**ĐẠI HỌC BÁCH KHOA HÀ NỘI**

**TRƯỜNG CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG**

**A red and white logo

AI-generated content may be incorrect.**

**BÁO CÁO TUẦN 4**

**Môn học: Project II (IT3931)**

**Đề tài: Tìm hiểu windows**

|  |  |
| --- | --- |
| **Sinh viên thực hiện:** | **Ngô Trung Hiếu - 20225316**  **Kỹ thuật máy tính 04 – K67** |
| **Giáo viên hướng dẫn:** | **Nguyễn Đức Toàn** |

**Hà Nội – 2025**

Contents

[I. WINDOWS 4](#_Toc194409880)

[1. Mục tiêu 4](#_Toc194409881)

[2. Lịch sử 4](#_Toc194409882)

[II. QUẢN LÝ BỘ NHỚ 5](#_Toc194409883)

[1. Bộ nhớ ảo và phân trang 5](#_Toc194409884)

[2. Paging (phân trang) 7](#_Toc194409885)

[3. Lỗi trang (Page Faults) 9](#_Toc194409886)

[4. Tập làm việc (Working Sets) 9](#_Toc194409887)

[III. BỘ NHỚ KERNEL & USER 10](#_Toc194409888)

[1. Bộ nhớ Kernel 12](#_Toc194409889)

[2. Bộ nhớ User 15](#_Toc194409890)

[IV. TIẾN TRÌNH VÀ LUỒNG 16](#_Toc194409891)

[V. APPLICATION PROGRAMMING INTERFACE (API) 17](#_Toc194409892)

[1) Dynamic Link Library (DLL) 18](#_Toc194409893)

[2) Phân loại Win32 API 18](#_Toc194409894)

[3) The Native API 20](#_Toc194409895)

[4) System Call 23](#_Toc194409896)

[VI. EXECUTABLE FORMAT 26](#_Toc194409897)

[1) PE File 26](#_Toc194409898)

[2) Cấu trúc cơ bản của một PE File 26](#_Toc194409899)

[a) DOS MZ Header 27](#_Toc194409900)

[b) DOS STUB 28](#_Toc194409901)

[c) PE Header 29](#_Toc194409902)

[d) Section Table 32](#_Toc194409903)

[3) Relocation 34](#_Toc194409904)

[4) Vấn đề của relocation 35](#_Toc194409905)

[5) Image Sections 36](#_Toc194409906)

[5) Section Alignment 38](#_Toc194409907)

[6) Dynamically Linked Libraries (DLLs) 39](#_Toc194409908)

[VII. THE I/O SYSTEM 42](#_Toc194409909)

[1) Hệ thống I/O (I/O System) trong Windows 42](#_Toc194409910)

[2) Chức năng chính của hệ thống I/O 42](#_Toc194409911)

[3) Mô hình phân tầng (layered architecture) 43](#_Toc194409912)

[4) Lợi ích của kiến trúc phân tầng trong hệ thống I/O 43](#_Toc194409913)

[5) Filter Driver 44](#_Toc194409914)

[a) Định nghĩa 44](#_Toc194409915)

[b) Chức năng của Filter Drivers 44](#_Toc194409916)

[c) Ứng dụng của Filter Drivers trong Reverse Engineering 44](#_Toc194409917)

[d) Tầm quan trọng của vị trí Filter Driver trong hệ thống I/O 45](#_Toc194409918)

[6) Win32 Subsystem 46](#_Toc194409919)

# WINDOWS

## Mục tiêu

Hệ điều hành đóng vai trò quan trọng trong phân tích mã nguồn ngược (reverse engineering). Điều này là do các chương trình được tích hợp chặt chẽ với hệ điều hành, và có thể thu thập được rất nhiều thông tin bằng cách kiểm tra giao diện này. Hơn nữa, mục tiêu cuối cùng của mọi chương trình là giao tiếp với thế giới bên ngoài (chẳng hạn như nhận dữ liệu đầu vào từ người dùng, hiển thị dữ liệu lên màn hình, ghi dữ liệu vào tệp, v.v.), điều này có nghĩa là việc xác định và hiểu rõ các điểm kết nối giữa chương trình ứng dụng và hệ điều hành là rất quan trọng.

## Lịch sử

Trước đây từng tồn tại hai hệ điều hành khác nhau có tên là Windows: Windows và Windows NT.

* Windows bao gồm các phiên bản Windows 95, Windows 98 và Windows Me, là hậu duệ của các phiên bản Windows 16-bit cũ.
* Windows NT bao gồm Windows 2000 và sau này là Windows XP, Windows Server 2003.

Windows NT là một thiết kế mới hơn mà Microsoft bắt đầu phát triển từ đầu những năm 1990. Nó được xây dựng từ đầu để trở thành một hệ điều hành 32-bit, có khả năng quản lý bộ nhớ ảo, đa luồng (multithreading) và hỗ trợ nhiều bộ xử lý (multiprocessor-capable), giúp nó phù hợp hơn với phần cứng và phần mềm hiện đại. **Hệ điều hành được mô tả trong chương này chủ yếu là Windows XP (32bit).**

# QUẢN LÝ BỘ NHỚ

Bài báo cáo này chỉ tập trung vào hệ điều hành Windows 32-bit.

## Bộ nhớ ảo và phân trang

A diagram of a virtual memory

AI-generated content may be incorrect.

Hình 1: Bộ nhớ ảo, bảng trang và bộ nhớ vật lý

1. Bộ nhớ ảo (Virtual Memory)

Bộ nhớ ảo là một khái niệm cơ bản trong các hệ điều hành hiện đại. Ý tưởng chính là thay vì cho phép phần mềm truy cập trực tiếp vào bộ nhớ vật lý, **bộ xử lý (processor)** kết hợp với **hệ điều hành (Operating System – OS)** tạo ra một lớp vô hình giữa phần mềm và bộ nhớ vật lý.

Mỗi khi có một thao tác truy cập bộ nhớ, bộ xử lý sẽ tham khảo một bảng đặc biệt gọi là **bảng trang (page table)** để xác định địa chỉ bộ nhớ vật lý thực sự được sử dụng. Tất nhiên, việc tạo một mục trong bảng cho từng byte của bộ nhớ là không khả thi (vì bảng sẽ lớn hơn tổng dung lượng bộ nhớ vật lý). Thay vào đó, bộ xử lý chia bộ nhớ thành **các trang (pages)**.

1. Trang bộ nhớ (Memory Pages)

Trang là các khối bộ nhớ có kích thước cố định, và mỗi mục trong bảng trang sẽ quản lý một trang bộ nhớ. Kích thước thực tế của một trang bộ nhớ khác nhau giữa các kiến trúc bộ xử lý, và một số kiến trúc hỗ trợ nhiều kích thước trang khác nhau. Ví dụ:

* Các bộ xử lý IA-32 (x86) thường sử dụng các trang có kích thước 4 KB, nhưng cũng hỗ trợ các trang lớn hơn 2 MB và 4 MB.
* Windows chủ yếu sử dụng các trang 4 KB, vì vậy kích thước này thường được coi là mặc định.

1. Lợi ích của bảng trang

Một trong những lợi ích quan trọng nhất là nó cho phép tạo ra nhiều không gian địa chỉ **(address space).**

**Không gian địa chỉ là một bảng trang riêng biệt, chỉ cho phép truy cập vào bộ nhớ liên quan đến chương trình hoặc tiến trình hiện tại.** Vì mỗi tiến trình không thể truy cập bảng trang của tiến trình khác, nó không thể vượt qua ranh giới này. Khả năng tạo nhiều không gian địa chỉ là một đặc điểm quan trọng của các hệ điều hành hiện đại, vì nó đảm bảo rằng các chương trình hoàn toàn bị cách ly với nhau và mỗi tiến trình có môi trường riêng để chạy.

1. Kiểm soát truy cập bộ nhớ

A screenshot of a computer

AI-generated content may be incorrect.

Hình 2: Bảng trang ngoài bao gồm trường vị trí - xác định vị trí của trang,

còn có trường flags - giới hạn quyền truy cập.

Ngoài việc quản lý không gian địa chỉ, bảng trang cũng giúp bộ xử lý dễ dàng áp đặt các quy tắc về cách bộ nhớ được truy cập. Ví dụ:

* Mỗi mục trong bảng trang có một tập hợp cờ (flags) để xác định các thuộc tính nhất định, chẳng hạn như trang có thể được truy cập từ chế độ không đặc quyền (nonprivileged mode) hay không.
* Điều này có nghĩa là mã của hệ điều hành có thể nằm trong không gian địa chỉ của tiến trình, nhưng chỉ cần đặt một cờ hạn chế trong bảng trang, hệ điều hành có thể ngăn ứng dụng truy cập dữ liệu quan trọng của nó.

Điều này dẫn chúng ta đến hai khái niệm cơ bản: chế độ nhân (kernel mode) và chế độ người dùng (user mode).

