
| IMAGE\_DLLCHARACTERISTICS\_NO\_BIND | 0x0800 | Do not bind the image |
| No constant name | 0x1000 | Reserved |
| IMAGE\_DLLCHARACTERISTICS\_WDM\_DRIVER | 0x2000 | A WDM driver |
| No constant name | 0x4000 | Reserved |
| IMAGE\_DLLCHARACTERISTICS\_TERMINAL\_SERVER\_AWARE | 0x8000 | The image is terminal server aware |

* SizeOfStackReserve :kích thước byte dữ trữ của vùng nhớ ảo dành cho stack. Mặc định là 0x100000 (1MB). Nhưng chỉ có lượng byte **SizeOfStackCommit** là cấp phát lúc nạp mà thôi. Phần còn lại sẽ cấp từng page một khi stack đầy.
* SizeOfStackCommit : Kích thước byte dành cho stack. Mặc định là 0x1000-byte (1 page)
* SizeOfHeapReserve : Kích thước byte dự trữ của vùng nhớ ảo cho local heap. Chỉ có lượng byte **SizeOfHeapCommit** được cấp lúc nạp. Phần còn lại sẽ cấp từng page một khi heap cạn.
* SizeOfHeapCommit : Kích thước byte dành cho heap. Mặc định là 0x1000-byte (1 page)
* LoaderFlags : ít được dùng, theo WINNT.H, thì chỉ dùng cho debug:

|  |  |
| --- | --- |
| 1 | Gọi breakpoint trước khi bắt đầu process |
| 2 | Gọi debugger cho process sau khi nó được nạp |

* NumberOfRvaAndSizes : Số entry của mảng DataDirectory. Mặc định luôn là 16.

The number of directory entries in the remainder of the optional header. Each entry describes a location and size.

* DataDirectory : Cung cấp RVAs và kích thước của nhiều cấu trúc dữ liệu, dùng cho thiết lập môi trường thực thi của module.  
  Entry quan trọng nhất trong DataDirectory là   
  **Exprot Directory   
  Import Directory  
  Resource Directory,   
  Bound Import directory**Offse byte sẽ tính từ bắt đầu của PE optional header.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Constant Name** | **Value** | **Description** | **Offset PE(32 bit)** | **Offset PE32+(64 bit)** |
| IMAGE\_DIRECTORY\_ENTRY\_EXPORT | 0 | Export Directory | 96 | 112 |
| IMAGE\_DIRECTORY\_ENTRY\_IMPORT | 1 | Import Directory | 104 | 120 |
| IMAGE\_DIRECTORY\_ENTRY\_RESOURCE | 2 | Resource Directory | 112 | 128 |
| IMAGE\_DIRECTORY\_ENTRY\_EXCEPTION | 3 | Exception Directory | 120 | 136 |
| IMAGE\_DIRECTORY\_ENTRY\_SECURITY | 4 | Security Directory | 128 | 144 |
| IMAGE\_DIRECTORY\_ENTRY\_BASERELOC | 5 | Base Relocation Table | 136 | 152 |
| IMAGE\_DIRECTORY\_ENTRY\_DEBUG | 6 | Debug Directory | 144 | 160 |
| IMAGE\_DIRECTORY\_ENTRY\_ARCHITECTURE | 7 | Architecture specific data | 152 | 168 |
| IMAGE\_DIRECTORY\_ENTRY\_GLOBALPTR | 8 | Global pointer register relative virtual address | 160 | 176 |
| IMAGE\_DIRECTORY\_ENTRY\_TLS | 9 | Thread Local Storage directory | 168 | 184 |
| IMAGE\_DIRECTORY\_ENTRY\_LOAD\_CONFIG | 10 | Load Configuration directory | 176 | 192 |
| IMAGE\_DIRECTORY\_ENTRY\_BOUND\_IMPORT | 11 | Bound Import directory | 184 | 200 |
| IMAGE\_DIRECTORY\_ENTRY\_IAT | 12 | Import Address Table | 192 | 208 |
| IMAGE\_DIRECTORY\_ENTRY\_DELAY\_IMPORT | 13 | Delay Import table | 200 | 216 |
| IMAGE\_DIRECTORY\_ENTRY\_COM\_DESCRIPTOR | 14 | COM descriptor table | 208 | 224 |
| No constant name | 15 | Reserved | 216 | 232 |

