

Pilotes filtres (non PnP)

- 16.1. La pile du clavier
- 16.2. Le blocage de spin
- 16.3. La procédure DriverEntry
- 16.4. La procédure DriverDispatch
 - 16.5.1. La procédure CDO_DispatchCreate
 - 16.5.2. La procédure CDO_DispatchClose
 - 16.5.3. La procédure CDO_DispatchDeviceControl
- 16.6. La procédure KeyboardAttach
- 16.7. La procédure KeyboardDetach
 - 16.8.1. La procédure FiDO_DispatchPower
 - 16.8.2. La procédure FiDO_DispatchPassThrough
 - 16.8.3. La procédure FiDO_DispatchRead
- 16.9. La procédure ReadComplete
- 16.10. Le programme de contrôle

Source code: KmdKit\examples\advanced\KbdSpy\ et KmdKit\examples\advanced\MousSpy

Cet article est le complément pratique du précédent, où nous avons examiné le cycle de vie des IRP. Sans lecture de l'article précédent, vous trouverez ce cours difficile à comprendre. Si vous utilisez une souris ou un clavier USB, malheureusement, vous verrez sans doute surgir quelques difficultés (voir la fin de l'article).

16.1. La pile du clavier

Pour commencer, un tout petit peu de théorie sur la façon dont la pile du clavier fonctionne.

Le microcontrôleur de clavier Intel 8042 (ou un contrôleur compatible avec lui) réalise le raccordement physique du clavier avec le bus. Sur les ordinateurs récents, il est intégré dans le chipset de la carte-mère. Ce contrôleur peut fonctionner dans deux modes : compatible AT et compatible PS/2. Il est difficile, maintenant, de trouver des claviers AT. Tous les claviers sont déjà depuis bien longtemps compatibles PS/2 ou bien reliés à une interface USB. En mode compatible PS/2, le microcontrôleur du clavier se relie également au bus et à la souris compatible PS/2. Ce montage dans son ensemble est régi par le pilote fonctionnel i8042prt (Intel Port Driver 8042), dont le code source complet, peut être trouvé dans le DDK (DDK\src\input\pnpi8042). Le pilote i8042prt crée deux objets "périphérique" non nommés et en relie un à la pile du clavier, et l'autre à la pile de la souris. Dans l'article précédent, dans la figure 15-4, vous avez vu que dans une machine avec un système serveur de terminal, il y a plus d'une pile de clavier et plus d'une de souris également (elles sont déterminées par la quantité de sessions terminales). Dans une machine "normale", les piles de clavier et les piles de "souris" apparaissent approximativement ainsi :

```
kd> !drvobj i8042prt
Driver object (818377d0) is for:
  \Driver\i8042prt
Driver Extension List: (id , addr)

Device Object list:
8181a020 8181b020

kd> !devstack 8181a020
!DevObj  !DrvObj          !DevExt  ObjectName
8181ae30  \Driver\Mouclass    8181aee8  PointerClass0
> 8181a020  \Driver\i8042prt    8181a0d8
81890df0  \Driver\ACPI        8186e008  00000017
!DevNode 8188fe48 :
DeviceInst is "ACPI\PNP0F13\4&2658d0a0&0"
ServiceName is "i8042prt"
```

```

kd> !devstack 8181b020
!DevObj !DrvObj !DevExt ObjectName
8181be30 \Driver\Kbdclass 8181bee8 KeyboardClass0
> 8181b020 \Driver\i8042prt 8181b0d8
81890f10 \Driver\ACPI 8189d228 00000016
!DevNode 8188ff28 :
DeviceInst is "ACPI\PNP0303\4&2658d0a0&0"
ServiceName is "i8042prt"

```

Au-dessus du pilote i8042prt, ou pour être plus précis au-dessus de ses périphériques, sont localisés les objets "périphérique" nommés des pilotes Kbdclass et Mouclass. Le nom "KeyboardClass" sert de base, et on lui ajoute les index supplémentaires (0, 1 et ainsi de suite). Le nom de base est stocké dans la sous-clé de registre HKLM\SYSTEM\CurrentControlSet\Services\Kbdclass\Parameters\KeyboardDeviceBaseName et peut être également "KeyboardPort" si le clavier utilise des pilotes de compatibilité (legacy), cependant je n'ai pas examiné ceci en détail (voir le code source du pilote Kbdclass). Nous utiliserons comme périphérique aux fins de connexion du filtre, un objet "périphérique" du nom de "KeyboardClass0".