* Chế độ nhân (Kernel mode) về cơ bản là thuật ngữ của Windows để chỉ chế độ đặc quyền của bộ xử lý (privileged processor mode). Nó thường được sử dụng để mô tả mã chạy trong chế độ đặc quyền hoặc vùng nhớ chỉ có thể truy cập khi bộ xử lý đang ở chế độ đặc quyền.
* Chế độ người dùng (User mode) là chế độ không đặc quyền (nonprivileged mode). Khi hệ thống hoạt động trong chế độ người dùng, nó chỉ có thể chạy mã trong chế độ người dùng và chỉ có thể truy cập vùng nhớ dành cho chế độ người dùng.

## Paging (phân trang)

1. Định nghĩa

Phân trang là một quá trình trong đó các vùng nhớ tạm thời được chuyển vào ổ cứng khi chúng không được sử dụng.

Ý tưởng rất đơn giản:

* Bộ nhớ vật lý (RAM) nhanh hơn nhưng cũng đắt hơn so với ổ cứng.
* Vì vậy, hệ thống sử dụng một tệp sao lưu (paging file) để lưu trữ tạm thời những vùng nhớ không sử dụng ngay lập tức.

Hãy tưởng tượng một hệ thống đang chạy nhiều ứng dụng: Khi một số ứng dụng không được sử dụng, thay vì giữ toàn bộ ứng dụng trong bộ nhớ vật lý, cấu trúc bộ nhớ ảo (virtual memory architecture) cho phép hệ thống chuyển toàn bộ vùng nhớ đó vào một tệp trên ổ cứng. Khi cần thiết, hệ thống sẽ nạp lại dữ liệu này vào bộ nhớ RAM. Quá trình này hoàn toàn vô hình đối với ứng dụng (tức là ứng dụng không nhận ra sự thay đổi này).

1. Cách hoạt động

Hệ thống phải duy trì một cơ chế theo dõi thời điểm một trang nhớ được truy cập lần cuối (bộ xử lý hỗ trợ quá trình này). Dựa trên thông tin đó, hệ thống có thể:

* Xác định các trang nhớ ít được sử dụng.
* Chuyển nội dung của chúng vào một tệp trên ổ cứng.
* Vô hiệu hóa các mục tương ứng trong bảng trang, nói cách khác đặt 1 trường thông tin của mục đó về *false* để xác định trang này chưa được nạp vào bộ nhớ chính.
* Giải phóng bộ nhớ vật lý để dùng cho các tiến trình khác.

Khi một chương trình cố gắng truy cập lại một trang đã bị chuyển sang ổ cứng, bộ xử lý sẽ phát sinh một lỗi trang *(page fault)* vì mục trong bảng trang đã bị vô hiệu hóa. Khi đó, hệ điều hành sẽ nhận diện rằng trang này đã bị phân trang ra ngoài (paged out) và sẽ đọc lại dữ liệu từ tệp phân trang trên ổ cứng và nạp lại vào bộ nhớ vật lý.

1. Lợi ích và hạn chế của việc phân trang

**Lợi ích:**

* Cho phép sử dụng nhiều bộ nhớ hơn so với lượng bộ nhớ vật lý có sẵn.
* Cải thiện hiệu suất tổng thể khi có nhiều ứng dụng chạy đồng thời.

**Hạn chế:**

* Nếu một ứng dụng liên tục sử dụng nhiều bộ nhớ hơn lượng RAM thực tế, hệ thống sẽ liên tục di chuyển dữ liệu giữa RAM và ổ cứng.
* Vì ổ cứng chậm hơn bộ nhớ vật lý khoảng 1.000 lần, điều này có thể khiến hệ thống chạy cực kỳ chậm.

## Lỗi trang (Page Faults)

Từ góc nhìn của bộ xử lý, một lỗi trang (page fault) xảy ra bất cứ khi nào một địa chỉ bộ nhớ được truy cập mà không có một mục hợp lệ trong bảng trang (page table entry). Trên thực tế, hệ thống khỏe mạnh có thể phát sinh hàng nghìn lỗi trang mỗi giây. Trong hầu hết các trường hợp, hệ điều hành xử lý lỗi trang một cách bình thường mà người dùng không nhận ra. Bên cạnh đó, có những lỗi trang không thể xử lý và làm cho hệ thống gặp sự cố (crash).

Ví dụ về lỗi trang hợp lệ

Một ví dụ điển hình về lỗi trang hợp lệ là khi một trang bộ nhớ đã bị phân trang ra ngoài (paged out) vào tệp phân trang (paging file) và một chương trình cần truy cập lại nó:

* Vì mục trong bảng trang đã bị vô hiệu hóa (invalid), bộ xử lý sẽ phát sinh một lỗi trang.
* Hệ điều hành nhận diện lỗi này, nạp lại nội dung của trang từ tệp phân trang vào bộ nhớ vật lý.
* Sau đó, chương trình tiếp tục hoạt động bình thường, như thể không có gì xảy ra.

## Tập làm việc (Working Sets)

Tập làm việc là một cấu trúc dữ liệu dành riêng cho từng tiến trình, liệt kê các trang bộ nhớ vật lý hiện đang được sử dụng trong không gian địa chỉ của tiến trình đó.

Hệ thống sử dụng tập làm việc để:

* Xác định mức độ sử dụng bộ nhớ vật lý của mỗi tiến trình.
* Kiểm tra các trang bộ nhớ không được truy cập trong một khoảng thời gian.
* Những trang này có thể được phân trang ra ổ đĩa (paged out) và bị loại khỏi tập làm việc của tiến trình.

**Đo lường mức sử dụng bộ nhớ của tiến trình**

Người ta có thể nói rằng mức sử dụng bộ nhớ của một tiến trình tại bất kỳ thời điểm nào có thể được đo bằng tổng kích thước của tập làm việc của nó.  
Điều này phần lớn là đúng, nhưng cũng là một cách đơn giản hóa vấn đề vì:

* Một phần lớn không gian địa chỉ của tiến trình thường chứa bộ nhớ chia sẻ (shared memory).
* Bộ nhớ chia sẻ này cũng được tính vào tổng kích thước tập làm việc.

# BỘ NHỚ KERNEL & USER

A diagram of a computer program

AI-generated content may be incorrect.

Hình 3: Bộ nhớ ảo của 1 process được chia ra làm 2 phần: Kernel và User

Có lẽ khái niệm quan trọng nhất trong quản lý bộ nhớ là sự phân biệt giữa bộ nhớ nhân (kernel memory) và bộ nhớ người dùng (user memory).

Trong một hệ điều hành ổn định và bảo mật, các ứng dụng không được phép truy cập vào các cấu trúc dữ liệu nội bộ của hệ điều hành. Lý do là:

* Tránh lỗi lập trình: Nếu một lập trình viên mắc lỗi làm ghi đè lên dữ liệu quan trọng của hệ điều hành, toàn bộ hệ thống có thể bị mất ổn định.
* Bảo vệ hệ thống khỏi mã độc: Ngăn chặn phần mềm độc hại chiếm quyền kiểm soát hoặc làm hỏng hệ thống bằng cách truy cập vào dữ liệu quan trọng của hệ điều hành.

**Cách Windows phân bổ bộ nhớ**

Windows sử dụng một địa chỉ bộ nhớ 32-bit (tổng cộng 4 GB bộ nhớ ảo), được chia thành hai phần bằng nhau:

* 2 GB dành cho ứng dụng người dùng (user memory).
* 2 GB dành cho bộ nhớ nhân (kernel memory), được chia sẻ giữa tất cả các tiến trình.

Mặc dù có một số trường hợp hệ thống 32-bit sử dụng cách phân bổ bộ nhớ khác, nhưng chúng không phổ biến.

**Lợi ích của thiết kế này:**

* Bộ nhớ nhân luôn sẵn sàng, bất kể tiến trình nào đang chạy.
* Bộ nhớ nhân được bảo vệ, các tiến trình user-mode không thể truy cập.

**Tác dụng phụ của thiết kế này**

Một tác dụng phụ của thiết kế này là:

* Các ứng dụng thực tế chỉ có không gian địa chỉ 31-bit.
* Bit quan trọng nhất (Most Significant Bit – MSB) luôn bằng 0 trong mọi địa chỉ hợp lệ của chế độ người dùng.

Gợi ý hữu ích khi reverse engineering (phân tích ngược):

* Một số nguyên 32-bit mà chữ số thập lục phân đầu tiên là 8 hoặc lớn hơn (0x80000000 trở lên) không thể là con trỏ hợp lệ trong chế độ người dùng.

## Bộ nhớ Kernel

Vậy, **2GB bộ nhớ được dành riêng cho kernel** dùng để làm gì? Không gian này được chia cho nhiều thành phần khác nhau của hệ điều hành. Chủ yếu, nó chứa toàn bộ **mã của kernel**, bao gồm chính kernel và các thành phần kernel khác như driver thiết bị.

