背景知識

IRQ 和中斷結構（包括 PIC 中斷結構、APIC 中斷結構等）
中斷向量編號（interrupt vector numbers）/中斷向量
中斷向量表/中斷描述器表格（IDT, Interrupt Descriptor Table）
硬體處理中斷事件和例外事件的方式

Windows 引發 Trap 分派的種類

例外

- 指程式執行過程中，執行了某些指令引發的同步狀態/特殊事件。
  在相同環境下再執行一次同一個程式、輸入完全相同的資料，就能重現例外。
- 例外來源：
  硬體產生的，如：除零錯誤和分頁錯誤。
  軟體產生的，如：程式中的中斷點事件。

中斷

- 跟處理器目前工作內容無關的非同步事件。
  任何時候均可能發生。
  爲並行性來源。
- 中斷種類：
  硬體中斷－由 I/O 裝置引發的。
  軟體插（中）斷－由 Windows 核心程式碼產生的，包括：Dispatch、DPC、APC。
- 用於處理下列工作：
  觸發 thread 分派。
  不急著完成的中斷處理工作。
  處理計時器到期。
  .在選定的 thread context 執行非同步程式。
  支援非同步 I/O 動作。

Windows 如何在系統層運用處理器的中斷能力

- 本機 APIC 的初始化工作 HAL 模組中完成。
- IDT 初始化工作：在 Windows 核心中完成
  （base\ntos\ke\i386\newsysbg.asm 檔案中的 KiSystemStartup 函式）。
  每個處理器的控制資料區 KPCR 結構的 IDT 成員指向該 CPU 的 IDT 所在位址。
  （當 CPU 執行於核心 kernel space 時，透過節區暫存器 FS 指向的節區描述器可獲得 CPU 的 KPCR 結構的位址）
  ntldr 在將控制權交給核心模組 ntoskrnl.exe 以前，IDT 設置完成。
- 中斷處理常式和例外處理常式定義於核心模組。
- 核心對中斷和例外分別定義了不同的系統架構：

- 提供可擴充的分發例外機制－
  允許核心偵錯器或行程偵錯器、核心或應用程式本身，以及環境子系統都有機會處理例外。
- 提供彈性的處理中斷機制－
  提供裝置驅動程式一種無需操縱 IDT、具可移植的途徑的中斷處理及擴充機制－稱中斷物件機制
  透過此機制，裝置驅動程式將一個中斷服務常式（ISR, Interrupt Service Routine）與某一特定的中斷向量關聯。
  裝置驅動程式爲它的裝置的中斷物件註冊一個 ISR（中斷服務常式）。
  裝置驅動程式透過註冊中斷物件的方式，將中斷服務常式添加到系統中。
  透過此機制允許硬體裝置共用相同的硬體中斷向量。
  Windows 核心本身使用組合語言編寫的中斷常式（不使用中斷物件），處理與系統硬體平臺緊密相關的中斷。
  包括計時器中斷、電源失敗、機器錯誤檢查，以及內部使用的一些軟體插斷。

例外分發

在 Intel x86 系統架構中，例外是透過 IDT 分發（與中斷相同）

- 中斷向量 0~0xlf 除 Intel 保留的部份及 2 號保留給 NMI（不可遮罩中斷）以外，其餘均用於處理各種條件引發的例外
- Windows 的例外處理常式爲_KiTrap??的組譯函式
  其中??是十六進位的兩位中斷向量（Intel 保留的部份均爲 0F）
  如：_KiTrap0E 是分頁錯誤的例外處理常式，其中 0x0E 是分頁錯誤的中斷向量。
  函式碼：KiTrap??位於 base\ntos\ke\i386\trap.asm 檔案
- WRK 中 IDT 中斷向量 0~0xlf 的初始定義(包含了相關例外的入口處理常式的位址)如下：

```
_IDT            label byte

IDTEntry        _KiTrap00, D_INT032             ; 0: Divide Error
IDTEntry        _KiTrap01, D_INT032             ; 1: DEBUG TRAP
IDTEntry        _KiTrap02, D_INT032             ; 2: NMI/NPX Error
IDTEntry        _KiTrap03, D_INT332             ; 3: Breakpoint
IDTEntry        _KiTrap04, D_INT332             ; 4: INTO
IDTEntry        _KiTrap05, D_INT032             ; 5: BOUND/Print Screen
IDTEntry        _KiTrap06, D_INT032             ; 6: Invalid Opcode
IDTEntry        _KiTrap07, D_INT032             ; 7: NPX Not Available
IDTEntry        _KiTrap08, D_INT032             ; 8: Double Exception
IDTEntry        _KiTrap09, D_INT032             ; 9: NPX Segment Overrun
IDTEntry        _KiTrap0A, D_INT032             ; A: Invalid TSS
IDTEntry        _KiTrap0B, D_INT032             ; B: Segment Not Present
IDTEntry        _KiTrap0C, D_INT032             ; C: Stack Fault
IDTEntry        _KiTrap0D, D_INT032             ; D: General Protection
IDTEntry        _KiTrap0E, D_INT032             ; E: Page Fault
IDTEntry        _KiTrap0F, D_INT032             ; F: Intel Reserved

IDTEntry        _KiTrap10, D_INT032             ;10: 486 coprocessor error
IDTEntry        _KiTrap11, D_INT032             ;11: 486 alignment
IDTEntry        _KiTrap0F, D_INT032             ;12: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032             ;13: XMMI unmasked numeric
exception
IDTEntry        _KiTrap0F, D_INT032             ;14: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032             ;15: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032             ;16: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032             ;17: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032             ;18: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032             ;19: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032             ;1A: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032             ;1B: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032             ;1C: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032             ;1D: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032             ;1E: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032             ;1F: Reserved for APIC
```

Windows 提供的例外處理常式針對各種不同的例外條件，分爲三種處理方式：

- 交由系統處理（處理過程簡單）
- 交由系統分發（處理過程複雜，其中涉及核心中多個組件）
- 傳遞給有關的核心元件，或交給使用者模式的元件（使用者模式程式碼）來處理
  －此爲 Windows 系統之例外分發機制

NOTE

採用 Windows 系統之例外分發機制，由於使用者模式程式碼有機會參與例外處理過程，這

使得系統例外機制可以與程式語言提供的例外機制連接起來。
如：在 c++程式中，透過 try-catch 語法可以捕捉到 DIV 組譯指令產生的例外

Windows 處理 Intel x86 處理器定義的例外的方式：

向量	名稱	例外處理說明
0	除法錯誤	交由系統分發
1	偵錯	交由系統分發
2	NMI 中斷	系統處理，嚴重錯誤
3	中斷點	交由系統分發
4	溢滿	交由系統分發
5	BOUND 越界	核心模式例外，則系統結束；使用者模式例外，則交由系統分發
6	無效操作碼	核心模式例外，則系統結束；使用者模式例外，則交由系統分發
7	輔助處理器不可用	核心模式例外由系統處理，使用者模式例外交由系統分發
8	雙重錯誤	嚴重錯誤（bugcheck ），系統終止
9	輔助處理器區段溢滿	僅在 80286 上發生，系統終止
10	無效 TSS	系統處理 終止行程（若使用者模式例外）或系統終止（若核心模式例外）
11	區段不存在	核心模式例外，系統終止；使用者模式例外，則交由系統分發
12	堆疊段錯誤	交由系統分發
13	一般保護錯誤	系統處理區段選取器不正確的情形，其他情形交由系統分發
14	分頁錯誤	缺頁情形由系統處理；若出錯程式碼的權限不夠，則由系統分發
16	x87 浮點	交由系統分發
17	對齊檢查	在 Intel x86 上，僅使用者模式下可能產生此例外，交由系統分發
18	機器檢查	若使用者模式例外，則終止行程；若核心模式例外，則終止系統
19	SIMD 浮點例外	交由系統分發

Windows 系統之結構化例外處理（SHE, Structured Exception Handling）機制

- 爲一種基於呼叫框架爲基礎的例外處理機制（技術），將例外處理常式（爲一塊特定的程式碼，用以處理某種特定型別的例外）與 "堆疊框架（stack frame）" 關聯起來。
  即於緒程的堆疊框架（stack frame）中包含（或關聯）一個或多個例外處理常式，因而當發生例外時，例外發送器（如：RtlDispatchException 或 KiUserExceptionDispatcher）得以根據（沿著）目前堆疊的堆疊框架來搜尋與之關聯的例外處理常式。
- Windows 核心和使用者模式程式碼都支援基於呼叫框架爲基礎的例外處理機制。
  核心模式程式碼，由 RtlDispatchException 函式完成。（base\ntos\rtl\i386\exdsptch.c 檔案的 126-364 行）
  使用者模式程式碼，由 ntdll.dll 中的 KiUserExceptionDispatcher 函式完成。

NOTE
Windows XP 以後，KiUserExceptionDispatcher 的處理流程除 SHE 機制外，存在另一種行程範圍的例外處理機制，向量化例外處理(VHE, Vectored Exception Handling)

Windows 系統之例外分發機制

- IDT 中的例外處理常式_KiTrap??透過 jmp 指令跳轉到共同的例外分發的入口處
- 共同的例外分發入口是組譯程式 CommonDispatchException
  根據不同的參數情形，跳轉點有所不同（比如 CommonDispatchException0Args）
  （base\ntos\ke\i386\trap.asm 檔案的 2019-2090 行）
- 共同的例外分發入口 CommonDispatchException 組譯程式的動作：
  在目前堆疊中分配一個例外記錄（exception record）
  利用例外處理常式_KiTrap??指定的參數資訊初始化該例外記錄
  呼叫例外分發常式 KiDispatchException（C 函式）

例外記錄 EXCEPTION_RECORD 的定義：

```
typedef struct _EXCEPTION_RECORD {
    NTSTATUS ExceptionCode;                          //例外碼
    ULONG ExceptionFlags;                            //例外旗標
    struct EXCEPTION_RECORD *ExceptionRecord;        //相關聯的例外記錄
    PVOID ExceptionAddress;                          //發生例外的指令位元元元址
    ULONG NumberParameters;                          //參數數目
    ULONG_PTR ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS] ; //參數陣列
} EXCEPTION_RECORD;
```