Les pilotes Kbdclass et Mouclass sont ainsi nommés car ce sont des pilotes de classe (class drivers) et réalisent le fonctionnement général pour tous les types de claviers et de souris, c'est-à-dire, pour la classe entière de ces périphériques. Ces deux pilotes sont établis comme pilotes filtres de haut niveau et leur code source complet peut également être trouvé dans le DDK (DDK\src\input\kbdclass et DDK\src\input\mouclass, respectivement). Dans les archives de cet article, dans le dossier SetKeyboardLeds est situé le pilote le plus simple, qui allume chacun des trois voyants du clavier. Le pilote fonctionnel i8042prt gère le périphérique "clavier". approximativement ainsi (c'est-à-dire, par les ports d'entrée-sortie). Dans le code source des pilotes Kbdclass et Mouclass, vous ne trouverez certainement pas d'accès à des ports.

La pile du clavier traite plusieurs types de demandes (pour la liste complète voir dans la section du DDK "Kbdclass Major I/O Requests"). Nous intéresseront seulement les IRP du type IRP_MJ_READ, qui fournissent les codes des touches. Le générateur de ces IRP est le thread d'entrées brutes **RawInputThread** du processus système csrss.exe. Ce thread ouvre l'objet "périphérique" du pilote de classe du clavier en utilisation exclusive et à l'aide de la fonction **ZwReadFile**, il dirige vers celui-ci les IRP de type IRP_MJ_READ. Après avoir obtenu l'IRP, le pilote Kbdclass, en utilisant la macro **IoMarkIrpPending**, le signale en attente de finalisation (pending), le place dans la queue et renvoie STATUS_PENDING. Le thread d'entrées brutes doit attendre la finalisation de l'IRP (pour être plus précis, **RawInputThread** obtient les événements clavier en faisant un appel à une procédure asynchrone (Asynchronous Procedure Call, APC)). En se connectant à la pile, le pilote Kbdclass enregistre dans le pilote i8042prt la procédure callback **KeyboardClassServiceCallback**, en lui transmettant IOCTL_INTERNAL_KEYBOARD_CONNECT comme IRP. Le pilote i8042prt enregistre également dans le système sa routine de service d'interruption (interrupt service routine, ISR) **I8042KeyboardInterruptService**, par l'appel à la fonction **IoConnectInterrupt**. Quand une touche est appuyée ou relâchée, le contrôleur du clavier générera une interruption matérielle. Son processeur appellera **I8042KeyboardInterruptService**, qui lira dans la queue interne du contrôleur de clavier les données nécessaires. Puisque le traitement des interruptions matérielles s'effectue à un IRQL élevé, l'ISR fait seulement le travail le plus urgent et place dans la queue un appel à une procédure différée (Deferred Procedure Call, DPC). Les DPC s'effectuent avec IRQL = DISPATCH_LEVEL. Quand l'IRQL est ramené à DISPATCH_LEVEL, le système appellera la procédure **I8042KeyboardIsrDpc**, qui appellera la procédure callback enregistrée par le pilote Kbdclass, **KeyboardClassServiceCallback** (qui s'effectue également à IRQL = DISPATCH_LEVEL). **KeyboardClassServiceCallback** extraira de la queue l'IRP en attente de finalisation, remplira la structure KEYBOARD_INPUT_DATA (en réalité i8042prt essaye de vider toute la queue du contrôleur du clavier, et Kbdclass de son côté essaie de remplir autant de structures KEYBOARD_INPUT_DATA, qu'il en entrera dans le tampon de l'IRP), qui porte toutes les informations nécessaires sur l'enfoncement/relâchement des touches et finalisera l'IRP. Le thread d'entrées brutes se réveille, traite l'information obtenue et envoie de nouveau un IRP de type IRP_MJ_READ au pilote de classe, qui place de nouveau dans la queue jusqu'à l'action suivante sur les touches. Ainsi, la pile du clavier a toujours, au moins, un IRP qui attend sa finalisation situé dans la queue du pilote Kbdclass. La pile de "souris" se comporte pareillement.

```

kd> !devobj 8181be30
Device object (8181be30) is for:
KeyboardClass0 \Driver\Kbdclass DriverObject 818372b0
Current Irp 815a2e68 RefCount 0 Type 0000000b Flags 00002044
DevExt 8181bee8 DevObjExt 8181bfd8
ExtensionFlags (0000000000)
AttachedTo (Lower) 8181b020 \Driver\i8042prt