1. Bảng Section

COFF Header định nghĩa số section trong file thực thi, mỗi section dài 40-bytes. Ví dụ từ một ứng dụng

2E746578\_74000000\_00100000\_00100000\_A8050000 .text

00040000\_00000000\_00000000\_00000000\_20000000

2E646174\_61000000\_00100000\_00200000\_86050000 .data

000A0000\_00000000\_00000000\_00000000\_40000000

2E627373\_00000000\_00200000\_00300000\_00000000 .bss

00000000\_00000000\_00000000\_00000000\_80000000

Cấu trúc section được mô tả bên dưới:

|  |  |  |
| --- | --- | --- |
| **Offset** | **Độ dài** | **Mục đích** |
| 0x00 | 8 bytes | Tên Section – Ví dụ .text .data .bss |
| 0x08 | 4 bytes | Kích thước Section khi nó nạp vào vùng nhớ |
| 0x0C | 4 bytes | RVA (Vị trí) của Section khi nó nạp vào vùng nhớ |
| 0x10 | 4 bytes | Kích thước vậy lý của section trên đĩa cứng |
| 0x14 | 4 bytes | Địa chỉ vật lý của section trên đĩa cứng (bắt đầu từ disk imaged) |
| 0x18 | 12 bytes | Dành riêng (thường là zero) (dùng trong định dạng objects) |
| 0x24 | 4 bytes | Section flags |

Bộ nạp file PE sẽ sắp xếp các sections tính theo RVAs và thường căn theo 0x1000, 1 page hệ thống trên hệ x86

Common sections gồm:

* .text/.code/CODE/TEXT – chứa mã thực thi (mã máy)
* .textbss/TEXTBSS - Present if Incremental Linking is enabled
* .data/.idata/DATA/IDATA – Chứa dữ liệu khởi tạo
* .bss/BSS – Chứa dữ liệu chưa khởi tạo

1. Section flag

Section flags là trường 32-bit, định nghĩa trong winnt.h gồm:

#define IMAGE\_SCN\_TYPE\_NO\_PAD 0x00000008 // Reserved.

#define IMAGE\_SCN\_CNT\_CODE 0x00000020 // Section contains code.

#define IMAGE\_SCN\_CNT\_INITIALIZED\_DATA 0x00000040 // Section contains initialized data.

#define IMAGE\_SCN\_CNT\_UNINITIALIZED\_DATA 0x00000080 // Section contains uninitialized data.

#define IMAGE\_SCN\_LNK\_OTHER 0x00000100 // Reserved.

#define IMAGE\_SCN\_LNK\_INFO 0x00000200 // Section contains comments or some other type of information.

#define IMAGE\_SCN\_LNK\_REMOVE 0x00000800 // Section contents will not become part of image.

#define IMAGE\_SCN\_LNK\_COMDAT 0x00001000 // Section contents comdat.

#define IMAGE\_SCN\_NO\_DEFER\_SPEC\_EXC 0x00004000 // Reset speculative exceptions handling bits in the TLB entries for this section.

#define IMAGE\_SCN\_GPREL 0x00008000 // Section content can be accessed relative to GP

#define IMAGE\_SCN\_MEM\_FARDATA 0x00008000

#define IMAGE\_SCN\_MEM\_PURGEABLE 0x00020000

#define IMAGE\_SCN\_MEM\_16BIT 0x00020000

#define IMAGE\_SCN\_MEM\_LOCKED 0x00040000

#define IMAGE\_SCN\_MEM\_PRELOAD 0x00080000

#define IMAGE\_SCN\_ALIGN\_1BYTES 0x00100000 //

#define IMAGE\_SCN\_ALIGN\_2BYTES 0x00200000 //

#define IMAGE\_SCN\_ALIGN\_4BYTES 0x00300000 //

#define IMAGE\_SCN\_ALIGN\_8BYTES 0x00400000 //

#define IMAGE\_SCN\_ALIGN\_16BYTES 0x00500000 // Default alignment if no others are specified.