Hầu hết **2GB này được chia cho một số thành phần quan trọng của hệ thống.** Việc phân chia này thường là **cố định**, nhưng một số **khóa registry** có thể ảnh hưởng đến kích thước của một số vùng này. Hình dưới đây minh họa cách bố trí bộ nhớ của Windows trong không gian địa chỉ của kernel.

Lưu ý rằng hầu hết các thành phần này có **kích thước động**, có thể thay đổi **tùy theo dung lượng bộ nhớ vật lý** và **các thiết lập của người dùng.**

A diagram of a system

AI-generated content may be incorrect.

Hình 4: Cấu trúc của không gian nhớ kernel

**Bộ nhớ Paged Pool và Nonpaged Pool**

Paged Pool và Nonpaged Pool là bộ nhớ heap của kernel-mode, được toàn bộ các thành phần kernel sử dụng. Vì chúng được lưu trong bộ nhớ kernel, chúng có sẵn trong tất cả các không gian địa chỉ, nhưng chỉ có mã chạy ở chế độ kernel mới có thể truy cập.

* Paged Pool: Đây là một heap lớn, bao gồm bộ nhớ có thể phân trang (paged memory). Đây là bộ nhớ mặc định cho hầu hết các thành phần kernel.
* Nonpaged Pool: Đây là một heap không thể phân trang (nonpaged memory). Bộ nhớ này không thể bị đẩy ra ổ cứng và luôn nằm trong bộ nhớ vật lý. Điều này rất hữu ích vì nhiều vùng quan trọng của hệ thống không được phép sử dụng bộ nhớ có thể phân trang.

**Bộ nhớ Cache của Hệ thống (System Cache)**

Đây là nơi Windows lưu trữ các tệp tin đang được cache.

Cơ chế caching trong Windows:

* Khi một chương trình mở một tệp tin, một đối tượng vùng nhớ (section object) sẽ được tạo ra cho tệp đó và được ánh xạ vào vùng cache của hệ thống.
* Khi chương trình đọc hoặc ghi dữ liệu, Windows sử dụng bộ nhớ cache thay vì truy cập ổ cứng ngay lập tức, giúp tăng tốc độ.
* Các API như CcCopyRead và CcCopyWrite được dùng để quản lý việc truy cập dữ liệu trong cache

**Bộ Nhớ Phiên của Terminal Services (Session Space)**

Đây là bộ nhớ dành riêng cho kernel-mode của Win32 Subsystem (*WIN32K.SYS*).

Windows hỗ trợ nhiều phiên làm việc đồng thời (*Terminal Services*), nghĩa là nhiều người dùng có thể truy cập cùng một máy từ xa. Để làm được điều đó, Windows tạo nhiều không gian phiên (session space), mỗi phiên có không gian bộ nhớ riêng biệt cho giao diện đồ họa.

* Trong kernel, mỗi phiên được tải vào cùng một địa chỉ ảo, nhưng được tách biệt bằng session space.
* Session Space chứa WIN32K.SYS và các cấu trúc dữ liệu cần thiết cho hệ thống Win32.
* Session Pool là một Paged Pool riêng biệt dành cho từng phiên.

**Page Tables & Hyper Space**

Đây là vùng chứa dữ liệu riêng của từng tiến trình, giúp xác định không gian địa chỉ hiện tại.

* Bảng trang (Page Tables): Đây là vùng ánh xạ các bảng trang hiện tại, giúp CPU biết địa chỉ bộ nhớ nào đang được sử dụng.
* Hyper Space: Được sử dụng để quản lý bộ nhớ đang hoạt động của tiến trình hiện tại. Chủ yếu, nó quản lý Working Set của tiến trình.

**System Working Set**

Là một cấu trúc dữ liệu toàn hệ thống (system-global), giúp quản lý bộ nhớ vật lý đang được sử dụng bởi bộ nhớ có thể phân trang (paged memory).

* Vùng bộ nhớ paged pool và system cache là hai phần lớn nhất được quản lý bởi cấu trúc này.
* Nó giúp Windows theo dõi các trang bộ nhớ nào đang được sử dụng và các trang nào có thể được phân trang ra đĩa.

**System Page-Table Entries (System PTE)**

Là vùng bộ nhớ được kernel sử dụng để cấp phát bộ nhớ lớn cho các driver và các thành phần kernel khác.

* Không phải là heap, mà chỉ là không gian bộ nhớ ảo, có thể được kernel hoặc driver sử dụng khi cần.
* Windows sử dụng System PTE để:
  + Nạp driver thiết bị vào bộ nhớ.
  + Lưu trữ kernel stacks (mỗi luồng trong hệ thống có một kernel stack).
* Driver có thể cấp phát bộ nhớ từ vùng này bằng cách gọi API MmAllocateMappingAddress.

## Bộ nhớ User

# TIẾN TRÌNH VÀ LUỒNG

# APPLICATION PROGRAMMING INTERFACE (API)

**Win32 API** (Windows 32-bit Application Programming Interface) là tập hợp các hàm lõi cho phép các chương trình tương tác trực tiếp với hệ điều hành Windows. Nó cung cấp quyền truy cập cấp thấp vào các tài nguyên hệ thống như quản lý bộ nhớ, xử lý tệp và giao diện đồ họa.

**Sự phát triển của lập trình Windows:**

1. **Thời kỳ đầu – Win32 API**
   * Ban đầu, các ứng dụng Windows được xây dựng bằng **Win32 API**.
   * Điều này yêu cầu lập trình viên phải viết nhiều mã phức tạp để quản lý cửa sổ, sự kiện và tương tác với hệ thống.
2. **Giới thiệu MFC (Microsoft Foundation Classes)**
   * Để đơn giản hóa việc phát triển, Microsoft đã giới thiệu **MFC**, một tập hợp các **lớp C++** giúp bọc quanh Win32 API.
   * MFC giúp việc tạo ứng dụng Windows trở nên dễ dàng hơn bằng cách cung cấp các đối tượng và cơ chế xử lý sự kiện có sẵn.
   * Tuy nhiên, bên trong, MFC vẫn gọi đến **Win32 API** để tương tác với hệ điều hành.
3. **Phát triển hiện đại – .NET Framework**
   * Sau đó, Microsoft ra mắt **.NET Framework**, cung cấp một mô hình lập trình cấp cao hơn.
   * Thay vì sử dụng MFC hoặc Win32 trực tiếp, lập trình viên sử dụng **lớp System** trong .NET để truy cập các tính năng của Windows.
   * **.NET Framework** vẫn sử dụng **Win32 API** ở cấp hệ thống.

**Tóm tắt:**

* **Win32 API** → API lõi cấp thấp của Windows.
* **MFC** → Một thư viện C++ giúp đơn giản hóa việc sử dụng Win32.
* **.NET Framework** → Một nền tảng lập trình hiện đại giúp lập trình Windows dễ dàng hơn nhưng vẫn dựa vào Win32 ở bên dưới.

Theo thời gian, Microsoft đã cung cấp các công cụ cấp cao hơn, dễ sử dụng hơn cho lập trình viên, nhưng **Win32 API vẫn là giao diện nền tảng cốt lõi của Windows**.

## Dynamic Link Library (DLL)

**Thư viện liên kết động** (tiếng anh: Dynamic Link Library - viết tắt: **DLL**) là một thành phần của các phần mềm. Đúng như tên gọi của nó, là một thư viện cho phép các ứng dụng có thể liên kết đến và sử dụng nó. Nó được xem là một tổ hợp các hàm và dữ liệu mà có thể được sử dụng bởi nhiều ứng dụng khác nhau ở cùng một thời điểm. *Ví dụ*: thư viện user32.dll hoặc kernel32.dll là các thư viện liên kết động mà mỗi ứng dụng Windows đều phải dùng đến.

Khi đã được viết và đóng gói trong một DLL, một hàm có thể được sử dụng ở bất cứ ứng dụng nào dưới dạng mã máy của nó mà không phải quan tâm đến mã nguồn nó được viết chi tiết ra sao. Với các ứng dụng trước kia, ta có thể đính kèm file mã nguồn và sử dụng các hàm trong chương trình; tuy nhiên, giả sử như để có thể thao tác với máy in, in một nội dung ta cần gọi hàm print() và ta đã có mã nguồn của hàm print này. Giả sử như không có DLL, thì bất cứ ứng dụng nào muốn sử dụng hàm print() sẽ bao gồm cả hàm này vào trong mã nguồn: từ chương trình Word, Excel, Internet Explorer,... như vậy trong hệ thống sẽ có rất nhiều bản sao của hàm print() này, làm cho dung lượng để chứa các chương trình tăng lên.

Ngoài ra, khi phần cứng thay đổi, ta lại phải thay đổi mã nguồn của hàm print() để nó hoạt động đúng (nếu cần), việc này đẫn tới việc thay đổi theo dây chuyền đến các ứng dụng, gây ra rất nhiều khó khăn.