```

```
Device queue is busy -- Queue empty.
```

```
kd> !irp 815a2e68 1
Irp is active with 6 stacks 6 is current (= 0x815a2f8c)
  No Mdl System buffer = 81664b48 Thread 8171bca0: Irp stack trace.
Flags = 00000970
ThreadListEntry.Flink = 8171beac
ThreadListEntry.Blink = 8171beac
IoStatus.Status = 00000103
IoStatus.Information = 00000000
RequestorMode = 00000000
Cancel = 00
CancelIrql = 0
ApcEnvironment = 00
UserIosb = e28b1028
UserEvent = 00000000
Overlay.AsynchronousParameters.UserApcRoutine = a0063334
Overlay.AsynchronousParameters.UserApcContext = e28b1008
Overlay.AllocationSize = e28b1008 - a0063334
CancelRoutine = f3ec82e0
UserBuffer = e28b1068
&Tail.Overlay.DeviceQueueEntry = 0006da94
Tail.Overlay.Thread = 8171bca0
Tail.Overlay.AuxiliaryBuffer = 00000000
Tail.Overlay.ListEntry.Flink = 00000000
Tail.Overlay.ListEntry.Blink = 00000000
Tail.Overlay.CurrentStackLocation = 815a2f8c
Tail.Overlay.OriginalFileObject = 8171aa08
Tail.Apc = 00000000
Tail.CompletionKey = 00000000
      cmd  flg  cl Device      File      Completion-Context
[  0, 0]   0   0 00000000 00000000 00000000-00000000
      Args: 00000000 00000000 00000000 00000000
[  0, 0]   0   0 00000000 00000000 00000000-00000000
      Args: 00000000 00000000 00000000 00000000
[  0, 0]   0   0 00000000 00000000 00000000-00000000
      Args: 00000000 00000000 00000000 00000000
[  0, 0]   0   0 00000000 00000000 00000000-00000000
      Args: 00000000 00000000 00000000 00000000
[  0, 0]   0   0 00000000 00000000 00000000-00000000
      Args: 00000000 00000000 00000000 00000000
>[  3, 0]   0   1 8181be30 8171aa08 00000000-00000000      pending
      \Driver\Kbdclass
      Args: 00000078 00000000 00000000 00000000
```

Comme on peut le voir, le pilote Kbdclass a un IRP de type IRP_MJ_READ non finalisé (numéro 3 entre crochets). La colonne cl montre le contenu du champ IO_STACK_LOCATION.Control. Dans notre cas, il s'agit du drapeau SL_PENDING_RETURNED, établi par l'appel à la macro **IoMarkIrpPending**. La ligne Args est le contenu de la structure IO_STACK_LOCATION.Read insérée. La taille du tampon (78h) peut héberger exactement 10 structures KEYBOARD_INPUT_DATA. Le tampon lui-même est localisé grâce à l'adresse du tampon System = 81664b48. Vous porterez également votre attention sur la ligne CancelRoutine = f3ec82e0. C'est l'adresse de la procédure d'annulation d'IRP (Cancel Routine), qui appartient au pilote Kbdclass. L'annulation d'IRP - c'est un grand sujet supplémentaire et relativement complexe (que je ne projette pas d'aborder). Plus tard, nous aurons à en parler un peu.

L'IRP qui attend sa finalisation, naturellement, appartient au thread **RawInputThread**.

```

kd> !thread 8171bca0
THREAD 8171bca0 Cid a4.bc Teb: 00000000 Win32Thread: e28ae5a8 WAIT:
(WrUserRequest) KernelMode Alertable
    8171bf20 SynchronizationEvent
    8171bc08 SynchronizationEvent
    8171bbc8 NotificationTimer
    8171bc48 SynchronizationEvent
IRP List:
    815a2e68: (0006,0148) Flags: 00000970 Mdl: 00000000
Not impersonating
Owning Process 81736160
Wait Start TickCount 57881
Context Switch Count 18896
UserTime 0:00:00.0000
KernelTime 0:00:00.0070
Start Address win32k!RawInputThread (0xa00ad1aa)
Stack Init bfd1d000 Current bfd1cafa0 Base bfd1d000 Limit bfd1a000 Call 0
Priority 19 BasePriority 13 PriorityDecrement 0 DecrementCount 0