例外處理常式_KiTrap??向例外分發常式 KiDispatchException 提供參數資訊的方式：

例外處理常式在入口處透過 ENTER_TRAPE 巨集爲目前發生的例外建立一個陷阱框架。

陷阱框架的 C 語言定義如下：（適用於例外與中斷）

```
typedef struct _KTRAP_FRAME {
    ULONG   DbgEbp;
    ULONG   DbgEip;
    ULONG   DbgArgMark;
    ULONG   DbgArgPointer;
    ULONG   TempSegCs;
    ULONG   TempEsp;
    ULONG   Dr0;
    ULONG   Dr1;
    ULONG   Dr2;
    ULONG   Dr3;
    ULONG   Dr6;
    ULONG   Dr7;
    ULONG   SegGs;
    ULONG   SegEs;
    ULONG   SegDs;
    ULONG   Edx;
    ULONG   Ecx;
    ULONG   Eax;
    ULONG   PreviousPreviousMode;
    PEXCEPTION_REGISTRATION_RECORD ExceptionList;
    ULONG   SegFs;
    ULONG   Edi;
    ULONG   Esi;
    ULONG   Ebx;
    ULONG   Ebp;
    ULONG   ErrCode;              /* 硬體填充 */
    ULONG   Eip;                  /* 硬體填充 */
    ULONG   SegCs;                /* 硬體填充 */
    ULONG   EFlags;               /* 硬體填充 */
    ULONG   HardwareEsp;          /* 硬體填充，僅當有模式切換時才使用 */
    ULONG   HardwareSegSs;        /* 硬體填充，僅當有模式切換時才使用 */
    ULONG   V86Es;                /* 硬體填充，僅當從 V86 模式切換過來時才使用 */
    ULONG   V86Ds;                /* 硬體填充，僅當從 V86 模式切換過來時才使用 */
    ULONG   V86Fs;                /* 硬體填充，僅當從 V86 模式切換過來時才使用 */
    ULONG   V86Gs;                /* 硬體填充，僅當從 V86 模式切換過來時才使用 */
} KTRAP_FRAME;

typedef KTRAP_FRAME *PKTRAP_FRAME;
typedef KTRAP_FRAME *PKEXCEPTION_FRAME;
```

ENTER_TRAPE 巨集完成如下：（base\ntos\ke\i386\kimacro.inc 檔案的 744-831 行）

其中陷阱框架中的高位址部分是由硬體放入，包括錯誤碼、EIP、CS 和 EFLAGS

若爲使用者模式例外，還包含例外發生前的堆疊資訊

若爲虛擬 86 模式例外，則另包含虛擬 86 所用的區段暫存器

將例外發生時刻的主要暫存器存放到堆疊中，包括：

non-volatile registers（ebp、ebx、esi 和 edi）

區段暫存器 fs

目前處理器的例外串列（位於 KPCR 的開頭）

發生例外時的處理器模式（核心模式或使用者模式）

volatile registers（eax、ecx 和 edx）

區段暫存器 ds、es 和 gs

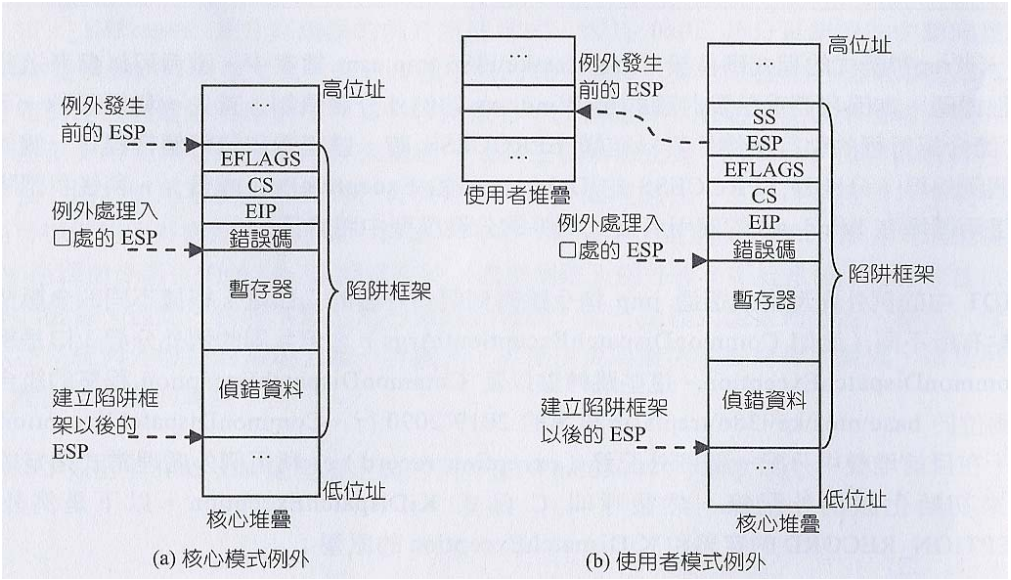
一些偵錯資料

NOTE

Non-volatile register（非易失性寄存器）

副程式呼叫前內容必須被保存的暫存器。即副程式被呼叫（改變此暫存器的值）之前，其舊的值必須保存堆疊中，並在返回之前恢復舊值。

Intel x86 處理器在例外（或中斷）發生時的堆疊如下圖所示：



。

例外處理常式的入口程式碼得知發生例外的指令是在核心模式或在使用者模式的方式：

常式利用陷阱框架中的 SegCS，指到發生例外時的指令所在的區段，來確定處理器模式。

SegCS 的 0~1 位元代表了區段選取器的特權等級，即例外發生時 CPU 目前的特權等級（CPL），由此可確定處理器的模式。

（SegCS 即 cs 程式節區暫存器，其 segment selector 的 index 指向含有程式指令的一個節區的 descriptor 其在 GDT 中的 index）

Windows 核心模式之例外分發中心－KiDispatchException 函式

（base\ntos\ke\exceptn.c 檔案的 1033-1425 行）

I. 呼叫 KeContextFromKframes(TrapFrame,ExceptionFrame,&ContextFrame)，透過已建立的例外記錄及參數傳入的陷阱框架等資訊，將例外發生時的執行環境訊息整合記錄於資料型別爲 CONTEXT 結構的 ContextFrame 變數中。

NOTE

核心核式之例外分發常式 KiDispatchException 函式原型：

```
VOID KiDispatchException (
    IN PEXCEPTION_RECORD ExceptionRecord,
    IN PKEXCEPTION_FRAME ExceptionFrame,
    IN PKTRAP_FRAME TrapFrame,
    IN KPROCESSOR_MODE PreviousMode,
    IN BOOLEAN FirstChance
    );
```

NOTE

ExceptionFrame 爲 NULL，這是 KEXCEPTION_FRAME 結構指標，爲 PowerPC 晶片設計的，用於保存附加寄存器的內容，x86 架構的處理器晶片無此需求。

NOTE

如果是中斷點例外，則修正 ContextFrame 變數中的 elp 成員，使它指向中斷點處的指令；
如果是因爲執行權限問題而發生的存取違例（例外狀態碼爲 KI_EXCEPTION_ACCESS_VIOLATION，在分頁錯誤 0x0E 例外中的情形）則需要檢查是否由於 ATL thunk 而造成了例外。
這裡的 ATL thunk 是一種程式設計技巧，其作法是，透過一個 thunk 物件（是個資料物件，但其實包含少量指令）將 Windows 視窗程式的視窗控制碼以極其高效率的方式變換成 C++程式中的 this 指標。

II. 根據發生例外時的處理器模式分別進行以下處理：

核心模式例外－發生例外的指令是在核心模式

i. 交給核心偵錯器處理該例外－第一次機會處理該例外（FirstChance 參數爲 TRUE）
若核心偵錯器處理了該例外，則返回，發生例外的指令流繼續執行；
若不存在核心偵錯器或核心偵錯器沒有處理該例外，則繼續下列步驟。

ii. 呼叫 RtlDispatchException 函式，將該例外分發到一個基於呼叫框架爲基礎的例外處理常式（call frame-based exception handler）。
只要能找到一個例外處理常式能處理此例外，則返回，發生例外的指令流繼續執行；否則繼續下列步驟。

iii. 交給核心偵錯器處理該例外－第二次機會處理該例外（FirstChance 參數爲 FALSE）
若核心偵錯器處理了該例外，則返回，發生例外的指令流繼續執行；
若不存在核心偵錯器或核心偵錯器沒有處理該例外，則繼續下列步驟。

iv. 進入錯誤檢查（bugcheck）。

v. 若例外仍未被處理，系統當機。

使用者模式例外－發生例外的指令是在使用者模式

i. 交給核心偵錯器處理該例外－第一次機會處理該例外（FirstChance 參數爲 TRUE）
若核心偵錯器處理了該例外，則返回，發生例外的指令流繼續執行；
若不存在核心偵錯器、核心偵錯器沒有處理該例外且認爲需要將例外交予使用者空間之行程偵錯器處理，則繼續下列步驟。

ii. 呼叫 DbgkForwardException 函式，將例外交給使用者空間之行程偵錯器處理
若發生例外的行程的偵錯埠不爲空，則發送一個訊息至偵錯埠，然後等待應答。
若行程偵錯器處理了例外，則返回，發生例外的指令流繼續執行。
若不存在行程偵錯器或行程偵錯器沒有處理該例外，則繼續下列步驟。

iii. KiDispatchException 函式於完成下列工作後返回：

◆ 把例外訊息整合記錄後放到使用者堆疊中
將陷阱框架轉換成型別爲 CONTEXT 結構之變數中，並複製到使用者堆疊。

- ◆ 設定使用者模式的 eip 爲使用者模式的例外分發常式 KeUserExceptionDispatcher 函式指標
  （指向 ntdll.dll 中的 KiUserExceptionDispatcher 函式，於行程管理子系統初始化時設置）
- ◆ 預備 KeUserExceptionDispatcher 函式所需的參數。