#define IMAGE\_SCN\_ALIGN\_32BYTES 0x00600000 //

#define IMAGE\_SCN\_ALIGN\_64BYTES 0x00700000 //

#define IMAGE\_SCN\_ALIGN\_128BYTES 0x00800000 //

#define IMAGE\_SCN\_ALIGN\_256BYTES 0x00900000 //

#define IMAGE\_SCN\_ALIGN\_512BYTES 0x00A00000 //

#define IMAGE\_SCN\_ALIGN\_1024BYTES 0x00B00000 //

#define IMAGE\_SCN\_ALIGN\_2048BYTES 0x00C00000 //

#define IMAGE\_SCN\_ALIGN\_4096BYTES 0x00D00000 //

#define IMAGE\_SCN\_ALIGN\_8192BYTES 0x00E00000 //

#define IMAGE\_SCN\_ALIGN\_MASK 0x00F00000

#define IMAGE\_SCN\_LNK\_NRELOC\_OVFL 0x01000000 // Section contains extended relocations.

#define IMAGE\_SCN\_MEM\_DISCARDABLE 0x02000000 // Section can be discarded.

#define IMAGE\_SCN\_MEM\_NOT\_CACHED 0x04000000 // Section is not cachable.

#define IMAGE\_SCN\_MEM\_NOT\_PAGED 0x08000000 // Section is not pageable.

#define IMAGE\_SCN\_MEM\_SHARED 0x10000000 // Section is shareable.

#define IMAGE\_SCN\_MEM\_EXECUTE 0x20000000 // Section is executable.

#define IMAGE\_SCN\_MEM\_READ 0x40000000 // Section is readable.

#define IMAGE\_SCN\_MEM\_WRITE 0x80000000 // Section is writeable.

1. Liên kết tới module khác

Khi viết ứng dụng, chúng ta cần gọi tới các phương thức phụ để xử lí, tức cần liên kết tới các thư viện code. Trình liên kết sẽ quyết định xử lí tiếp theo.

Hai dạng liên kết đó là static và dynamic.

* Static là thư viện các hàm đã biên dịch: các code đã biên dịch có thể chèn vào file thực thi để triển khai hàm, giúp tiết kiệm thời gian phát triển
* Dynamic link cho phép code subroutine nằm trên một file khác (module khác), sẽ được nạp tại thời điểm runtime của OS. Nó được gọi là DLL, thư viện liên kết động. File DLL chứa hàng loạt các phương thức hay dữ liệu có thể export.

Trong khi file thực thi sẽ import thư viện để dùng. Từ module ở đây đại diện cho bất kì file có định dạng PE, và “thư viện” là bất kì module nào exports và import hàm, dữ liệu.

Ưu điểm liên kết động:

* Tiết kiệm vùng nhớ đĩa cứng, có thể chia sẽ cho bất kì file thực thi nào cần chúng.
* Cho phép các phương thức cập nhật lên phiên bản mới mà không cần tái khởi động file thực thi
* Tiết kiệm vùng nhớ vật lý qua cơ chế map Code trong thư viện vào process.
* Tính đa hình: mỗi phương thức có thể điều chỉnh mà không cần lập trình lại ứng dụng, khả năng này rất quan trọng để tương thích ngược với OS.

Lưu ý rằng tất cả hàm, dữ liệu import hay export giữa các modules đều được coi là subroutines. Mỗi module (thực thi hay thư viện) phải khai báo hàm hay dữ liệu nào mà nó exports tới modules khác, và những hàm, dữ liệu nào imports từ modules khác

Điểm yếu của liên kết động là tốn thời gian liên kết tại thời điểm runtime. Chúng ta không thể dám chắc rằng một phương thức trong DLL luôn tồn tại trong địa chỉ cố định của vùng nhớ. Nếu địa chỉ đó không sẵn sàng hay thư viện đã được update, thì nó đã di chuyển ra chỗ khác, khiến ứng dụng có thể bị crash.

Để khắc phục, các module được thiết kế bảng địa chỉ hàm, giúp chúng ta định vị địa chỉ bất kì hàm, dữ liệu cần import dù nó đã được nạp vào vùng nhớ.

1. Exports
2. Bảng Export directory

Các hàm và dữ liệu muốn chia sẻ với module khác phải được khai báo trong bảng “Export Directory”, được dùng để map giữa tên export (hay “Ordinal”), với vị trí trong vùng nhớ, nơi mà code hay dữ liệu có thể tìm thấy.

Offset của export directory được định vị bởi IMAGE\_DIRECTORY\_ENTRY\_EXPORT trong trường DataDirectory.

Tất cả dữ liệu export phải tồn tại cùng một section.