## Phân loại Win32 API

Win32 API lõi bao gồm khoảng 2000 hàm, chia thành ba nhóm chính: Kernel, USER và GDI. Figure dưới đây sẽ mô tả mối tương tác, quan hệ giữa Win32 API và các thành phần kernel.

A diagram of a computer application process

AI-generated content may be incorrect.

Figure 1 : Mối quan hệ giữa Win32 interface DLLs và các thành phần của kernel

Đây là các thư viện DLL (Dynamic Link Libraries) mà ứng dụng tương tác trực tiếp:

**Thành Phần Chạy Ở Chế Độ Người Dùng (User-Mode - Phần Trên)**

1. KERNEL32.DLL (Thành phần API cơ bản)
   * Cung cấp các chức năng hệ thống như quản lý tiến trình, bộ nhớ và thao tác tệp.
   * Gọi các hàm trong NTDLL.DLL, đóng vai trò là giao diện gốc để truy cập kernel.
2. NTDLL.DLL (Giao diện API gốc - Native API Interface)
   * API cấp thấp nhất ở chế độ người dùng, cung cấp truy cập trực tiếp vào nhân Windows (NTOSKRNL.EXE).
   * Hầu hết các lệnh hệ thống đều đi qua NTDLL.DLL trước khi đến kernel.
3. USER32.DLL (Thành phần API người dùng - User API Client Component)
   * Cung cấp các hàm để quản lý cửa sổ, xử lý sự kiện, menu và nhập liệu (bàn phím/chuột).
   * Gọi đến WIN32K.SYS để thực hiện các thao tác UI cấp kernel.
4. GDI32.DLL (Thành phần API đồ họa - GDI API Client Component)
   * Xử lý các thao tác đồ họa và vẽ, chẳng hạn như hiển thị văn bản, hình ảnh, và giao diện người dùng.
   * Gọi đến WIN32K.SYS để thực hiện các tác vụ đồ họa cấp kernel.

**Thành Phần Chạy Ở Chế Độ Kernel (Kernel-Mode - Phần Dưới)**

Đây là các thành phần lõi của hệ thống, thực hiện các thao tác có quyền cao:

1. NTOSKRNL.EXE (The Windows Kernel)
   * Quản lý các chức năng lõi của hệ điều hành, bao gồm quản lý bộ nhớ, lập lịch tiến trình/luồng, bảo mật, và giao tiếp phần cứng.
   * Xử lý các lời gọi hệ thống từ NTDLL.DLL.
2. WIN32K.SYS (The Win32 Kernel Implementation)
   * Xử lý các tác vụ giao diện người dùng và đồ họa ở cấp kernel, chẳng hạn như vẽ cửa sổ, xử lý đầu vào, và hiển thị đồ họa hiệu quả.
   * Được gọi bởi USER32.DLL và GDI32.DLL để thực hiện các thao tác UI cần truy cập phần cứng trực tiếp.

## The Native API

Native API là giao diện lập trình cấp thấp nhất trong hệ điều hành Windows NT, cho phép các thành phần hệ thống và trình điều khiển giao tiếp trực tiếp với nhân hệ điều hành (Windows Kernel).

Win32 API chỉ là một lớp trung gian nằm trên Native API để giúp các lập trình viên làm việc với Windows dễ dàng hơn.

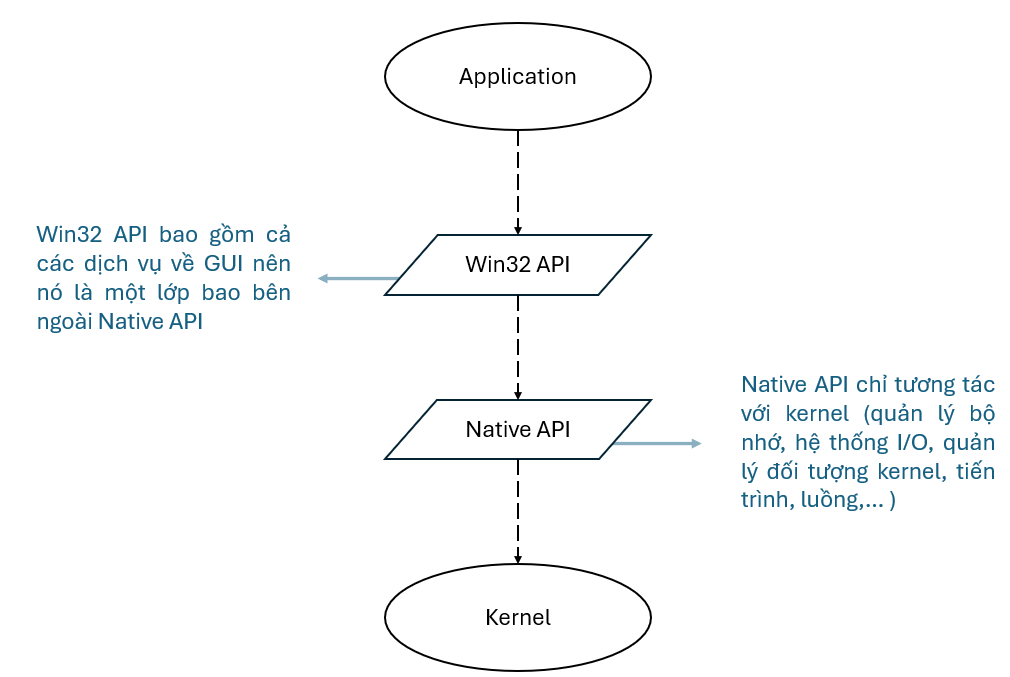


Figure 2 : Minh họa giữa Win32 API và Native API

1. Native API được triển khai ở đâu?

* Native API gồm các hàm được xuất từ:
  + NTDLL.DLL: Dành cho các chương trình chạy ở chế độ User-Mode.
  + NTOSKRNL.EXE: Dành cho các thành phần chạy ở chế độ Kernel-Mode (như trình điều khiển).

2. Nt và Zw có nghĩa là gì?

* Tất cả các hàm trong Native API đều bắt đầu bằng tiền tố Nt hoặc Zw.
  + Ví dụ: NtCreateFile, ZwCreateFile
* Sự khác biệt giữa hai phiên bản này:
  + Trong chế độ User-Mode (NTDLL.DLL):
    - Cả Nt\* và Zw\* trỏ đến cùng một đoạn mã.
  + Trong chế độ Kernel-Mode (NTOSKRNL.EXE):
    - Nt\* là phiên bản thực sự của API.
    - Zw\* là một stub (hàm chuyển tiếp), gọi vào hệ thống thông qua System Call Mechanism (Cơ chế gọi hệ thống).

3. Tại sao lại có sự phân biệt này?

* Lý do sử dụng System Call Mechanism trong Kernel-Mode:
  + Khi gọi một API từ chế độ người dùng (User-Mode), hệ điều hành kiểm tra xem các tham số có hợp lệ không (ví dụ: không được truyền con trỏ trỏ đến vùng nhớ của kernel).
  + Khi gọi API từ chế độ Kernel-Mode, ta cần chứng minh rằng cuộc gọi thực sự đến từ Kernel-Mode, để hệ thống không áp dụng các kiểm tra bảo mật dành cho User-Mode.
  + *Các hàm Zw giúp Kernel-Mode bỏ qua các kiểm tra không cần thiết*\*, vì vậy các driver thường dùng chúng để tăng hiệu suất.

4. Kết luận

* *Native API có hai nhóm hàm Nt và Zw*\*\*, dùng trong các ngữ cảnh khác nhau.
* User-Mode (NTDLL.DLL): Hai nhóm này là giống nhau.
* Kernel-Mode (NTOSKRNL.EXE): Nt\* là bản gốc, còn Zw\* là phiên bản gọi qua System Call Mechanism để xác nhận cuộc gọi từ kernel.
* *Mục tiêu chính của Zw API trong Kernel-Mode*\* là tránh kiểm tra bảo mật không cần thiết, giúp các tiến trình nhân hệ điều hành chạy nhanh hơn.

A diagram of a software application

AI-generated content may be incorrect.

Figure 3: Minh họa các API của Native API

## System Call

1. Hệ thống gọi hệ thống là gì?

* System Call (Gọi hệ thống) là cơ chế để chương trình ở chế độ User-Mode gọi các hàm ở chế độ Kernel-Mode.
* Khi một ứng dụng gọi API của hệ điều hành (Win32 API), lệnh thực sự thường được chuyển xuống nhân hệ điều hành (Kernel) để thực hiện.

2. Tại sao không thể gọi trực tiếp từ User-Mode vào Kernel-Mode?

* Nếu ứng dụng có thể gọi trực tiếp vào Kernel, nó có thể:
  + Gọi vào một địa chỉ bộ nhớ không hợp lệ, gây sập hệ thống (crash).
  + Gọi vào một địa chỉ độc hại, giành quyền kiểm soát hệ thống (lỗ hổng bảo mật nghiêm trọng).
* Vì thế, Windows sử dụng một cơ chế đặc biệt để chuyển đổi từ User-Mode sang Kernel-Mode một cách an toàn.