iv. 執行 KiUserExceptionDispatcher 函式，將該例外分發給一個基於呼叫框架爲基礎的例外處理常式。
只要能找到一個例外處理常式能處理此例外，則返回，發生例外的指令流繼續執行；如果該例外仍然未被處理，則繼續下列步驟。

v. KiUserExceptionDispatcher 函式透過呼叫 NtRaiseException 系統服務（呼叫 API 函式 RaiseException 產生軟體例外），將控制權切回核心位址空間。

vi. NtRaiseException 呼叫核心 KiRaiseException 函式，透過 KiRaiseException 呼叫 KiDispatchException 函式控制權交回核心模式的例外分發常式。

vii. 判斷爲第二次機會處理該例外（FirstChance 參數爲 FALSE），呼叫 DbgkForwardException 函式，將例外交給使用者空間之行程偵錯器處理。
若發生例外的行程的偵錯埠不爲空，則發送一個訊息至偵錯埠，然後等待應答。
若行程偵錯器處理了例外，則返回，例外的指令流繼續執行。
若不存在行程偵錯器或行程偵錯器沒有處理該例外，則繼續下列步驟。

viii. 發送一個訊息至行程例外埠，然後等待應答。
若連接例外埠的環境子系統（Win32 子系統）處理了該例外（表引發該例外的緒程的行程屬於此環境子系統所管轄），則返回，發生例外的指令流繼續執行；否則，繼續下列步驟。

ix. 若例外仍未被處理，則引發該例外的緒程所在的行程行程被終止。

NOTE
偵錯埠和例外埠爲行程的執行體物件資料結構 EPROCESS 中的二個成員，分別是 DebugPort 和 ExceptionPort 成員。

NOTE
將例外交給核心偵錯器是 Kernel Debugger（其爲核心的一部份）是透過呼叫 KiDebugRoutine 函式處理。
向偵錯埠或例外埠發送訊息是透過 DbgkForwardException 函式完成。
（base\ntos\dbgk\dbgkport.c 檔案）

NOTE
關於軟體例外的建構和處理過程，讀者可以參考 NtRaiseException 和 KiRaiseException 函式的實作。（分別位於 base\ntos\ke\i386\trap.asm 和 base\ntos\ke\raisexcp.c 檔案）

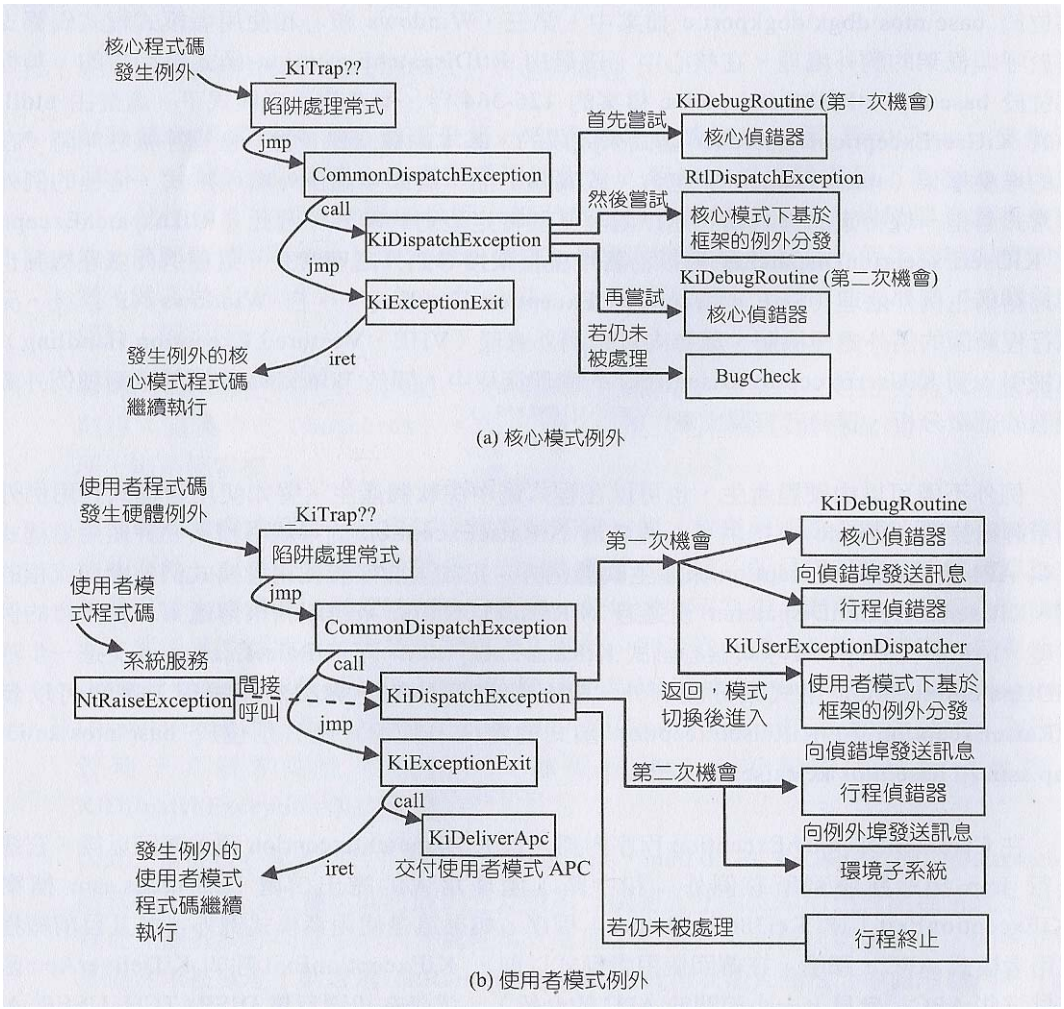
NOTE
Windows 平臺的錯誤處理機制和錯誤報告機制 (WER, Windows Error Reporting)
透過在一個緒程的堆疊頂端插入一個特殊的例外處理常式，從而捕捉到未被系統或應用程

式處理的例外。

例外處理後發生例外的指令流之返回

- 由核心模式的例外分發常式 KiDispatchException 函式返回共同的例外分發入口 CommonDispatchException 組譯程式。
- 透過執行 jmp _KiExceptionExit（或 Kei386EoiHelper）跳到所有例外（和中斷）處理常式的總出口處，即 traps.asm 檔案的_KiExceptionExit（或 Kei386EoiHelper）。
- 若是使用者模式例外，並且目前緒程有使用者模式 APC，在傳回使用者模式以前，_KiExceptionExit 呼叫 KiDeliverApc 函式交付這些 APC（透過組譯巨集 DISPATCH_USER_APC 中完成）
- 恢復必要的暫存器。
- 若是核心模式例外，需調整 esp 暫存器的值。
- 透過 iret 指令回到例外發生處。
  （上述最後三個動作爲在組譯巨集 EXIT_ALL 中完成，EXIT_ALL 亦用於系統服務之結束處理）

Windows 的例外發生和分發流程如下圖所示：



中斷要求層級（IRQL）

APIC 中斷控制器提供了中斷優先層級支援。

24 根中斷接腳沒有優先層級區別，透過編寫程式可設定每根接腳的優先層級及中斷編號。

Windows 定義自己的中斷優先層級方案，屬邏輯中斷層級，稱爲中斷要求層級（IRQL, Interrupt ReQuest Level）。

- Windows 使用 0~31（Intel x86 系統）表示中斷優先層級；
  數值越大，優先層級越高（3~26 除外）
  其中 0 爲一般 thread 執行
  1~2 爲軟體插斷
  3~31 爲硬體中斷
- 軟體插斷或非中斷程式碼的 IRQL 是在核心中管理。
  硬體中斷在 HAL 中被對應到對應的 IRQL
  HAL 模組實作 IRQL 與硬體中斷控制器的對應，控制硬體中斷優先層級。

Intel x86 系統的 IRQL 分配方案

31	HIGH_LEVEL（高）	硬體中斷
30	POWER_LEVEL（電源）	
29	IPI_LEVEL（處理器間中斷）	
28	CLOCK_LEVEL（時鐘）	
27	PROFILE_LEVEL（效能剖析）	
26	DIRQL（裝置）	
…	…	
3	DIRQL（裝置）	
2	DISPATCH_LEVEL（緒程調度&DPC）	→軟體中斷
1	APC_LEVEL（非同步行程 APC）	→軟體中斷
0	PASSIVE_LEVEL（被動）	→一般緒程執行

任何時候，每個處理器都執行在某一層級 IRQL。

- 每個處理器此時的 IRQL 決定了它此時如何處理中斷，以及是否允許接收哪些中斷。
  - 同等及較低（<=）的 IRQL 的緒程或中斷會被遮罩。
    如：CPU 的 IRQL 不低於（>=）DISPATCH_LEVEL，不允許存取可以 swap out 的頁面。
  - 較高（>）的 IRQL 的中斷或緒程會被接收/搶佔－中斷它目前所在做的事。
    如：CPU 的 IRQL 不高於（<=）APC_LEVEL 的層次，允許存取可以 swap out 的頁面。
    緒程的切換發生於 CPU 行將從 DISPATCH_LEVEL 級別下降的時候。
    CPU 執行在 DISPATCH_LEVEL 上（的緒程），只有可能被更高層級的中斷搶佔。
- 處理器的 IRQL 應盡可能保持在 PASSIVE_LEVEL，使用者程式碼才有機會得以執行。
- IRQL 的變數在內核中，執行在使用者空間時無法改變 IRQL。
  - 每個處理器的控制資料區 KPCR 結構中有一個 Irql 成員記錄了該處理器目前的 IRQL。

- 核心模式程式碼呼叫 KeGetCurrentIrql 獲得目前處理器的 IRQL。

核心模式中的緒程根據它所要做的事情來提升或者降低目前處理器的 IRQL

- 每當 CPU 進入更底層、更核心的層次存取全域的資料結構或執行關鍵的工作時，提高 IRQL，在完成以後，則降低 IRQL。
  如：與緒程調度相關的資料結構只能在 DISPATCH_LEVEL 上才可以存取。
- 改變 IRQL 透過 KeRaiseIrql 和 KeLowerIrql 函式/KfRaiselrql 和 KfLowerIrql 函式來完成的。（hal.dll）