Cấu trúc Export directory:

struct IMAGE\_EXPORT\_DIRECTORY {

long Characteristics;

long TimeDateStamp;

short MajorVersion;

short MinorVersion;

long Name;

long Base;

long NumberOfFunctions;

long NumberOfNames;

long \*AddressOfFunctions;

long \*AddressOfNames;

long \*AddressOfNameOrdinals;

};

* **Characteristics:** sẽ không dùng
* **TimeDateStamp** là thời điểm export directory được tạo
* **MajorVersion** và **MinorVersion** là số phiên bản của direcotry, mặc định chúng không được định nghĩa. Các giá trị này ít ảnh hưởng tới bảng exports.
* **Name:** là RVA trỏ tới chuỗi kí tự ASCII có “\0”, chỉ tên module.
* **AddressOfFunctions**: nó là một RVA trỏ tới mảng các RVAs, mỗi phần tử trong mảng sẽ trỏ tới các hàm hay giá trị export.
* **NumberOfFunctions:** Cho biết kích thước mảng RVA trỏ bởi **AddressOfFunctions** .
* **Base:** được dùng như là oridinal của export đầu tiên, và RVA kế tiếp trong mảng sẽ là **Base** + 1, và tiếp tục như vậy.

Mỗi entry trong mảng **AddressOfFunctions** nhận dạng bởi tên export và “ordinal” (dạng index), phải tìm bằng địa chỉ RVA trong trường **AddressOfNames**.

* **AddressOfNames** trỏ tới mảng địa chỉ RVAs, số phần tử qui định trong **NumberOfNames**. Mỗi RVA trong mảng trỏ tới chuỗi ASCII có byte kết zero ‘\0’, mỗi chuỗi là tên export.
* **NumberOfNames:** số nguyên 16-bit chỉ số phần tử trong mảng trỏ bởi **AddressOfNames** cũng như **AddressOfNameOrdinals**
* **AddressOfNameOrdinals** : trỏ tới mảng địa chỉ RVAs có cùng số phần tử với mảng RVAs trỏ bởi **AddressOfNames**, mỗi giá trị sẽ là ordinal của một entry export

Hai mảng **AddressOfNames** và **AddressOfNameOrdinals** map qua lại với nhau cho từng entry export, và được dùng để lấy giá trị export từ **AddressOfFunctions**.

Để tìm export bằng tên, chúng ta tìm kiếm trong mảng **AddressOfNames**, và lấy giá trị tương ứng từ mảng **AddressOfNameOrdinals**

Giá trị lấy từ **AddressOfNameOrdinals** sau đó map qua **AddressOfFunctions** (index là zero-based, không phải base-biased ordinal như tài liệu chính thức công bố)

1. Chuyển tiếp Export

Ngoài khả năng export hàm và giá trị trong module, bảng export direcotry có thể chuyển tiếp export sang thư viện khác. Cho phép linh hoạt khi tái tổ thức thư viện thay vì chỉnh sửa module gốc.

Forwarding được thực hiện bởi RVA trong mảng **AddressOfFunctions,** trỏ tới section chứa export directory.

Tại vị trí này, phải là chuỗi kí tự ASCII ‘\0’ có dạng “LibraryName.ExportName” ứng với nơi để chuyển tiếp export tới.

1. Imports
2. Cơ chế

Trước thời điểm runtime, trình biên dịch và trình liên kết không biết nơi nào trong vùng nhớ có entry import tồn tại. Bảng import directory giải quyết bằng cách tạo mảng con trỏ tại runtime.

Mỗi con trỏ trỏ tới một địa chỉ trong vùng nhớ nơi chứa entry import.

Mảng con trỏ tồn tại bên trong module tại địa chỉ RVA xác định trước, với cách này trình liên kết có thể dùng địa chỉ nội bộ module để truy xuất giá trị bên ngoài.

1. Bảng Import directory

Phần đầu của bảng import directory được trỏ bởi hai entry **IMAGE\_DIRECTORY\_ENTRY\_IAT** và **IMAGE\_DIRECTORY\_ENTRY\_IMPORT** trong resource directory.

Tại đây sẽ tồn tại mảng các entry kiểu **IMAGE\_IMPORT\_DESCRIPTOR**, mỗi entry đại diện cho một thư viện hay module import tới.

Mảng sẽ kết thúc bởi entry có tất cả các trường là zero.