3. Cách hệ thống gọi hàm từ User-Mode xuống Kernel-Mode

* Một lệnh đặc biệt của CPU được sử dụng để chuyển đổi sang Kernel-Mode.
* Khi gọi một API hệ thống, quá trình diễn ra như sau:
  1. Mã User-Mode gọi API (Ví dụ: ReadFile trong Win32 API).
  2. API kiểm tra tham số để đảm bảo không có lỗi.
  3. Gọi vào Native API (ZwReadFile trong NTDLL.DLL).
  4. Gọi vào System Call Mechanism để chuyển quyền kiểm soát xuống Kernel-Mode.
  5. Kernel thực hiện lệnh và trả kết quả về lại User-Mode.

4. Cách thực hiện System Call trong Windows 2000 trở về trước

* Trong Windows 2000 và các phiên bản trước, Kernel-Mode được gọi bằng ngắt (interrupt) 2E.
* Dưới đây là một đoạn mã Assembly minh họa cách Windows 2000 thực hiện ZwReadFile:

A black and white text

AI-generated content may be incorrect.

Giải thích:

* + mov eax, 0xa1: Đặt mã số hệ thống (System Call Number) vào thanh ghi EAX.
  + lea edx, [esp+0x4]: Đặt địa chỉ tham số vào EDX.
  + int 2e: Kích hoạt ngắt 2E, giúp CPU chuyển sang Kernel-Mode và gọi hàm trong Kernel.
  + ret 0x24: Trả kết quả về User-Mode.

5. Kết luận

* System Call Mechanism là cách để chương trình từ User-Mode gọi vào nhân hệ điều hành một cách an toàn.
* Windows 2000 và các phiên bản trước sử dụng ngắt 2E (interrupt 2E) để thực hiện gọi hệ thống.
* Từ Windows XP trở đi, Microsoft đã thay đổi cơ chế này (sử dụng lệnh sysenter trên CPU Intel hoặc syscall trên CPU AMD) để tăng hiệu suất.

Giải thích về cơ chế gọi hệ thống với ngắt 2E trong Windows 2000

1. Tóm tắt quy trình gọi hệ thống

* Khi một ứng dụng ở chế độ User-Mode muốn gọi một hàm trong Kernel-Mode, nó sử dụng một cơ chế đặc biệt gọi là System Call Mechanism.
* Trong Windows 2000 và các phiên bản trước, lệnh ngắt 2E (int 2e) được sử dụng để chuyển từ User-Mode sang Kernel-Mode.

2. Vai trò của các thanh ghi EAX và EDX

* EAX: Chứa mã số dịch vụ (Service Number) tương ứng với hàm hệ thống cần gọi.
* EDX: Chứa địa chỉ tham số đầu tiên được truyền vào hàm Kernel.

3. Cách CPU xử lý lệnh int 2e

1. Lệnh int 2e được thực thi, CPU sẽ:
   * Tra cứu Bảng Mô Tả Ngắt (Interrupt Descriptor Table - IDT).
   * IDT xác định hàm xử lý ngắt tương ứng cho số hiệu 2E.
2. IDT trỏ đến hàm xử lý hệ thống KiSystemService
   * IDT là một bảng do CPU quản lý, lưu trữ thông tin về các hàm xử lý khi có ngắt (interrupt) hoặc ngoại lệ (exception) xảy ra.
   * Mỗi mục trong IDT chứa con trỏ đến một hàm xử lý ngắt tương ứng.
   * Mục IDT cho ngắt 2E trỏ đến hàm KiSystemService trong NTOSKRNL.EXE.
3. KiSystemService xử lý yêu cầu và gọi hàm Kernel tương ứng
   * KiSystemService là bộ điều phối dịch vụ hệ thống (kernel service dispatcher).
   * Nó kiểm tra mã số dịch vụ (EAX) và con trỏ stack để đảm bảo yêu cầu hợp lệ.
   * Sau đó, nó tìm kiếm hàm Kernel tương ứng trong một bảng gọi là KiServiceTable.

4. KiServiceTable – Bảng ánh xạ các hàm hệ thống

* KiServiceTable chứa con trỏ đến tất cả các hàm dịch vụ Kernel mà Windows hỗ trợ.
* KiSystemService sử dụng giá trị trong EAX như một chỉ mục, tra cứu KiServiceTable để tìm đúng hàm và thực thi nó.
* Điều này giúp Kernel có thể gọi đúng hàm cần thiết một cách nhanh chóng và có tổ chức.

5. Kết luận

* Khi int 2e được gọi, CPU tìm trong IDT, xác định handler là KiSystemService.
* KiSystemService kiểm tra tham số, sau đó tra cứu KiServiceTable để gọi đúng hàm Kernel.
* Cơ chế này giúp User-Mode giao tiếp với Kernel-Mode một cách an toàn và có kiểm soát.

# EXECUTABLE FORMAT

Sự hiểu biết cơ bản về định dạng thực thi là rất quan trọng cho quá trình RE. Bởi lẽ, định dạng thực thi của mỗi chương trình thường đem lại những gợi ý đáng kể về kiến trúc của chương trình đó. Chính vì vậy, để hiểu được hệ thống, cần hiểu định dạng thực thì của hệ thống.

## PE File

PE File Format (Portable Executable File Format): là định dạng file riêng của Win32. Tất cả các file có thể thực thi được trên Win32 như: \*.EXE, \*.DLL (32 bit),\*.COM,\*.NET, \*.CPL,… đều là định dạng PE; ngoại trừ các tập tin VxDs và \*.DLL (16 bit). Ngay cả NT’s kernel mode driver cũng sử dụng định dạng tệp PE. Do đó nghiên cứu định dạng tập tin PE cung cấp cho bạn cái nhìn sâu sắc về cấu trúc của Windows.

## Cấu trúc cơ bản của một PE File

A diagram of a diagram

AI-generated content may be incorrect.

Figure 4 : Cấu trúc cơ bản của PE File

PE file được chia làm 2 phần Header và Section, trong đó Header dùng để lưu các giá trị định dang file và các offset của các section trong phần Section. Sau đâu chúng ta đi tìm hiểu về phần Header của file.

### DOS MZ Header

Tất cả các PE file đếu bắt đầu bằng một DOS MZ Header đơn giản. Nó chiếm 64 bytes đầu tiên. Vùng này được dùng  trong trường hợp chương trình chạy trên nền DOS, hệ điều hành DOS nhận biết đây là một file thực thi hợp lệ và sẽ thực thi nội dung trong phần DOS STUB.

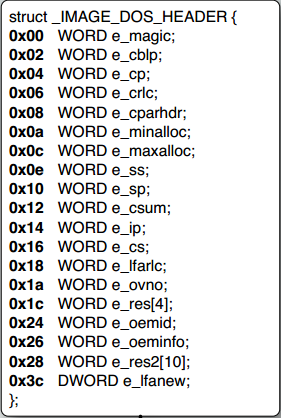


Figure 5: Cấu trúc của DOS MZ Header

Trong đó chúng ta cần quan tâm tới hai trường:

* **e\_magic**: Chữ ký của PE file, giá trị: 4Dh, 5Ah (Ký tự “MZ”, tên của người sáng lập MS-DOS: Mark Zbikowsky). Giá trị này đánh dấu một DOS Header hợp lệ và được phép thực thi tiếp.
* **e\_lfanew**: là một DWORD nằm ở cuối cùng của DOS Header, là trường chứa offset của PE Header so với vị trí đầu file, ta có thể nói nó là 1 con trỏ trỏ tới PE Header.

### DOS STUB

DOS Stub chỉ là một chương trình DOS EXE nhỏ hiển thị một thông báo lỗi, là phần để tương thích với Windows 16bit. Ví dụ như trong hình minh họa dưới đây, thông báo sẽ hiện ra như sau: “This is program cannot be run in DOS mode”

A blue and white rectangular grid with black numbers

AI-generated content may be incorrect.

Figure 6: Thông báo chương trình không thể chạy trên DOS mode

### PE Header

PE Header thực chất là cấu trúc **IMAGE\_NT\_HEADERS**bao gồm các thông tin cần thiết cho quá trình loader load file lên bộ nhớ. Cấu trúc này gồm 3 phần được định nghĩa trong windows.inc

A white background with black text

AI-generated content may be incorrect.

Figure 7: Cấu trúc IMAGE\_NT\_HEADERS

**Signature**: là 1 DWORD bắt đầu PE Header chứa chữ ký PE: 50h, 45h, 00h, 00

**FILE\_HEADER**: bao gồm 20 bytes tiếp theo của PE Header, phần này chứa thông tin về sơ đồ bố trí vật lý và các đặc tính của file.