Windows 程式碼中的 IRQL 定義如下：

```
#define PASSIVE_LEVEL 0                 //Passive release level
#define LOW_LEVEL 0                     //Lowest interrupt level
#define APC_LEVEL 1                     //APC interrupt level
#define DISPATCH_LEVEL 2                //Dispatcher level

#define PROFILE_LEVEL 27                //timer used for profiling.
#define CLOCK1_LEVEL 28             //Interval clock 1 level - Not used on x86
#define CLOCK2_LEVEL 28             //Interval clock 2 level
#define IPI_LEVEL 29                    //Interprocessor interrupt level
#define POWER_LEVEL 30                  //Power failure level
#define HIGH_LEVEL 31                   //Highest interrupt level

#if defined (NT_UP)
#define SYNCH_LEVEL DISPATCH_LEVEL  //針對單一處理器系統的同步層級
#else
#define SYNCH_LEVEL (IPI_LEVEL-2)       //針對多處理器系統的同步層級
#endif
```

Windows 定義的中斷優先層級介紹：

- PASSIVE_LEVEL
  - 最低的 IRQL。
  - 執行在此層級的緒程可以被任何高階 IRQL 的事情打斷。
  - 當 CPU 運行於使用者空間，或者雖然進入了核心但還只是運行於 Executive 層的時候，其運行級別就是 PASSIVE_LEVEL。
  - 所有使用者模式程式碼執行在此 IRQL

- APC_LEVEL
  - 僅僅比 PASSIVE_LEVEL 高
  - 在一個緒程中插入一個 APC 可以打斷該緒程
  - APC 請求相當於對使用者空間程式的(軟體)中斷

- DISPATCH_LEVEL
  - 是最高的軟體插斷 IRQL，它低於所有硬體中斷的 IRQL
  - 大致相當於 CPU 運行於 Windows 內核中的核心層
  - 以下二件事情在此層級執行：
    正在進行緒程調度/切換，緒程（系統）排程器正在執行，選擇新的緒程，正在分配處理器的運算資源
    正在處理一個硬體中斷的後半部分（不那麼緊急的部分），稱爲 DPC（Deferred Procedure Calls）
  - 核心程式碼（中斷處理常式、DPC 程式碼等）執行在這個層級或更高的 IRQL 上
    不得切換到其他緒程
    不能做任何等待動作
    如：不能存取分頁記憶體集區（只能處理非分頁記憶體）、等待一個同步物件
    不能與使用者程式碼或核心中相同或較低 IRQL 程式碼進行同步的操作（中斷處理常式中）

- DIRQL 或裝置 IRQL
  - 3~26 之間的 IRQL 被分配給裝置
  - HAL 規定它們的分配方案
    （Intel x86 多處理器系統）HAL 迴圈地將中斷向量號對應到這段 IRQL 範圍
  - DIRQL 範圈中的 IRQL 並無優先層級區別，不同的裝置中斷只是被對應到相同或不同的 IRQL（APIC 中斷控制器支援 200 多個中斷向量，3~26 無法精確表達各個硬體中斷之間的優先層級和遮罩關係）

- PROFILE_LEVEL
  - 核心執行核心效能剖析功能時
  - 核心的效能剖析功能打開，系統時鐘就會週期性地對被中斷處的程式碼的位址進行採樣，根據建立起的位址採樣表，分析各個核心模組被執行的情況

- CLOCK_LEVEL
  - 系統時鐘中斷使用的 IRQL
  - 核心於此 IRQL 更新系統時間以及觸發計時器

- IPI_LEVEL
  處理器之間通訊的中斷層級

- POWER_LEVEL
  - 電源失敗
  - 未在 Windows 中真正使用

- HIGH_LEVEL
  - 最高層級的工作
  - 禁止所有其他層級的中斷，即遮罩所有其他的中斷
  - 當系統發生不可修復錯誤而進入錯誤檢查（BugCheck）狀態時

中斷物件之中斷處理及擴充機制 - 硬體裝置層級之硬體中斷

中斷物件 KINTERRUPT 結構定義:

```
typedef struct _KINTERRUPT {
    CSHORT Type;
    CSHORT Size;
    LIST_ENTRY InterruptListEntry;
      //同一個中斷向量關聯的中斷物件所構成的一個雙串列
    PKSERVICE_ROUTINE ServiceRoutine;
      //中斷物件的處理常式，即中斷服務常式（ISR）
    PVOID ServiceContext;
    KSPIN_LOCK SpinLock;
    ULONG TickCount;
    PKSPIN_LOCK ActualLock;
    PKINTERRUPT ROUTINE DispatchAddress;
    ULONG Vector;
      //中斷物件的向量編號
    KIRQL Irql;
      //硬體中斷的 IRQL
    KIRQL SynchronizeIrql;
    BOOLEAN FloatingSave;
    BOOLEAN Connected;
    CCHAR Number;
    BOOLEAN ShareVector;
      //是否允許共用中斷（允許多裝置共用一條中斷線）
    KINTERRUPT_MODE Mode;
      //中斷物件的模式，有兩種 LevelSensitive 或 Latched 模式的中斷：
      //LevelSensitive 模式：
      //當插斷要求信號出現（asserted）時，中斷就會發生，中斷物件的常式必須消除中斷
      //Lafuhed 模式：
      //僅當插斷要求信號從無信號（deasserted）到有信號（asserted）發生變化時，
      //中斷才會發生。
    ULONG ServiceCount;
    ULONG DispatchCount;
    ULONG DispatchCode[DISPATCH_LENGTH];
      //中斷發生首先被執行的中斷處理程式碼，爲於中斷物件初始化時填妥的組譯指令
} KINTERRUPT;
```

中斷物件初始化－呼叫 KeInitializeInterrupt 函式（base\ntos\ke\i386\intobj.c 檔案的 82-226 行）

i. 將傳入函式參數的值指定給中斷物件中相關的成員，如 ServiceRoutine、Vector、Irql 等
ii. 將 KiInterruptTemplate 陣列中的指令複製到中斷物件的 DispatchCode 陣列成員

iii. 修正中斷物件結構成員 DispatchCode[]程式碼中的一行指令：
mov edi,0，使得 mov 到 edi 爲指向目前中斷物件 KINTERRUPT 結構的指標（位址）

將初始化的中斷物件掛到中斷物件中透過 Vector 成員指定的中斷向量的 IDT 項中
－呼叫 KeConnectInterrupt 函式（base\ntos\ke\i386\intobj.c 檔案的 228-401 行）
指定的 IDT 項尚未連接任何一個中斷物件

i. 將該中斷物件作爲串列開頭－將中斷物件結構 DispatchCode[]成員起始位址填入指定的 IDT 項（gate）之 32 位元常式偏移
ii. 呼叫 KiConnectVectorAndInterruptObject 函式，修正中斷物件結構 DispatchCode 陣列成員程式碼中的一行指令：
jmp _KeSynchronizeExecution，使得 jmp 的目標位址爲中斷分發函式起始位址，即：
KiInterruptDispatch/KiInterruptDispatch2ndLvl（針對二級分發，由 HAL 決定）或
KiFloatingDispatch/KiFloatingDispatch2ndLv1（針對二級分發）－處理需要儲存浮點數運算狀態的 ISR
iii. 打開中斷開關

指定的 IDT 項已與某一個中斷物件關連

i. 檢查、判斷該中斷物件的設定是否與已連接於指定的 IDT 項的中斷物件的設定相容
包括：SharedVector 成員爲 true、使用相同的中斷模式：LevelSensitive 或 Latched。
ii. 若相容，將該中斷物件插入到既有由相同中斷向量之中斷物件所串連的串列尾部（具相同的中斷向量的中斷物件透過 InterruptListEntry 成員串連）
iii. 呼叫 KiConnectVectorAndInterruptObject 函式，修正串列上所有中斷物件之結構成員 DispatchCode[]程式碼中的一行指令：
jmp _KeSynchronizeExecution，使得 jmp 的目標位址爲鏈式中斷分發函式 KiChainedDispatch/KiChainedDispatch2ndLv1（針對二級分發，由 HAL 決定）起始位址
iv. 打開中斷開關

NOTE
中斷物件連接 IDT 項之原則
每個中斷物件只能被連接到一個 IDT 項
一個服務常式需要被連接到多個處理器的同一個中斷上，必須建立多個中斷物件
多個中斷物件可以被連接到同一個 IDT 項

核心接收到一個中斷時

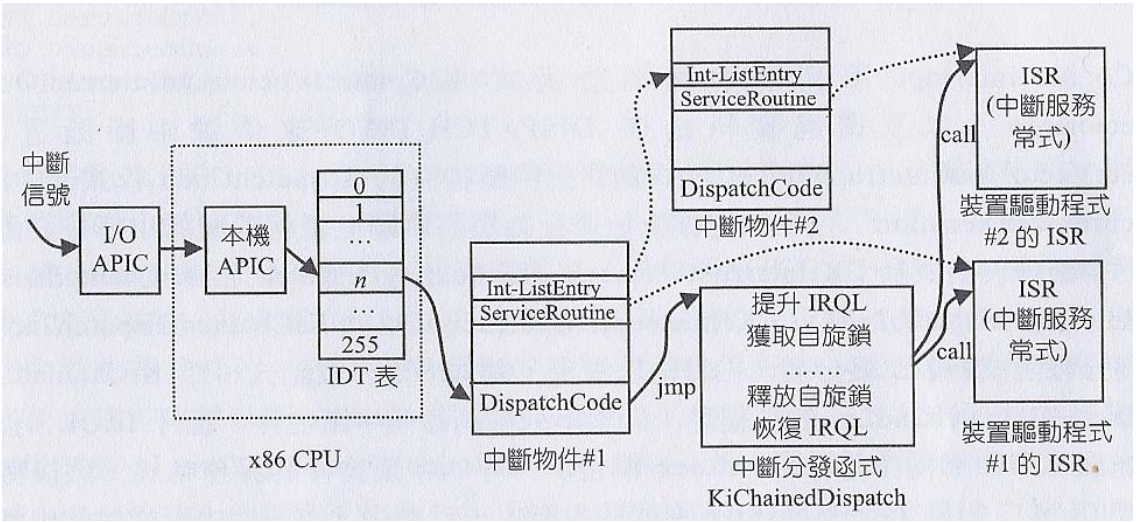
i. 利用中斷向量號作爲索引，找到 IDT 中對應項目
ii. 將該中斷分發到註冊於該 IDT 項的中斷物件的 DispatchCode
iii. 中斷物件的 DispatchCode 程式碼完成下列二件事：
1. 將暫存器 edi 指向中斷物件之 KINTERRUPT 結構
2. 將控制流程轉到該中斷向量/中斷物件對應的（鏈式）中斷分發函式
iv. （鏈式）中斷分發函式碼完成下列幾件事：
1. 提升 IRQL 至 KINTERRUPT 結構的 Irql 成員值