Cấu trúc entry **IMAGE\_IMPORT\_DESCRIPTOR:**

struct IMAGE\_IMPORT\_DESCRIPTOR {

long \*OriginalFirstThunk;

long TimeDateStamp;

long ForwarderChain;

long Name;

long \*FirstThunk;

};

* **TimeDateStamp** : có liên quan tới hoạt động "Binding"
* **Name:** là RVA trỏ tới chuỗi ASCII có ‘\0’, chỉ tên của thư viện import tới.

Đối với entry import nằm trong chuỗi forward export sẽ giải thích sau.

Điều quan trọng ở đây là RVAs **OriginalFirstThunk** và RVAs **FirstThunk**.

Cả hai đều trỏ tới mảng các RVAs, mỗi phần tử mảng trỏ tới cấu trúc **IMAGE\_IMPORT\_BY\_NAMES**, các mảng kết thúc bởi entry có các trường zero.

Hai mảng tồn tại song song và trỏ tới cùng cấu trúc, theo cùng thứ tự. Chúng ta sẽ tìm hiểu lý do sau.

Trước hết hãy xem cấu trúc **IMAGE\_IMPORT\_BY\_NAMES**:

struct IMAGE\_IMPORT\_BY\_NAME {

short Hint;

char Name[1];

};

* **Name:** chuỗi kí tự ASCII có ‘\0’, là tên imported. Dùng để tra cứu giá trị trong bảng export directory thông qua mảng **AddressOfNames**
* **Hint:** là giá trị index của mảng **AddressOfNames.** Để tiết kiệm thời gian tìm kiếm chuỗi, đầu tiên bộ nạp sẽ kiểm tra entry **AddressOfNames**, tương ứng với **Hint.**

Tóm lại: Bảng import sẽ chứa mảng nhận dạng **IMAGE\_IMPORT\_DESCRIPTORs**, kết thúc bởi cấu trúc full zero field. Những entry này tương ứng với một thư viện import.

Có hai mảng địa chỉ RVA song song, mỗi entry trong mảng trỏ tới cấu trúc **IMAGE\_IMPORT\_BY\_NAME,** đại diện cho giá trị được import.

1. Imports khi runtime

Bằng cách dùng bảng import khi runtime, bộ nạp tìm đúng module, nạp nó vào vùng nhớ và kiếm chính xác entry export.

Tuy nhiên, để dùng các entry export, con trỏ tới nó phải được thiết lập tại nơi nào đó trong vùng nhớ của module đang import. Đó là lý do tại sao phải dùng tới 2 mảng song song, **OriginalFirstThunk** và **FirstThunk**, để định vị cấu trúc **IMAGE\_IMPORT\_BY\_NAME**.

Khi entry import được phân giải, con trỏ tới nó nằm trên mảng **FirstThunk.** Con trỏ này được dùng khi ứng dụng đang chạy để xác định địa chỉ của entry import.

1. Liên kết import entry

Định dạng file PE hỗ trợ tính năng đặc biệt, đó là binding (liên kết). Quá trình nạp và phân giải địa chỉ import có thể tốn thời gian, và có thể tránh được trong vài trường hợp.

Nếu lập trình viên chắc chắn rằng thư viện không bị update hay thay đổi, thì địa chỉ trong vùng nhớ của giá trị import sẽ cố định mỗi khi ứng dụng được nạp. Do đó, địa chỉ import có thể tính toán lại và chứa trong **FirstThunk** trước thời điểm runtime, cho phép bộ nạp ngưng phân giải quá trình import – các entry import sẽ liên kết tới vùng nhớ nhất định.

Tuy nhiên, nếu phiên bản giữa các module không khớp, hya thư viện import cần tái phân bố, bộ nạp sẽ xem địa chỉ liên kết là không hợp lệ, và phân giải các import.

Thành viên "**TimeDateStamp**" của entry import trong bảng được dùng cho quá trình kiểm soát liên kết module.

* Nếu nó bị set zero, thì bảng import sẽ không được liên kết.
* Nếu nó khác zero, nó được liên kết tới module khác.

Tuy nhiên, **TimeDateStamp** trong bảng import phải khớp với thành viên **TimeDateStamp** trong Header của module đã liên kết, nếu không giá trị liên kết sẽ bị loại bỏ bởi bộ nạp.

1. Chuyển tiếp và liên kết

Quá trình liên kết có thể gặp vấn đề khi thư viện liên kết chuyển tiếp exports entry của nó sang module khác.