A close-up of a list of symbols

AI-generated content may be incorrect.

Figure 8: Cấu trúc \_IMAGE\_FILE\_HEADER

Trong trường này chúng ta cần chú ý tới trường

* **Machine**: giá trị xác định PE File này được biên dịch cho dòng máy nào
* **NumberOfSections:**đây là trường chứa số section của file. Nếu muốn thêm/xoá section trong PE file, ta cần thay đổi tương ứng trường này.
* **Characteristics**: là bit cờ, xác định định dạng PE File.

**OPTIONAL\_HEADER**: bao gồm 224 bytes tiếp theo sau FILE\_HEADER. Cấu trúc này được định nghĩa trong windows.inc, đây là phần chứa thông tin về sơ đồ logic trong PE file. Dưới đây là danh sách các trường trong cấu trúc này, đồng thời sẽ đưa ra một số chỉ dẫn về thông tin của một số trường cần quan tâm khi muốn chỉnh sửa file.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 9: Cấu trúc \_IMAGE\_OPTIONAL\_HEADER

Trong đó, các thành phần được mô tả như sau:

* **Magic** (2 bytes): xác định là file 32 bit (0B 01) hay 64 bit (0B 20)
* **AddressOfEntryPoint** (4bytes): chứa địa chỉ ảo tương đối (RVA) của câu lệnh đầu tiên sẽ được thực thi khi chương trình PE loader sẵn sàng để chạy PE File (.text). Nếu muốn chương trình bắt đầu từ một địa chỉ khác (để thực thi câu lệnh với mục đích khác) thì cần thay đổi địa chỉ này về địa chỉ tương đối của câu lệnh muốn thực thi.
* **ImageBase:** địa chỉ nạp được ưu tiên cho PE File.
* **Section Alignment:** Phần liên kết của các Section trong bộ nhớ. Tức là một section luôn luôn được bắt đầu bằng bội số của sectionAlignment. Ví dụ: sectionAlignment là 1000h, section đầu tiên bắt đầu ở vị trí 401000h và kích thước là 10h, section tiếp theo sẽ bắt đầu tại địa chỉ 402000h.
* **File Alignment:** Phần liên kết của các Section trong File. Tương tự như SectionAlignment nhưng áp dụng với file.
* **SizeOfImage:** Toàn bộ kích thước của Pe image trong bộ nhớ, là tổng của tất cả các headers và sections được liên kết tới Section Alignment
* **SizeOfHeaders:** Kích thước của tất cả các headers + section table =kích thước file trừ đi tổng kích thước của các section trong file.
* **Data Directory:** là một mảng gồm 16 phần tử, trong đó mỗi phần liên quan đến một cấu trúc dữ liệu quan trọng trong PE File.

### Section Table

Section Table là thành phần ngày sau PE Header, bao gồm một mảng những cấu trúc IMAGE\_SECTION\_HEADER, mỗi phần tử chứa thông tin về một section trong PE file.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 10: Cấu trúc Section table

Thông tin về một số trường quan trọng:

* **VirtualSize:** Kích thước thật sự của dữ liệu trên section tính theo byte, giá trị này có thể nhỏ hơn kích thước trên ổ đĩa (SizeOfRawData)
* **VirtualAddress:** RVA của section, là giá trị để ánh xạ khi section được load lên bộ nhớ
* **SizeOfRawData:** Kích thước section data trên ổ đĩa
* **PointerToRawData:** là offset từ vị trí đầu file tới section data.
* **Characteristics:** đặc tính của section: thực thi, dữ liệu khởi tạo …

Nhìn lại một cách tổng thế cấu trúc PE file ở dưới đây:

A screen shot of a cell phone

AI-generated content may be incorrect.

Figure 11: Cấu trúc chi tiết của PE Header

## Relocation

1. Tệp thực thi có thể di dời nghĩa là gì?

* Một tệp thực thi (executable file) có thể được tải vào một địa chỉ ảo khác nhau mỗi lần nó chạy.
* Tuy nhiên, sau khi đã được nạp vào bộ nhớ, nó không thể bị di dời nữa.

2. Tại sao cần di dời (relocation)?

* Một tệp thực thi không hoạt động độc lập mà phải chia sẻ không gian địa chỉ với nhiều tệp thực thi khác.
* Mỗi tiến trình có không gian địa chỉ riêng, nhưng trong không gian đó có nhiều thành phần khác nhau được nạp vào:
  + Tệp .exe chính (chương trình mà người dùng chạy).
  + Các thư viện động (DLLs), do chương trình hoặc hệ điều hành tải vào.
  + Các thư viện hệ thống của Windows (do hệ điều hành nạp).
* Nếu một tệp thực thi cố định ở một địa chỉ cụ thể, nó có thể bị xung đột với các tệp khác.

3. Cách hệ điều hành quản lý việc tải tệp thực thi

* Hệ điều hành có thể nạp các tệp thực thi tại các địa chỉ ảo khác nhau mỗi lần chạy.
* Các DLLs và thư viện hệ thống cũng có thể được nạp vào các địa chỉ khác nhau tùy theo chương trình cần dùng.

4. Điều gì xảy ra sau khi chương trình đã được nạp?

* Sau khi chương trình được nạp vào bộ nhớ, nó không thể bị di dời nữa.
* Điều này có nghĩa là địa chỉ bộ nhớ của chương trình phải cố định trong suốt quá trình chạy.

5. Kết luận

* Các tệp thực thi có thể được nạp vào các địa chỉ khác nhau mỗi lần chạy để tránh xung đột địa chỉ với các tệp khác.
* Hệ điều hành quyết định địa chỉ nạp dựa trên các thành phần khác đang hoạt động trong cùng không gian địa chỉ.
* Sau khi được nạp, chương trình giữ nguyên địa chỉ và không thể thay đổi.

## Vấn đề của relocation

1. Vấn đề: Xung đột địa chỉ bộ nhớ giữa các tệp thực thi

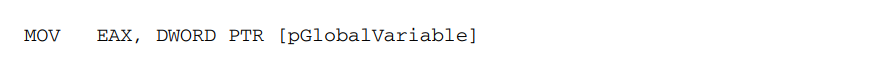
* Trong một không gian địa chỉ của tiến trình, nhiều tệp thực thi (.exe, .dll) được tải vào cùng một lúc.
* Do việc nạp không được lập kế hoạch trước, hai hoặc nhiều mô-đun có thể cố gắng sử dụng cùng một địa chỉ bộ nhớ → Gây lỗi xung đột bộ nhớ.
* Để giải quyết vấn đề này, hệ điều hành sẽ di dời (relocate) một trong các mô-đun đó, nạp nó vào một địa chỉ khác với địa chỉ ban đầu dự kiến.

2. Tại sao tệp thực thi cần biết trước địa chỉ của nó?

* Một tệp thực thi không thể đơn giản được nạp vào bất kỳ địa chỉ nào mà không có sự chuẩn bị trước.
* Lý do là trong mã máy có nhiều tham chiếu chéo (cross-references), nơi một vị trí trong mã trỏ đến một vị trí khác.

3. Ví dụ về tham chiếu chéo trong mã máy

* Giả sử chương trình cần truy cập một biến toàn cục (pGlobalVariable):



* + pGlobalVariable được lưu trong vùng dữ liệu của tệp thực thi.
  + Nhưng khi tạo tệp .exe, trình biên dịch cần xác định địa chỉ cụ thể của biến này.

4. Cách tiếp cận khả thi để giải quyết vấn đề

Cách 1: Dùng địa chỉ tương đối

* Một cách đơn giản là dùng địa chỉ tương đối, tức là tính toán địa chỉ của biến dựa trên vị trí bắt đầu của tệp thực thi.
* Điều này giúp chương trình hoạt động linh hoạt, bất kể nó được nạp vào đâu.
* Tuy nhiên, CPU thường yêu cầu địa chỉ tuyệt đối (absolute address), và việc tính toán liên tục gây giảm hiệu suất.

Cách 2: Hệ điều hành sửa đổi địa chỉ sau khi nạp (Cách thực tế sử dụng)

* Khi chương trình được tải vào bộ nhớ, Loader của hệ điều hành sẽ:
  1. Duyệt qua mã của chương trình để tìm các địa chỉ tuyệt đối.
  2. Cập nhật lại các địa chỉ này để chúng trỏ đến vị trí thực sự trong bộ nhớ.
  3. Điều này giúp chương trình hoạt động bình thường mà không cần tính toán địa chỉ trong thời gian chạy.

5. Kết luận

* Vấn đề chính: Khi nhiều tệp thực thi được tải vào bộ nhớ, có thể xảy ra xung đột địa chỉ.
* Giải pháp: Hệ điều hành di dời (relocate) một số mô-đun và cập nhật lại các địa chỉ tuyệt đối trong mã thực thi.
* Mục tiêu: Giữ hiệu suất cao, tránh việc phải tính toán địa chỉ liên tục trong thời gian chạy.