2. 獲得服務常式自旋鎖，即 KINTERRUPT 的 ActualLock 成員
3. 呼叫中斷物件的服務常式（ISR），即 KINTERRUPT 的 ServiceRoutine 成員
4. （自 ISR 返回）
5. 釋放自旋鎖
6. 恢復 IRQL

v. 依據不同的（鏈式）中斷分發函式以及中斷物件的模式，可能反覆上列步驟，處理中斷物件串列（由 InterruptListEntry 成員串連）的其他中斷物件：
   - 若中斷分發函式爲鏈式中斷分發函式且爲 LevelSensitive 模式的中斷
     只要串列上有一個中斷物件的中斷服務常式（ISR）處理了中斷，則後面的中斷物件沒有機會處理該中斷
   - 若中斷分發函式爲鏈式中斷分發函式且爲 Latched 模式的中斷
     鏈式中斷分發函式會遍歷整個串列，串列中所有的中斷物件的 ISR 都有機會被執行。

vi. 中斷分發函式透過呼叫 Kei386EoiHeiper 輔助函式執行 iretd 指令結束整個中斷處理。

下圖爲兩個中斷物件連接到同一個中斷向量情形下的控制流



中斷物件之移除（base\ntos\ke\i386\intobj.c 檔案的 403-560 行）

- 若該物件不是串列中唯一的物件，則從所屬之中斷物件串列中移除
- 若該物件是串列中唯一的物件，則斷開 IDT 項與中斷分發函式的連接（中斷指向中斷分發函式的指標）

裝置驅動程式對中斷物件之操作

中斷物件之初始及與 IDT 項之連接

不直接呼叫 KeInitializeInterrupt 和 KeConnectInterrupt 函式

呼叫 IoConnectInterrupt 函式。此函式封裝以上兩個核心函式的使用過程。

（base\ntos\io\iomgr\iosubs.c 檔案的 3861-4112 行）

中斷物件之移除

不直接呼叫 KeDisconnectInterrupt 函式

呼叫 IoDisconnectInterrupt 函式。此函式封裝了 KeDisconnectInterrupt 函式的使用過程。

中斷物件之 ISR 執行之 IRQL 相關議題

- ISR 執行於硬體裝置 IRQL
- ISR 執行之 IRQL 高於 DISPATCH_LEVEL，即在該常式的執行過程中，不能導致任何緒程環境的切換
- ISR 必須在處理器上一直執行，直至常式結束，除非被更高優先層級的中斷打斷

分派 Dispatch - DPC/dispatch 層級的軟體中斷

kernel 發出 dispatch 軟體中斷，啓動緒程調度（分派）。

緒程調度程式執行之 IRQL 相關議題

- 執行在 DISPATCH_LEVEL
- 低於任何一個硬體中斷的 IRQL，不會遮罩任何硬體中斷
- 高於或等於任何軟體插斷的 IRQL，遮罩了 DPC 程式執行，並可打斷任何緒程的執行

使用時機

- 當 thread 不能繼續執行下去的時候，如：像是 thread 終止、主動進入等待狀態等情形，發出 dispatch 中斷，立刻執行 context switch。
- Kernel 於同等或更高之 IRQL 時偵測到需要重新排程，但需要在完成目前工作之後才執行。
  發出一個 dispatch 軟體中斷（此中斷將暫緩緩處理）
  Kernel 完成目前工作
  IRQL 降到 DPC/dispatch 層級以下
  檢查是否存在中斷等待處理。若存在，將 IRQL 拉到 DPC/dispatch 層級並處理該中斷－引發 thread 調度（分派），。

延遲程式呼叫（DPC, Deferred Procedure Call）- DPC/dispatch 層級的軟體中斷

DPC 透過一種核心控制物件來表達，稱爲 DPC 物件。

DPC 的主要用途

- 裝置驅動程式在其 ISR 中，將相對不緊急的任務（如資料傳輸、後續處理工作）放到 DPC 物件中處理，縮短 CPU 停留在硬體裝置 IRQL 的時間。
- 讓 Kernel 能在 kernel mode（預先）產生 DISPATCH_LEVEL 層級中斷，並於（軟體）中斷被觸發（DPC 被交付）時執行 DPC 延遲函式，完成系統工作。如：透過 DPCs 處理計時器到期時釋放正在等待計時器的 threads 的工作、thread 的 quantum 用完時執行排程及電源失敗恢復等工作。

DPC 物件之 DPC 程式執行之 IRQL 相關議題

- 執行在 DISPATCH_LEVEL
- 低於任何一個硬體中斷的 IRQL，不會遮罩任何硬體中斷
- 高於或等於任何軟體插斷的 IRQL，遮罩了緒程調度，並可打斷任何緒程的執行

DPC 物件的定義：

```
typedef struct _KDPC {
    UCHAR Type;
      //DPC 物件的型別，爲 DpcObject 或 ThreadedDpcObject
      //二者爲列舉型別 KOBJECTS 的值（base\ntos\inc\ke.h）
    UCHAR Importance;
      //DPC 物件的重要程度：
      //   低（LowImportance）、中（MediumImportance）或高（HighImportance）
      //值爲高（HighImportance），DPC 物件被插入到 DPC 串列的開頭，否則插入到尾部
      //預設值：中（MediumImportance）
    UCHAR Number;
      //設定處理 DPC 物件的目標處理器，即它被插入到哪個處理器的 DPC 串列中
      //預設值：0（插入目前處理器的 DPC 串列）
    UCHAR Expedite;
    LIST_ENTRY DpcListEntry;
      //DPC 串列的節點（DPC 串列藉此成員形成串列）
    PKDEFERRED_ROUTINE DeferredRoutine;
      //被延遲執行的函式（延遲函式）指標
      //處理的工作因沒有目前的工作重要，可能不會立刻執行。故稱爲延遲函式。
    PVOID DeferredContext;
      //指向任意資料結構的指標，在 DPC 物件初始化時指定
      //當延遲函式被執行時，被傳遞到延遲函式中
    PVOID SystemArgument1;
      //延遲函式被執行時代入的參數
```

```
    PVOID SystemArgument2;
      //延遲函式被執行時代入的參數
    PVOID DpcData;
      //記錄了它被插入到哪一個 DPC 串列
} KDPC, *PKDPC, *PRKDPC;
```

DPC 的種類及特性

- 普通的（normal）的 DPC：
  可以在任何一個緒程環境中執行
  緒程的（threaded）的 DPC：
  只能在一個專門的 DPC 緒程中執行，需要透過 DPC thread 執行的 DPCs
- DPC 是系統全域的，每個處理器有自己的 DPC 串列，包括一個普通 DPC 串列和一個緒程 DPC 串列
- 每個處理器的 KPRCB 結構之 DpcData 陣列成員的二個元素分別記錄普通 DPC 串列和緒程 DPC 串列

DPC 物件的操作

DPC 物件的初始化－KelnitializeDpc/KilnitializeDpc.函式

DeferredRoutine 和 DeferredContext 成員由傳入之參數指定，其餘成員均付予預設值。

將指定的 DPC 物件插入到 DPC 串列－KeInsertQueueDpc 函式

i. 在插入過程，將 IRQL 提升到最高（HIGH_LEVEL）
ii. 根據 Number 成員的設置，確定處理此 DPC 物件之目標處理器
iii. 找到目標處理器之 DPC 串列，並獲取其自旋鎖
iv. 若該 DPC 物件已存在某個 DPC 串列中，即物件之 DpcData 成員不爲 0，則什麼也不做。否則，依據成員設定值完成插入操作（依 Importance 成員值插入至串列開頭或結尾）
v. 確定是否要立刻發出一個軟體插斷。
   - 當 DPC 物件的 Importance 成員值爲較高，或目標串列中的 DPC 物件數量超過目標處理器設定的最大閾值，或目標處理器的 DPC 請求率小於最低閾值時（參見 dpcobj.c 檔案的 403-423 行），則立刻發出 DISPATCH_LEVEL 的軟體插斷。
   - 當目標處理器並非目前執行之處理器，則立刻發出 IPI_LEVEL 的 IPI（處理器間中斷）請求。
vi. 釋放目標處理器的 DPC 串列的自旋鎖
vii. 恢復 IRQL

從串列中移除指定的 DPC 物件－KeRemoveQueueDpc 函式

i.關閉中斷
ii.獲取目標處理器的 DPC 串列的自旋鎖
iii.從物件所在之 DPC 串列中移除指定的 DPC 物件
iv.釋放目標處理器的 DPC 串列的自旋鎖
v.打開中斷開關。

DPC 物件的交付

DPC 物件在以下三種情況下被交付：（根據 DPC 物件在插入時是否觸發軟體插斷）

- 當處理器的 IRQL 從 DISPATCH_LEVEL 或更高層級降低到 APC_LEVEL 或 PASSIVE LEVEL 時，核心依次呼叫該處理器的 DPC 串列中的 DPC 物件的延遲函式並執行，直至串列爲空。
- 當 KeInsertQueueDpc 插入 DPC 物件時，若依據 DPC 物件的重要程度，以及目標處理器 DPC 串列中的 DPC 數量或請求率，有必要立刻發出一個 DISPATCH_LEVEL 的軟體插斷，則 DPC 串列中的 DPC 物件立即有機會獲得處理。