Các imports không chuyển tiếp có thể tìm thấy, nhưng entry imports chuyển tiếp phải được nhận diện để bộ nạp có thể phân giải chúng, nhờ thành viên **ForwarderChain** của cấu trúc **IMAGE\_IMPORT\_DESCRIPTOR**.

**ForwarderChain** là index của hai mảng song song **FirstThunk** và **OriginalFirstThunk**. **OriginalFirstThunk** cho **ForwarderChain** chứacác entry import kiểu cấu trúc **IMAGE\_IMPORT\_BY\_NAME** cần cho phân giải, và **FirstThunk** cho **ForwarderChain** sẽ chứa index của entry khác cần phân giải.

Vòng lặp tiếp tục khi **FirstThunk** = -1, tức không còn entry chuyển tiếp cần import nữa.

1. Resources
2. Resource structures

Resources are data items in modules which are difficult to be stored or described using the chosen programming language. This requires a separate compiler or resource builder, allowing insertion of dialog boxes, icons, menus, images, and other types of resources, including arbitrary binary data. A number of API calls can then be used to retrieve resources from the module. The base of resource data is pointed to by the IMAGE\_DIRECTORY\_ENTRY\_RESOURCE entry of the data directory, and at that location there is an IMAGE\_RESOURCE\_DIRECTORY structure:

struct IMAGE\_RESOURCE\_DIRECTORY

{

long Characteristics;

long TimeDateStamp;

short MajorVersion;

short MinorVersion;

short NumberOfNamedEntries;

short NumberOfIdEntries;

};

Characteristics is unused, and TimeDateStamp is normally the time of creation, although it doesn't matter if it's set or not. MajorVersion and MinorVersion relate to the versioning info of the resources: the fields have no defined values. Immediately following the IMAGE\_RESOURCE\_DIRECTORY structure is a series of IMAGE\_RESOURCE\_DIRECTORY\_ENTRYs, the number of which are defined by the total of NumberOfNamedEntries and NumberOfIdEntries. The first portion of these entries are for named resources, the latter for ID resources, depending on the values in the IMAGE\_RESOURCE\_DIRECTORY struct. The actual shape of the resource entry structure is as follows:

struct IMAGE\_RESOURCE\_DIRECTORY\_ENTRY

{

long NameId;

long \*Data;

};

The NameId value has dual purpose: if the most significant bit (or sign bit) is clear, then the lower 16 bits are an ID number of the resource. Alternatly, if the top bit is set, then the lower 31 bits make up an offset from the start of the resource data to the name string of this particular resource. The Data value also has a dual purpose: if the most significant bit is set, the remaining 31 bits form an offset from the start of the resource data to another IMAGE\_RESOURCE\_DIRECTORY (i.e. this entry is an interior node of the resource tree). Otherwise, this is a leaf node, and Data contains the offset from the start of the resource data to a structure which describes the specifics of the resource data itself (which can be considered to be an ordered stream of bytes):

struct IMAGE\_RESOURCE\_DATA\_ENTRY

{

long \*Data;

long Size;

long CodePage;

long Reserved;

};

The Data value contains an RVA to the actual resource data, Size is self-explanatory, and CodePage contains the Unicode codepage to be used for decoding Unicode-encoded strings in the resource (if any). Reserved should be set to 0.

1. Layout

The above system of resource directory and entries allows simple storage of resources, by name or ID number. However, this can get very complicated very quickly. Different types of resources, the resources themselves, and instances of resources in other languages can become muddled in just one directory of resources. For this reason, the resource directory has been given a structure to work by, allowing separation of the different resources.

For this purpose, the "Data" value of resource entries points at another IMAGE\_RESOURCE\_DIRECTORY structure, forming a tree-diagram like organisation of resources. The first level of resource entries identifies the type of the resource: cursors, bitmaps, icons and similar. They use the ID method of identifying the resource entries, of which there are twelve defined values in total. More user defined resource types can be added. Each of these resource entries points at a resource directory, naming the actual resources themselves. These can be of any name or value. These point at yet another resource directory, which uses ID numbers to distinguish languages, allowing different specific resources for systems using a different language. Finally, the entries in the language directory actually provide the offset to the resource data itself, the format of which is not defined by the PE specification and can be treated as an arbitrary stream of bytes.