## Image Sections

Giải thích về Cấu Trúc của Tệp Thực Thi (Executable Image Sections)

1. Tệp thực thi được chia thành các vùng riêng biệt (sections)

* Khi một tệp thực thi (.exe, .dll) được tạo, nó không phải là một khối dữ liệu duy nhất, mà được chia thành các phần (sections) khác nhau.
* Lý do: Mỗi phần trong tệp có mục đích sử dụng khác nhau và cần được quản lý khác nhau trong bộ nhớ.

2. Các loại vùng (sections) chính trong tệp thực thi

1. Mã (Code Section – .text section)

* Chứa mã thực thi của chương trình.
* Được đánh dấu là chỉ đọc (Read-Only) và có thể thực thi (Executable) để tránh bị thay đổi.

1. Dữ liệu (Data Section – .data, .bss section)

* Chứa dữ liệu của chương trình, bao gồm:
  + Dữ liệu đã khởi tạo (.data section): Chứa các biến toàn cục hoặc hằng số có giá trị được gán trước.
  + Dữ liệu chưa khởi tạo (.bss section): Chứa các biến toàn cục chưa có giá trị ban đầu (mặc định là 0).
* Có thể được đánh dấu là đọc-ghi (Read-Write), nhưng không thể thực thi.

3. Cách bộ quản lý bộ nhớ (Memory Manager) xử lý các sections

* Khi chương trình được tải vào bộ nhớ, Memory Manager sẽ:
  1. Cấp quyền truy cập dựa trên cấu hình của từng section.
     + Ví dụ: .text chỉ cho phép đọc và thực thi, .data cho phép đọc-ghi.
  2. Nạp từng section vào đúng vị trí trong bộ nhớ.

4. Ví dụ về cách dữ liệu được lưu trong các sections

Xét đoạn mã sau:

char szMessage[] = "Welcome to my program!";

* Biến szMessage chứa chuỗi đã khởi tạo "Welcome to my program!".
* Trình biên dịch sẽ lưu trữ chuỗi này vào phần dữ liệu đã khởi tạo (.data section) của tệp thực thi.
* Khi chương trình chạy, giá trị này sẽ được nạp vào bộ nhớ theo đúng quyền truy cập được thiết lập.

5. Kết luận

* Tệp thực thi được chia thành nhiều vùng (sections) để giúp hệ điều hành quản lý bộ nhớ hiệu quả hơn.
* Mã chương trình và dữ liệu được lưu trữ ở các sections khác nhau để bảo vệ chống ghi đè hoặc thực thi mã không mong muốn.
* Bộ quản lý bộ nhớ đặt quyền truy cập phù hợp với từng loại dữ liệu trong chương trình.

## Section Alignment

1. Vấn đề về căn chỉnh (Alignment) trong bộ nhớ và trên đĩa

* Khi một tệp thực thi (.exe, .dll) được tải vào bộ nhớ, nó không thể được nạp tùy ý vào bất kỳ địa chỉ nào.
* Hệ điều hành phải căn chỉnh (align) các phần (sections) của tệp theo kích thước trang bộ nhớ (page size) để đảm bảo đúng quyền truy cập và hiệu suất.
* Tuy nhiên, trên đĩa, việc căn chỉnh theo kích thước trang bộ nhớ sẽ gây lãng phí dung lượng vì có nhiều khoảng trống không sử dụng.

2. Hai loại căn chỉnh trong tệp thực thi PE (Portable Executable)

Trong phần header của tệp PE, có hai loại căn chỉnh chính:

🔹 Section Alignment (Căn chỉnh trên bộ nhớ)

* Đây là cách các phần (sections) được căn chỉnh khi tệp thực thi được nạp vào bộ nhớ.
* Thường được căn chỉnh theo kích thước trang bộ nhớ (thường là 4KB hoặc 64KB).
* Quan trọng vì bộ quản lý bộ nhớ của hệ điều hành cấp phát bộ nhớ theo từng trang (pages).

🔹 File Alignment (Căn chỉnh trên đĩa)

* Đây là cách các phần được căn chỉnh trong tệp thực thi trên đĩa.
* Thường có kích thước nhỏ hơn section alignment, để tránh lãng phí dung lượng trên đĩa (ví dụ: 512 bytes hoặc 4KB).
* Giúp tệp có kích thước nhỏ gọn hơn mà vẫn đảm bảo có thể nạp vào bộ nhớ đúng cách.

3. Vấn đề về RVA (Relative Virtual Address) và Alignment

* RVA (Relative Virtual Address) là địa chỉ tương đối so với nơi chương trình được nạp vào bộ nhớ.
* Vấn đề xảy ra khi truy cập trực tiếp tệp thực thi trên đĩa mà không tính đến sự khác biệt giữa section alignment (trên bộ nhớ) và file alignment (trên đĩa).
* Vì RVA được tính toán dựa trên section alignment, nếu mở tệp thực thi trực tiếp trên đĩa, địa chỉ có thể không đúng.

4. Tóm lại

* Section Alignment: Định nghĩa cách sections được căn chỉnh khi nạp vào bộ nhớ (thường theo kích thước trang).
* File Alignment: Định nghĩa cách sections được lưu trên đĩa (nhỏ hơn để tiết kiệm dung lượng).
* RVA sử dụng Section Alignment, nên khi truy cập tệp trên đĩa, có thể cần điều chỉnh để tìm đúng dữ liệu.

## Dynamically Linked Libraries (DLLs)

1. Khái niệm về DLL (Thư viện liên kết động)

* DLL (Dynamic Link Library) là một thành phần quan trọng trong hệ điều hành Windows.
* Thay vì gộp toàn bộ mã chương trình vào một tệp thực thi duy nhất (.exe), chương trình có thể được chia nhỏ thành nhiều thư viện DLL.
* Mỗi DLL chịu trách nhiệm cho một chức năng cụ thể và chỉ được nạp vào bộ nhớ khi cần thiết.

2. Lợi ích của việc sử dụng DLL

✔ Tiết kiệm bộ nhớ

* DLL chỉ được tải khi cần, giúp giảm mức tiêu thụ bộ nhớ hệ thống.
* Windows có thể chia sẻ DLL giữa các chương trình thay vì nạp lại nhiều lần cùng một thư viện vào các vùng nhớ khác nhau.

✔ Dễ dàng bảo trì và nâng cấp

* Nếu một chức năng trong chương trình cần thay đổi, chỉ cần cập nhật DLL thay vì sửa đổi toàn bộ chương trình.

✔ Tăng tính linh hoạt

* Một chương trình có thể sử dụng các thư viện DLL của hệ điều hành hoặc bên thứ ba để mở rộng chức năng.

3. So sánh DLL với thư viện tĩnh (.lib - Static Library)

🔹 Thư viện tĩnh (.lib)

* Khi biên dịch, mã từ thư viện .lib được nhúng trực tiếp vào tệp thực thi.
* Nếu hai chương trình sử dụng cùng một thư viện .lib, mã của thư viện đó sẽ bị nhân bản trong bộ nhớ khi chạy.

🔹 Thư viện động (DLL)

* Mã của DLL không được nhúng vào tệp thực thi, thay vào đó, chương trình chỉ gọi đến DLL khi cần.
* Hệ điều hành có thể chia sẻ một DLL giữa nhiều chương trình, tiết kiệm bộ nhớ hệ thống.

4. Hai phương pháp liên kết DLL trong runtime

Trong Windows, có hai cách để chương trình sử dụng DLL:

🔹 Static Linking (Liên kết tĩnh - Không nhầm với liên kết tĩnh trong build-time!)

* Khi một chương trình được biên dịch, nó sẽ có một bảng nhập (Import Table) liệt kê tất cả các DLL mà nó cần.
* Khi chương trình chạy, Windows Loader sẽ tự động nạp tất cả DLL được liệt kê và ánh xạ các hàm từ DLL vào chương trình.
* Ưu điểm: Đơn giản, dễ sử dụng.
* Nhược điểm: Nếu một DLL cần thiết bị thiếu hoặc bị lỗi, chương trình có thể không khởi động được.

🔹 Runtime Linking (Liên kết động lúc chạy - Dynamic Loading)

* Chương trình tự quyết định thời điểm nạp DLL bằng cách sử dụng các hàm như LoadLibrary() hoặc GetProcAddress().
* Ưu điểm: Linh hoạt, có thể thay đổi DLL trong runtime.
* Nhược điểm: Phức tạp hơn, vì lập trình viên phải tự xử lý việc nạp và gọi hàm từ DLL.

5. Ứng dụng trong Reverse Engineering

* Static Linking dễ phân tích hơn vì danh sách các hàm được gọi từ DLL được hiển thị rõ ràng trong bảng nhập (Import Table).
* Runtime Linking khó phân tích hơn, vì chương trình chỉ tải DLL khi cần và tìm hàm theo tên hoặc địa chỉ, khiến việc dò tìm khó khăn hơn.