  NOTE

  以上二者實作由 HAL 呼叫核心中的 KiDispatchInterrupt 函式（base\ntos\ke\i386\ctxswap.asm 檔案，與中斷物件分發函式 KiInterruptDispatch 函式非同一支），再由 KiDispatchInterrupt 呼叫 KiRetireDpcList 函式完成實作，執行流程見下方。

- 每個處理器執行空閒緒程時，若其 DPC 串列有 DPC 物件尚未被執行，則交付這些 DPC 物件。（3.4.5 空閒迴圈的介紹）

  NOTE

  以上二者實作由 KiIdleLoop（見 base\ntos\ke\i386\ctxswap.asm 檔案）呼叫 KiRetireDpcList 函式（base\ntos\ke\dpcsup.c 檔案的 1041-1223 行）完成實作。

DPC 物件交付之實作－KiRetireDpcList 函式

i.透過傳入的參數獲得指向處理器控制區塊 KPRCB 結構的指標

ii.透過 KPRCB 結構的 DpcData 成員取得 DPC 串列

iii.透過 do 迴圈處理計時器和 DPC 物件。處理順序如下：

先處理計時器之 DPC 物件

再透過一個內層 do 迴圈依次交付 DPC 串列中的 DPC 物件

（存取 DPC 串列時，必須獲得 DPC 串列的自旋鎖；從串列中移除一個 DPC 物件後，立即放該鎖）

迴圈反覆執行直至串列爲空。

iv.呼叫 KiProcessDeferredReadyList 函式處理目前處理器的延遲就緒串列，使處於延遲就緒狀態的緒程有機會變成就緒或備用狀態。（參考 3.5.2 節緒程狀態轉移的介紹）

裝置驅動程式對於 DPC 物件之操作

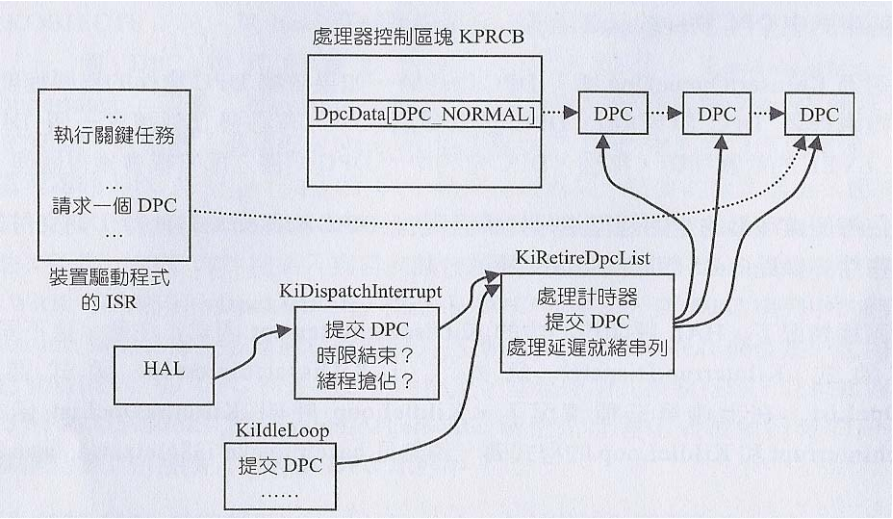
方式1. 透過呼叫 KelnitializeDpc 和 KelnsertQueueDpc 函式

方式2. 透過執行 I/O 管理員的包裝函式 IoInitializeDpcRequest 和 IoRequestDpc 巨集，定義如下（見 base\ntos\inc\io.h）

```
#define IoInitializeDpcRequest( DeviceObject, DpcRoutine ) (\
    KeInitializeDpc( &(DeviceObject)->Dpc,                  \
                     (PKDEFERRED_ROUTINE) (DpcRoutine),     \
                     (DeviceObject) ) )

#define IoRequestDpc( DeviceObject, Irp, Context ) ( \
    KeInsertQueueDpc( &(DeviceObject)->Dpc, (Irp), (Context) ) )
```

下圖爲裝置驅動程式在其中斷服務常式（ISR）中插入一個 DPC 物件，以及 DPC 串列被處理的過程。（實線箭頭代表控制流，虛線箭頭代表了指標關係）



Threaded DPCs

- 最高優先權的 thread 都會被等待執行的 DPC 搶先執行。
- 有些 DPC 的執行時間很長，這正是一般系統或工作站使用者感到系統回應遲鈍的主因，如：讓使用者覺得影片或聲音跳格，甚至覺得鍵盤或滑鼠卡住。
- Windows 爲需要長時間執行 DPC 的驅動程式提供 threaded DPCs。

Threaded DPC 與專門執行 Threaded DPCs 的 DPC 緒程（以下簡稱 DPC 緒程）

- DPC 緒程之啓動函式爲 KiExecuteDpc（見 basc\ntos\ke\dpcsup.c 檔案的 47-218 行）
- DPC 緒程之建立：
  於系統於階段 1 初始化時，透過 Phase1InitializationDiscard 呼叫 KeInitSystem 函式爲每個處理器建立一個 DPC 緒程。
- DPC 緒程於緒程環境中執行 threaded DPC 程式且執行時之 IRQL 爲 passive_level 的 real-time priority（priority 31）。
- 當某緒程執行之時限結束，DPC 緒程中的等待事件會接到信號，從而觸發之前插入之 Threaded DPC 物件被執行（被交付）。
- threaded DPC 程式之執行優先權－
  高於大多數 user-mode threads（大多做應用程式的 threads 不會在 real-time priority 執行）
  低於其他中斷、normal DPC（不需要透過 DPC thread 執行的 DPCs）、APCs 與高優先權的 threads。

非同步程式呼叫（APC, Asynchronous Procedure Call）- APC 層級的軟體中斷

APC 透過一種核心控制物件來表達，稱爲 APC 物件

APC 的主要用途

- APC 物件直接與緒程相關，且當一個緒程獲得控制權時，其 APC 程式立刻被執行。
- APC 非常適合於實作各種非同步通知事件，即向特定的緒程發送非同步通知。
  如：I/O 管理員利用 APC 發送 I/O 讀寫操作的完成通知（支援非同步檔案 I/O 的 Windows API，如 ReadFileEx 和 WriteFileEx 使用了使用者模式 APC）。

APC 物件之 APC 程式執行之 IRQL 相關議題

- 執行在 APC_LEVEL
- 不會遮罩緒程調度，及任何硬體中斷
- 高於 PASSIVE_LEVEL，執行優先於緒程程式碼。可打斷任何緒程的執行

APC 物件的定義：

```
typedef struct _KAPC {
    UCHAR Type;
      //值爲 KOBJECTS 列舉型別的 ApcObject
    UCHAR SpareByte0;
    UCHAR Size;
      //KAPC 結構的大小
    UCHAR SpareBytel;
    ULONG SpareLong0;
    struct _KTHREAD *Thread;
      //指向此 APC 物件所在的緒程 KTHREAD 物件
    LIST_ENTRY ApcListEntry;
      //APC 串列的節點（APC 串列藉此成員形成串列）
    PKKERNEL_ROUTINE KernelRoutine;
      //函式指標（必填），該函式將在核心模式的 APC_LEVEL 上被執行
    PKRUNDOWN_ROUTINE RundownRoutine;
      //函式指標（可選），當緒程終止時若其 APC 串列還有 APC 物件，
      //且其 RundownRoutine 成員非空，則呼叫此成員所指的函式
    PKNORMAL_ROUTINE NormalRoutine;
      //函式指標（可選），指向一個在 PASSIVE_LEVEL 上執行的函式
      //使用者模式 APC 的此常式爲在使用者模式下執行的函式（位於使用者位址空間）
    PVOID NormalContext;
      //與 NormalRoutine 相關成員
    PVOID SystemArgumentl;
      //提供 KernelRoutine 或 NormalRoutine 函式的參數
    PVOID SystemArgument2;
```

```
    //提供 KernelRoutine 或 NormalRoutine 函式的參數
  CCHAR ApcStateIndex;
    //KTHREAD 結構成員 ApcStatePointer 陣列之索引值
    //值是 KAPC_ENVIRONMENT 列舉型別的成員
    //當 APC 物件被插入到緒程的 APC 串列中，
    //   透過此成員可確認（追尋）到其所位於緒程物件的哪個 APC 串列中
  KPROCESSOR_MODE ApcMode;
    //分爲 KernelMode 與 UserMode，爲普通核心模式 APC 與使用者模式 APC 之區隔
  BOOLEAN Inserted;
    //指明該 APC 物件是否已被插入到緒程的 APC 串列
} KAPC, *PKAPC, *PRKAPC;
```

APC 分類及特性

- 核心模式 APC，又依物件結構 NormalRoutine 成員是否爲空分爲：

  特殊 APC

  NormalRoutine 成員爲 NULL 的 APC 物件

  普通 APC

  NormalRoutine 成員不爲 NULL 且 ApcMode 成員爲 KernelMode 的 APC 物件
- 使用者模式 APC

  NormalRoutine 成員不爲 NULL，且 ApcMode 成員爲 UserMode 的 APC 物件
- 每個 APC 是在特定的緒程環境中執行的，亦即一定在特定的行程環境中執行。
- 每個緒程有自己的 APC 串列，同一個緒程的 APC 是排隊執行的