1. Xử lí file DLL

LoadLibrary can be used to load a library module into the address space of the process and return a handle that can be used in GetProcAddress to get the address of a DLL function. LoadLibrary can also be used to load other executable modules. For example, the function can specify an .exe file to get a handle that can be used in FindResource or LoadResource. However, do not use LoadLibrary to run an .exe file. Instead, use the CreateProcess function.

If the specified module is a DLL that is not already loaded for the calling process, the system calls the DLL's DllMain function with the DLL\_PROCESS\_ATTACH value. If DllMain returns TRUE, LoadLibrary returns a handle to the module. If DllMain returns FALSE, the system unloads the DLL from the process address space and LoadLibrary returns NULL. It is not safe to call LoadLibrary from DllMain. For more information, see the Remarks section in DllMain.

Module handles are not global or inheritable. A call to LoadLibrary by one process does not produce a handle that another process can use — for example, in calling GetProcAddress. The other process must make its own call to LoadLibrary for the module before calling GetProcAddress.

If lpFileName does not include a path and there is more than one loaded module with the same base name and extension, the function returns a handle to the module that was loaded first.

If no file name extension is specified in the lpFileName parameter, the default library extension .dll is appended. However, the file name string can include a trailing point character (.) to indicate that the module name has no extension. When no path is specified, the function searches for loaded modules whose base name matches the base name of the module to be loaded. If the name matches, the load succeeds. Otherwise, the function searches for the file.

The first directory searched is the directory containing the image file used to create the calling process (for more information, see the CreateProcess function). Doing this allows private dynamic-link library (DLL) files associated with a process to be found without adding the process's installed directory to the PATH environment variable. If a relative path is specified, the entire relative path is appended to every token in the DLL search path list. To load a module from a relative path without searching any other path, use GetFullPathName to get a nonrelative path and call LoadLibrary with the nonrelative path. For more information on the DLL search order, see Dynamic-Link Library Search Order.

The search path can be altered using the SetDllDirectory function. This solution is recommended instead of using SetCurrentDirectory or hard-coding the full path to the DLL.

If a path is specified and there is a redirection file for the application, the function searches for the module in the application's directory. If the module exists in the application's directory, LoadLibrary ignores the specified path and loads the module from the application's directory. If the module does not exist in the application's directory, LoadLibrary loads the module from the specified directory. For more information, see Dynamic Link Library Redirection.

If you call LoadLibrary with the name of an assembly without a path specification and the assembly is listed in the system compatible manifest, the call is automatically redirected to the side-by-side assembly.

The system maintains a per-process reference count on all loaded modules. Calling LoadLibrary increments the reference count. Calling the FreeLibrary or FreeLibraryAndExitThread function decrements the reference count. The system unloads a module when its reference count reaches zero or when the process terminates (regardless of the reference count).

##### Kỹ thuật Hooking API Window

1. Giới thiệu

Kỹ thuật này áp dụng cho Window NT 4.0 trở lên.

Trước hết chúng ta phải biết về process trên window, asembler, cấu trúc PE file và các API liên quan

Khi dùng thuật ngữ API, nghĩa là thay đổi toàn bộ API. Do đó, khi gọi tới các API bị hooked, code của chúng ta chạy ngay lập tức thay vì API hệ thống.

1. Mục đích

Mục đích là thay thế code trong hàm với code của chúng ta tại thời điểm trước runtime.

Giải pháp là can thiệp ở tầng process của ứng dụng chạy bởi người dùng với mục đích thay đổi hành vi ứng dụng.

Ví dụ kinh điển nhất là crack, bao gồm cả việc cho phép chạy không cần CD\_ROM trong game, bằng cách thay đổi nội dung phương thức CD request.

Tuy nhiên, chúng ta không thể thực hiện nếu không biết process mục tiêu. Do đó phải dùng kỹ thuật hooking trong runtime. Ví dụ các trình rootkit hay virus dùng kỹ thuật này chống lại ứng dụng anti-virus

1. Hook trước runtime

Kỹ thuật này sẽ can thiệp vào module thực thi (ext và dll), có 3 cách làm:

* Các đầu tiên là tìm entry point của hàm và viết lại code: khuyết điểm là số lượng hàm ít nhưng chúng ta có thể nạp vài module DLL, vậy là đủ.

Các hàm trong kernel32.dll có thể dùng trong nhiều trường hợp vì mỗi process trong Windows đều có bản sao module này.