Kết luận

* DLL giúp giảm mức tiêu thụ bộ nhớ, nâng cao khả năng bảo trì và mở rộng chức năng của chương trình.
* Static Linking thì dễ sử dụng nhưng thiếu linh hoạt, còn Runtime Linking linh hoạt nhưng phức tạp hơn.
* Trong lĩnh vực phân tích phần mềm (reverse engineering), Static Linking dễ theo dõi hơn, còn Runtime Linking cần phân tích sâu hơn để xác định các hàm được gọi trong runtime.

# THE I/O SYSTEM

## Hệ thống I/O (I/O System) trong Windows

Hệ thống I/O (Input/Output System) trong Windows là tập hợp các thành phần trong nhân (kernel) chịu trách nhiệm quản lý trình điều khiển thiết bị (device drivers) và giao tiếp giữa các ứng dụng với phần cứng. Nó đóng vai trò trung gian, giúp các ứng dụng có thể gửi yêu cầu đến thiết bị phần cứng thông qua trình điều khiển một cách nhất quán và hiệu quả.

## Chức năng chính của hệ thống I/O

1. **Quản lý trình điều khiển thiết bị**
   * Khi một trình điều khiển thiết bị được cài đặt vào hệ thống, nó sẽ đăng ký với hệ thống I/O.
   * Việc đăng ký này giúp hệ điều hành biết trình điều khiển nào sẽ xử lý các yêu cầu liên quan đến thiết bị đó.
2. **Chuyển tiếp yêu cầu từ ứng dụng đến trình điều khiển thiết bị**
   * Khi một ứng dụng muốn thao tác với thiết bị (ví dụ: đọc/ghi file, in tài liệu, giao tiếp với bàn phím), nó sẽ gửi yêu cầu đến hệ thống I/O.
   * Hệ thống I/O sẽ chuyển yêu cầu này đến trình điều khiển thích hợp để thực hiện.
3. **Hỗ trợ yêu cầu chung và yêu cầu đặc thù của thiết bị**
   * **Yêu cầu chung**: Các thao tác cơ bản như đọc/ghi file trên hệ thống tệp (file system) hoạt động một cách nhất quán, không phụ thuộc vào loại thiết bị lưu trữ.
   * **Yêu cầu đặc thù của thiết bị**: Một số thiết bị có thể có các yêu cầu riêng, chẳng hạn như điều chỉnh độ sáng màn hình hoặc thay đổi chế độ hoạt động của ổ đĩa.

## Mô hình phân tầng (layered architecture)

Hệ thống I/O của Windows được thiết kế theo mô hình nhiều lớp (layered), nghĩa là một thiết bị có thể được quản lý bởi nhiều trình điều khiển xếp chồng lên nhau.

Ví dụ:

* **Lớp trình điều khiển hệ thống tệp (File System Driver)**: Quản lý việc đọc/ghi dữ liệu nhưng không quan tâm thiết bị lưu trữ cụ thể là gì (HDD, SSD, USB).
* **Lớp trình điều khiển lưu trữ (Storage Driver)**: Quản lý phần cứng lưu trữ nhưng không quan tâm đến hệ thống tệp sử dụng trên thiết bị.

Hệ thống I/O đảm bảo rằng các trình điều khiển này có thể hoạt động cùng nhau mà không cần chỉnh sửa đặc biệt, miễn là chúng tuân thủ các giao diện tiêu chuẩn của hệ thống I/O.

## Lợi ích của kiến trúc phân tầng trong hệ thống I/O

✅ **Tính linh hoạt**: Cho phép thay thế hoặc nâng cấp từng phần mà không ảnh hưởng đến các thành phần khác. Ví dụ, bạn có thể cài đặt hệ thống tệp mới mà không cần thay đổi trình điều khiển ổ cứng.

✅ **Khả năng mở rộng**: Hỗ trợ nhiều loại thiết bị và giao thức khác nhau mà không cần sửa đổi toàn bộ hệ thống.

✅ **Tăng tính tương thích**: Các ứng dụng có thể hoạt động với nhiều loại thiết bị mà không cần thay đổi mã nguồn, vì hệ thống I/O đảm nhiệm việc tương thích.

✅ **Cải thiện hiệu suất**: Hệ thống có thể tối ưu hóa các hoạt động I/O bằng cách sử dụng bộ đệm (buffering), sắp xếp thứ tự yêu cầu, và quản lý tài nguyên một cách hiệu quả.

## Filter Driver

### Định nghĩa

Hệ thống I/O của Windows có kiến trúc phân tầng (layered architecture), giúp dễ dàng bổ sung các **trình điều khiển lọc (filter drivers)**. Đây là các lớp bổ sung có nhiệm vụ theo dõi hoặc thay đổi luồng dữ liệu giữa ứng dụng và trình điều khiển thiết bị hoặc giữa các trình điều khiển với nhau.

### Chức năng của Filter Drivers

Filter drivers có thể thực hiện các tác vụ quan trọng như:  
✅ **Giám sát dữ liệu**: Theo dõi luồng dữ liệu giữa ứng dụng và thiết bị để thu thập thông tin. Ví dụ: Một chương trình bảo mật có thể sử dụng filter driver để giám sát các file đang được truy cập.

✅ **Chỉnh sửa hoặc xử lý dữ liệu**: Có thể thay đổi dữ liệu trước khi nó đến trình điều khiển đích. Ví dụ:

**Trình điều khiển mã hóa file (File Encryption Driver)**: Mã hóa dữ liệu trước khi lưu vào ổ đĩa.

**Trình điều khiển nén file (File Compression Driver)**: Nén dữ liệu để tiết kiệm dung lượng.

✅ **Cải thiện bảo mật**: Có thể chặn hoặc cho phép các hoạt động dựa trên chính sách bảo mật. Ví dụ: Một phần mềm chống virus có thể sử dụng filter driver để chặn file đáng ngờ trước khi nó được thực thi.

### Ứng dụng của Filter Drivers trong Reverse Engineering

Trong phân tích phần mềm (reverse engineering), filter drivers thường được sử dụng để **giám sát** và **trích xuất thông tin** từ chương trình mục tiêu.

📌 **Cách hoạt động**:

Các công cụ phân tích có thể **chèn mã lọc đặc biệt** vào hệ thống I/O để ghi lại luồng dữ liệu.

Tùy vào vị trí đặt filter driver, ta có thể thu thập dữ liệu khác nhau:

**Giám sát hệ thống file** → Xem chương trình đang đọc/ghi file nào.

**Giám sát mạng** → Xem chương trình gửi/nhận dữ liệu qua giao thức nào.

**Giám sát thiết bị ngoại vi** → Xem cách chương trình tương tác với phần cứng.

🛠 **Ví dụ về công cụ giám sát sử dụng filter drivers:**

**Process Monitor (ProcMon)** → Theo dõi hoạt động của chương trình trên hệ thống file, registry và mạng.

**Wireshark** → Giám sát gói tin mạng bằng cách lọc dữ liệu ở các tầng giao thức khác nhau.

### Tầm quan trọng của vị trí Filter Driver trong hệ thống I/O

Vị trí của filter driver trong **ngăn xếp I/O (I/O Stack)** ảnh hưởng trực tiếp đến loại dữ liệu mà nó có thể thu thập hoặc chỉnh sửa.

📌 **Ví dụ trong giám sát mạng:**

**Filter driver nằm trên tầng giao thức cao (như TCP)** → Nhận dữ liệu ứng dụng gửi đi mà **không có** tiêu đề TCP, IP, hoặc Ethernet.

**Filter driver nằm ở tầng mạng thấp (Network Interface Layer)** → Nhận được **toàn bộ** dữ liệu, bao gồm tiêu đề TCP, IP, Ethernet.

📌 **Ví dụ trong giám sát hệ thống file:**

**Filter driver ở tầng hệ thống file** → Theo dõi các file đang được mở, đọc hoặc ghi.

**Filter driver ở tầng thiết bị lưu trữ** → Nhận dữ liệu thô từ ổ đĩa, nhưng không biết file nào đang được truy cập.

Do đó, để **thu thập đúng dữ liệu**, cần lựa chọn công cụ giám sát có filter driver hoạt động ở mức phù hợp.

## Win32 Subsystem

Hệ thống con Win32 là thành phần chịu trách nhiệm về toàn bộ giao diện người dùng của Windows. Nó bao gồm:

* GDI (Graphics Device Interface) – Xử lý đồ họa cấp thấp.
* USER – Quản lý các thành phần GUI cấp cao như cửa sổ, menu và xử lý đầu vào từ người dùng.

Bên trong hệ thống con Win32, hệ thống con Win32 không chịu trách nhiệm toàn bộ API của Win32, mà chỉ quản lý USER và GDI. Các API cơ bản (BASE API) từ KERNEL32.DLL được gọi trực tiếp vào Native API, không liên quan đến hệ thống con Win32.