NOTE

KAPC 物件中並沒有一個旗標成員說明它是哪種型別的 APC，分類是依實作方式區分。

特殊 APC 與普通 APC 執行原則：

特殊 APC 和普通 APC 共用同一個 APC 串列

特殊 APC 位於 APC 串列前半部，普通 APC 位於 APC 串列後半部

特殊 APC 可以搶佔普通 APC 的執行

普通 APC 在被交付時，執行 KernelRoutine 常式與 NormalRoutine 常式

（若 KernelRoutine 常式執行過程將其輸出參數 NormalRoutine 清爲 NULL，則 NormalRoutine 常式將不執行）

執行 APC 物件之緒程所屬行程環境之切換

與 APC 操作相關的緒程的 KTHREAD 結構成員

- ApcState 結構：

  KAPC_STATE 型別，記錄與目前緒程所屬行程（附加的對象行程）環境相關的資訊，爲目前緒程操作/記錄/執行 APC 之對象，如：在此環境中執行 APC 物件、插入的 APC 物件進入此結構的串列中、操作 APC 串列爲此結構的串列。
- SavedApcState 結構：

KAPC_STATE 型別，若緒程被附加到另一個行程（發生 attach），則此結構記錄與原本（附加前的）緒程所屬行程環境相關的資訊

- ApcStatePointer[2]：
  指標陣列，指向 KAPC_STATE 型別結構
- ApcStateIndex：
  ApcStatePointer 陣列之索引值，指明指向 ApcState 結構是那一個 ApcStatePointer 陣列元素值是 KAPC_ENVIRONMENT 列舉型別的成員

KAPC_ENVIRONMENT 列舉型別的定義：

```
typedef enum _KAPC_ENVIRONMENT {
   OriginalApcEnvironment,
      //對緒程而言表示原始的行程環境
      //對 APC 而言表示它所關聯的行程環境
   AttachedApcEnvironment,
      //對緒程而言表示附加以後的新行程環境
      //對 APC 而言表示它所關聯的行程環境
   CurrentApcEnvironment,
      //表示在 APC 物件初始化時緒程所屬的行程環境
   InsertApcEnvironment
      //表示在 APC 物件被插入時緒程所屬的行程環境
} KAPC_ENVIRONMENT;
```

緒程在它自己原本所屬行程中執行
  ApcStatePointer[0]指向 ApcState 結構
  ApcStateIndex 等於 0

當緒程被附加到另一個行程（attach）
  尚未交付的 APC 物件從 ApcState 結構轉移到 SavedApcState 結構
  設定 ApcStatePointer[0]指向 SavedApcState 結構
  設定 ApcStatePointer[1]指向 ApcState 結構
  設定 ApcStateIndex 等於 1
  完成 attach 動作，緒程從此在附加的對象行程環境中執行

當緒程欲回到它自己原本所屬行程中執行（detach）－KeDetachProcess 函式
  交付屬於目前（被附加到的）行程的 APC 物件，即 ApcState 中的 APC 物件
  將 SavedApcState 結構中的 APC 轉移回到 ApcState 結構中
  設定 ApcStatePointer[0]指向 ApcState 結構
  設定 ApcStatePointer[1]指向 SavedApcState 結構
  設定 ApcStateIndex 等於 0
  完成 detch 動作，緒程回到它自己原本所屬行程環境中執行

KAPC_STATE 結構的定義：

```
typedef struct _KAPC_STATE {
   LIST_ENTRY ApcListHead[MaximumMode]；  /* MaximumMode=2*/
      //陣列兩個元素分別代表核心模式 APC 物件串列和使用者模式 APC 物件串列
   struct _KPROCESS *Process;
      //指向一個行程物件，代表了這些 APC 所關聯的行程
      //此爲 PsGetCurrentProcess 要透過 ApcState 來獲得目前行程的原因 (3.4.2)
   BOOLEAN KernelApcInProgress;
      //表示該緒程目前正在處理一個普通的核心 APC 物件
   BOOLEAN KernelApcPending;
      //有核心模式 APC 物件正在等待交付
   BOOLEAN UserApcPending;
      //有使用者模式 APC 物件正在等待交付
} KAPC_STATE, *PKAPC_STATE, *PRKAPC_STATE;
```

APC 物件的操作

初始化 APC 物件－透過 KeInitializeApc 函式實作

（base\ntos\ke\apcobj.c 檔案的 29-126 行）

若參數中指定 APC 緒程之行程環境爲 CurrentApcEnvironment，

則 APC 物件的 ApcStateIndex←目標緒程的 ApcStateIndex（使用目標緒程目前的行程環境）；

否則保留 APC 環境參數的值。

初始 APC 物件的緒程物件

初始三個函式指標

將 APC 物件插入到其已指定的目標緒程－KeInsertQueueApc/KiInsertQueueApc 函式

（apcobj.c 檔案的 227-305 行－KeInsertQueueApc 函式）

（base\ntos\ke\apcsup.c 檔案的 380-609 行－KiInsertQueueApc 函式）

i. 將 IRQL 提升至 SYNCH_LEVEL（多處理器系統中，此 IRQL 值爲 IPI_LEVEL-2，即 27）
ii. 獲得目標緒程的 APC 佇列鎖
iii. 若目標緒程的 ApcQueueable 旗標爲 FALSE（當緒程結束的時候），則該緒程不接受 APC，否則執行下一步驟
iv. 呼叫 KiInsertQueueApc 函式（以下步驟由此函式完成）
v. 根據 APC 環境設置和 ApcStateIndex 成員，確定 APC 要插入的目標緒程的 APC 串列
vi. 依據 APC 類型決定 APC 插入 APC 串列中之位置
    使用者模式 APC 插入在 APC 串列首
    普通核心模式 APC 插入 APC 串列尾
    特殊核心模式 APC 則從 APC 串列尾部開始往前搜尋，直到找到第一個 NormalRoutine 爲 NULL 的 APC 物件，將該 APC 物件插入其後。
vii. 判斷 APC 物件欲插入之緒程的行程環境是否符合以下所描述之情況，若符合則執行對應動作：
    插入的是核心模式 APC 物件

- ◆ 若目標緒程爲目前緒程，且緒程未禁止特殊 APC（SpecialApcDisable 旗標爲 0），則呼叫 KiRequestSoftwareInterrupt 函式，發出 APC_LEVEL 的軟體插斷，目前緒程的 APC 被交付。
- ◆ 若目標緒程爲閘道等待狀態的緒程，則呼叫 KiInsertDeferredReadyList 函式使它進入延遲就緒狀態，從而被調度到就緒和執行狀態。
- ◆ 若目標緒程爲等待狀態的緒程，則呼叫 KiUnwaitThread 函式解除該緒程的等待，KiUnwaitThread 會呼叫 KiReadyThread 函式來喚醒該緒程（3.5.2），緒程醒來後就會交付其 APC 物件。

插入的是使用者模式 APC 物件

若目標緒程是可警醒（alertable）的，呼叫 KiUnwaitThread 函式使它解除等待。

viii. 釋放取得的自旋鎖

ix. 恢復 IRQL

APC 交付觸發時機

- ➢ 當核心程式碼離開一個關鍵區段或守護區（呼叫 KeLeaveGuardedRegion 或 KeLeaveCritiealRegion）時，透過 KiCheckForKernelApcDelivery 函式直接呼叫軟體插斷處理常式 KiDeliverApc 函式，或呼叫 KiRequestSoftwareInterrupt 函式發出一個 APC_LEVEL 的軟體插斷，及時地交付核心模式 APC。
- ➢ 當緒程經過環境切換而取得 CPU 執行權時，若它存在核心模式 APC 需要被交付，則在 KiSwapThread 函式傳回以前，呼叫軟體插斷處理常式 KiDeliverApc 函式交付核心模式 APC。
- ➢ 當系統服務或例外處理函式傳回使用者模式時，軟體插斷處理常式 KiDeliverApc 函式被呼叫交付使用者模式 APC。
- ➢ 在 APC_LEVEL 軟體插斷發生時，HAL 模組中的軟體插斷處理函式 HalpDispatehSoftwareInterrupt 呼叫 KiDeliverApc 函式交付核心模式 APC。
- ➢ 當核心程式碼呼叫 KeLowerIrql 函式降低 IRQL 到 PASSIVE_LEVEL 時，軟體插斷處理常式 KiDeliverApc 函式被被呼叫，交付 APC。。

同步

同步保護機制之需求

由於多處理器、多核或者中斷等各種並行性（concurrency）因素的存在，同樣的程式碼有可能被並行執行，資料可能被並行存取。

同步保護機制目標

對於可能被並行存取（共用）的資料進行必要的同步(synchronization)保護，使核心和使用者程式能夠正確地執行，避免因並行存取而造成資料破壞或發生邏輯錯誤。

Windows 作業系統提供的同步保護機制分類

依執行環境中 IRQL 值是否爲 PASSIVE_LEVEL，將同步機制分爲下二種：

- 不依賴於緒程排程的同步機制－
  當 IRQL 大於等於 DISPATCH_LEVEL 時，Windows 提供的同步保護機制，供核心自身或裝置驅動程式使用（此時不適合或不能利用緒程切換的能力來同步資料存取）
- 基於緒程排程的同步機制
  緒程之間等待方式的同步機制

不依賴緒程調度的同步機制

- 提升 IRQL 實作資料同步
  - KPCR 資料結構的 Irql 成員，表示該處理器目前的中斷要求層級。
  - 當處理器在某個 IRQL 上執行時，它只能被更高層級的中斷打斷，低 IRQL 的程式無法搶佔目前的執行程式。
  - 在單一處理器系統上：
    當核心程式碼要存取一個共用資源時，將 IRQL 提升到任何有可能存取該資源的中斷源的最高 IRQL 或以上，即可遮罩掉所有可能的存取衝突。
  - 在多處理器系統上：
    因其他的處理器可能也位於較高的 IRQL，可能並行存取資源。因此採用此機制，需使用其他互斥存取機制配合，如互鎖操作或自旋鎖。

- 互鎖操作
  - 利用 Intel x86 處理器提供的 lock 指令前置，Windows 核心實作了一組輔助函式，如下表所列，以不可分割（原子）方式進行整數運算，實作對整數進行操作時的保護。
  - 每個函式中，當對一個整數記憶體單元執行運算時，從讀取該記憶體單元，到完成運算並把結果寫回記憶體單元，這整個過程不會被其他的處理器打斷，換言之，其他處理器看不到這一操作的中間結果。
  - 可以在任何 IRQL 上被呼叫。
  - 核心和驅動程式大量使用了這樣的函式來保護整數運算。
  - 只能在小細微性的資料上進行同步保護。
  - 屬於指令級的保護。