Thuận lợi là chúng ta có thể dùng con trỏ trực tiếp, vì địa chỉ của module kernel trong vùng nhớ là tĩnh sau khi moduel được nạp. Phương pháp này ảnh hưởng bởi module.

* Cách thứ hai là thay thế hàm trong module bởi phiên bản mở rộng: chúng ta phải chọn giữa việc thay thế 5-bytes đầu tiên bằng cách nhảy theo địa chỉ tương đối hay viết lại IAT.

Trong trường hợp nhảy địa chỉ, nó sẽ chuyển hướng code thực thi tới code chúng ta.  
Điều khó khăn của phương pháp này là chúng ta phải xử lí được DLL header.

* Cách thứ 3 là thay thế toàn bộ module: tức tạo lại phiên bản mới module với hàm thay thế mới. Tuy nhiên không thể áp dụng với module lớn được (chứa hàng trăm exports trở lên).

1. Hook trong runtime

Hooking trước runtime rất hữu dụng cho các ứng dụng hay module liên kết. Nếu thay thế hàm trong kernel32.dll hay ntdll.dll (chỉ trên hệ Window NT ), chúng ta có thể thành công cho mọi process chạy sau đó, nhưng nó rất khó thực hiện vì cần tính tỉ mỉ.

Nhưng nếu chúng ta chỉ quan tâm tới một process nhất định cần hook

Vấn đề tiếp theo là truy xuất các file này vì Window NT sẽ cố bảo vệ chúng.

Hooking before running is mostly very special and intimately oriented for concrete application (or module). If we replace function in kernel32.dll or in ntdll.dll (only on NT OS) we will get perfect replace of this function in all processes which will be run later, but it is so difficult to make it because we have to take care about accuracy and code prefection of new functions or whole new modules, but the main problem is that only process which will be run later will be hooked (so for all process we have to reboot system). Next problem could be access to these files because NT OS tries to protect them. Much more pretty solution is to hook process during running. This method require more knowledge but the result is perfect. Hooking during running can be done only on process for which we have writing access to their memory. For the writing in itself we will use API function WriteProcessMemory. We will start from hooking our own process during running.

1. Own process hooking dùng IAT

There are many possibilities here. At first I will show you how to hook function by rewriting IAT. Following picture shows structure of PE file:

+-------------------------------+ - offset 0

| MS DOS Header ("MZ") and stub |

+-------------------------------+

| PE signature ("PE") |

+-------------------------------+

| .text | - module code

| Program Code |

| |

+-------------------------------+

| .data | - initialized (global static) data

| Initialized Data |

| |

+-------------------------------+

| .idata | - information for imported functions

| Import Table | and data

| |

+-------------------------------+

| .edata | - information for exported functions

| Export Table | and data

| |

+-------------------------------+

| Debug symbols |

+-------------------------------+

Important part for us here is Import Address Table (IAT) in the .idata part. This part contains description of imports and mainly imported functions addresses. Now it is important to know how are PE files created. When calling arbitrary API indirectly in programming language (that means we call it using its name, no using its OS specific address) the compiler does not link direct calls to the module but it links call to IAT on jmp instruction which will be filled by process loader while OS is loading process to the memory. This is why we can use the same binary on two different version os Windows where modules can be loaded to another addresses. Process loader will fill out direct jmp instructions in IAT which is used by our calls from the program code. So, if we are able to find out specific function in IAT which we want to hook, we can easily change jmp instruction there and redirect code to our address. Every call after doing this will execute our code. Advantage of this method is its perfection. Disadvantage is often amount of functions which should be hooked (e.g. if we want to change program behaviour in the file searching APIs we will have to change functions FindFirstFile and FindNextFile, but we have to know that these functions have its ANSI and WIDE version, so we have to change IAT address for FindFirstFileA, FindFirstFileW, FindNextFileA and also FileNextFileW. But there still some others like FindFirstFileExA and its WIDE version FindFirstFileExW which are called by previous mentioned functions. We know that FindFirstFileW calls FindFirstFileExW but this is done directly - not usinig IAT. And still some others to go. There are e.g. ShellAPI functions like SHGetDesktopFolder which also directly calls FindFirstFileW or FindFirstFileExW). But if we will get all of them, the result will be perfect.

We can use ImageDirectoryEntryToData from imagehlp.dll to find out IAT easily.