- Windows 核心支援的整數原子操作

函式	說明
InterlockedIncrement	對一個 LONG 變數加一，使用 lock xadd 指令
InterlockedDecrement	對一個 LONG 變數減一，使用 lock xadd 指令(加-1)
InterlockedExchange	將一個值交換給一個變數，傳回變數原值，使用 xchg 指令，雖然不需要 lock 前置，但預設爲原子操作
InterlockedExchangeAdd	將一個值加到一個變數中，傳回變數原值，使用 lock xadd 指令
InterlockedCompareExchange	先做比較，若相等則設定值，使用 lock cmpxchg 指令
InterlockedOr	對一個 LONG 變數做邏輯或運算，使用 lock or 指令
InterlockedAnd	對一個 LONG 變數做邏輯與運算，使用 lock and 指令
InterlockedXor	對一個 LONG 變數做邏輯互斥運算，使用 lock xor 指令

➢ 無鎖的單串列實作
- 對於簡單的資料結構，譬如其狀態不超過 8 位元組，使用 Intel x86 處理器本身提供的原子指令，實作無鎖操作。
- Windows 利用 64 位元互鎖指令來實作無鎖的單串列資料結構，
- 無鎖的單串列資料結構使得多個處理器以極其高效率的方式共用一個串列。
- Windows 的記憶體管理員和 I/O 管理員使用頻繁。

➢ 自旋鎖
- 處理器爲了存取全域共用資源，不停地檢查用以保護該資源的鎖的狀態。
  獲得其自旋鎖以前，一直在檢查鎖的狀態，無法選擇一個就緒緒程來執行，也不無法交付任何 APC。
  一旦其狀態變成可用，則立即獲取該鎖，從而繼續後面的執行流程。
- 使用方式：先獲得鎖，再存取資料，存取完畢後應立即釋放鎖。
- 僅適用預期等待時間很短（小於兩次環境切換），寧可空轉而不進行環境切換的情況。
- 若一個處理器長時間地抓住一把鎖不放，則有可能導致其他處理器空轉很長時間。
- 自旋鎖在 DISPATCH_LEVEL 或更高的 IRQL 上使用。
  妨礙了緒程調度和 APC 的交付
  不能引發分頁錯誤，也不能調度緒程，否則會導致系統當機（藍底白字畫面）
- 使用者程式不能使用自旋鎖。
- 核心和裝置驅動程式使用自旋鎖來保護共用的資料結構。
  如：每個行程有一個行程自旋鎖，每個緒程有一個緒程自旋鎖。

基於緒程調度的同步機制

緒程進入等待

- 當緒程執行到導致等待的程式碼時，緒程排程器將其 CPU 執行權交出。
- 此段等待執行條件被滿足的期間，緒程物件的一個等待區塊串列，記錄了該緒程正在等待的物件。
- 隨著這些物件的狀態發生變化，該緒程的執行條件可能被滿足。
- 串列節點上的等待的物件是無信號狀態，則該緒程始終處於等候狀態。
- 當緒程的等待條件滿足，系統將該緒程狀能轉爲延遲的就緒狀態。
- 其他的緒程可以透過改變這些物件的狀態來控制等待中的緒程可執行與否。
- 被等待的物件可以用來協調緒程之間的行爲，稱爲同步物件，或發送器物件。
- 在 Windows 核心中，凡是物件標頭以 DISPATCH_HEADER 開頭的物件都是發送器物件。

DISPATCHER_HEADER 結構定義如下：（base\ntos\inc\ntosdef.h 檔案）

```
typedef struct _DISPATCHER_HEADER {
   union {
         struct {
         UCHAR Type;
            //物件的型別
         union {
            UCHAR Absolute;
            UCHAR Npxlrql;
         };
         union {
            UCHAR Size;
            UCHAR Hand;
         };
         union {
            UCHAR Inserted;
            BOOLEAN DebugActive;
         };
       };
       volatile LONG Lock;
          //一個鎖 Lock
   };
   LONG SignalState;
      //表示該發送器物件的信號狀態
   LIST_ENTRY WaitListHead;
      //爲一個串列開頭，此串列包含了所有正在等待該發送器物件的緒程
} DISPATCHER_HEADER;
```

緒程和發送器物件之間的關係

- 緒程物件的 KTHREAD 結構的 WaitBlockList 成員指向一個串列，串列上的每個節點代表了

該緒程正在等待的發送器物件。

- 發送器物件 DISPATCHER HEADER 結構的 WaitListHead 成員指向一個串列，串列上的每個節點代表了正在等待該物件的緒程。
- 以上兩個串列上的節點都是資料型別爲 KWIAIT_BLOCK 結構的等待區塊（wait block）物件。
- 串列節點上的發送器物件是無信號狀態，則該緒程始終處於等候狀態。

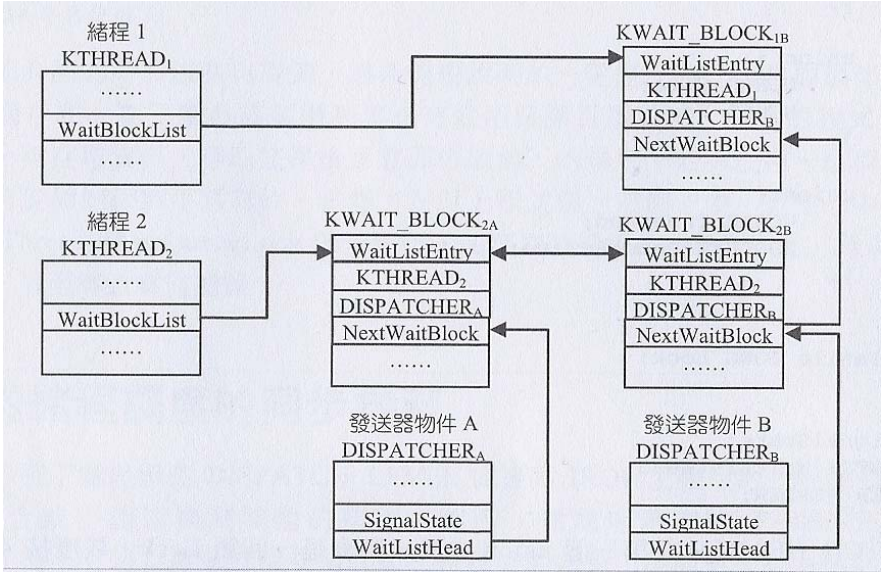
等待區塊（wait block）物件

KWIAIT_BLOCK 結構定義如下：

```
typedef struct _KWAIT_BLOCK {
   LIST_ENTRY WaitListEntry;
   struct _KTHREAD *Thread;
   PVOID Object;
   struct KWAIT_BLOCK *NextWaitBlock;
   USHORT WaitKey;
      //當 Thread 等待物件成功從而被解除等待時的完成狀態值
   UCHAR WaitType;
      //WAIT_TYPE 列舉型別，說明緒程 Thread 等待被滿足的條件
      //   WaitAll 表需要等待串列上所有物件都變成有信號狀態等待條件才被滿足。
      //   WaitAny 表只需要串列上任何一個物件變成有信號狀態等待條件即被滿足。
   UCHAR SpareByte;
} KWAIT_BLOCK, *PKWAIT_BLOCK, *PRKWAIT_BLOCK;
```

- KWAIT_BLOCK 物件描述了哪個緒程（Thread 成員）在等待哪個發送器物件（物件成員）。
- 等待區塊物件同時加入到兩個串列中：
  透過 WaitListEntry 成員加入緒程物件 KTHREAD 結構的 WaitBlockList 成員指向的串列。
  透過 NextWaitBlock 成員加入發送器物件 DISPATCHER_HEADER 結構的 WaitListHead 成員指向一個串列。

緒程物件、發送器物件和等待區塊物件三者之間的關係見下圖：



案例：

緒程物件 1 正在等待發送器物件 B

緒程物件 2 正在等待發送器物件 A 和 B

當發送器物件 B 變成有信號狀態時，緒程物件 1 的等待條件已滿足，它變成延遲的就緒狀態

緒程物件 2 的等待條件是否滿足，要取決於它的等待類型：

若是 WaitType 值爲 WaitAny，則緒程物件 2 的等待條件滿足；

若是 WaitType 值爲 WaitAll，則緒程物件 2 繼續等待。

從發送器物件 B 的角度來看，當它變成了有信號狀態，它喚醒緒程的數量爲一個或全部：

（喚醒緒程的數量依發送器物件的功能類別而有不同）

用於同步的發送器物件，通常只喚醒串列中的一個等待中的緒程

用於通知的發送器物件，通常喚醒串列中所有的等待中的緒程。

Windows 中的發送器物件及其 Signaled 狀態設定與狀態改變之影響

發送器物件的型別由 DISPATCHER_HEADER 結構 Type 成員決定

Windows Server 2003 共支援 8 種發送器物件

事件物件和計時器物件分別有兩種類型：同步或通知

Windows Server 2003 共有以下 10 種核心物件可用於緒程等待機制：

發送器物件類型	KOBJECTS 的型別定義	設爲 Signaled 狀態的時機	對等待的 threads 的影響
Process	ProcessObject	最後一個 thread 結束	全部釋放
Thread	ThreadObject	Thread 結束	全部釋放
Event(Notification)	EventNotificationObject	Thread 設定事件	全部釋放
Event(Synchronization)	EventSynchronizationObject	Thread 設定事件	釋放一個 thread；event 物件重設
Gate	GateObject	Thread 傳訊給 gate	釋放第一個等待的 thread
Semaphore	SemaphoreObject	Semaphore 計數器降 1	釋放一個 thread
Timer (Notification)	TimerNotificationObject	抵達設定時間，或過完等待時間。	全部釋放

Timer (Synchronization)	Synchronization	抵達設定時間，或過完等待時間。	釋放一個 thread
Mutex	MutantObject	Thread 釋放 mutex	釋放一個 thread
Queue	QueueObject	有物件放上 queue	釋放一個 thread