```

Quand nous installons le filtre, alors l'IRP nous atteint une première fois. Puisque l'IRP de type IRP_MJ_READ est vraiment une requête de lecture de données, alors quand il descend le long de la pile, son tampon est naturellement vide. Le tampon de lecture ne contiendra de données qu'après finalisation de l'IRP. Pour les voir (les données), le filtre doit établir dans chaque IRP (plus précisément dans son bloc de pile) une procédure de finalisation. Puisque le pilote Kbdclass a envoyé l'IRP placé dans la queue, avant que nous installions le filtre, alors qu'il ne comportait pas notre procédure de finalisation, il n'y a aucun moyen pour nous de pouvoir voir le code de la touche qui sera appuyée juste après l'installation du filtre. Quand la touche est appuyée, elle s'attend à ce que la finalisation de l'IRP soit terminée et **RawInputThread** enverra de nouveau un IRP de type IRP_MJ_READ. Cet IRP et tous les suivants, nous les intercepterons et nous appellerons la procédure de finalisation. Quand la touche est relâchée, nous lirons son code lors de la procédure de finalisation. C'est pour cette raison que le filtre ne voit pas le moment de la pression sur la première touche, et, dans le moniteur, vous verrez toujours, pour la première touche appuyée, seulement son code de relâchement (break code). Ensuite, le filtre intercepte toutes les pressions et tous les relâchements de toutes les touches.

Avant de passer au code du pilote, continuons un peu nos digressions.

16.2. Le blocage de spin

Puisque le traitement des IRP est, par nature, asynchrone, souvent "la main droite ne sait pas que fait la gauche". Par conséquent, dans chaque pilote sérieux, il est nécessaire d'employer un mécanisme de synchronisation nommé "accès concurrentiel". Dans la treizième partie du cycle - "Technologie de base. Synchronisation : Accès concurrentiel", nous avons déjà présenté comment mettre en place l'accès concurrentiel à l'aide de mutex. Par conséquent, je ne parlerai pas des macros MUTEX_INIT, MUTEX_ACQUIRE et MUTEX_RELEASE, que nous utiliserons de nouveau.

La synchronisation à l'aide de mutex est pratique, mais, malheureusement, elle a un défaut : il est bien connu, qu'elle ne peut pas s'appliquer à des objets d'IRQL égal ou supérieur à DISPATCH_LEVEL. Pour mettre en place un accès concurrentiel à un IRQL égal à DISPATCH_LEVEL, il existe un mécanisme nommé "blocage de spin" (spin lock).

```

KfAcquireSpinLock proc ; Capture de spin sur HAL monoprocasseur
    xor     eax, eax
    mov     al, ds:0FFDF024h ; al = KPCR.Irql
    mov     byte ptr ds:0FFDF024h, 2 ; KPCR.Irql = DISPATCH_LEVEL
    ret
KfAcquireSpinLock endp

KfAcquireSpinLock proc ; Capture de spin sur HAL multiprocasseur
    mov     edx, ds:0FFFE0080h ; edx = APIC[TASK PRIORITY REGISTER]
    mov     dword ptr ds:0FFFE0080h, 41h
    ; APIC[TASK PRIORITY REGISTER] = DPC VECTOR
    shr     edx, 4
    movzx   eax, ds:HalpVectorToIRQL[edx]; OldIrql

```

```

trytoacquire:
    lock bts dword ptr [ecx], 0 ; Essayons de bloquer atomiquement.
    jb     spin                ; Si le blocage est occupé dans le cycle
    ret                               ; Nous avons bloqué, remettons l'IRQL
                                           ; auquel se trouvait le processeur lors du
                                           ; blocage.

    align 4

spin:                                ; Blocage occupé, nous bouclons.
    test  dword ptr [ecx], 1      ; Vérifions le blocage.
    jz    trytoacquire           ; S'il est libre, essayons de le prendre.
    pause                               ; Sinon, continuons à boucler.
                                           ; L'instruction pause est spécifique
                                           ; elle est prévue pour le blocage de spin.
                                           ; Des détails dans la section "PAUSE-Spin
                                           ; Loop Hint" de IA-32 Intel Architecture
                                           ; Software Developer's Manual
                                           ; Volume 2 : Instruction Set Reference.

    jmp   spin

KfAcquireSpinLock endp

```

Dans une machine monoprocesseur, la saisie du blocage de spin consiste en une simple augmentation de l'IRQL à DISPATCH_LEVEL. Comme on sait, à cet IRQL, la planification des threads n'a pas lieu et le thread qui contrôle le processeur sera exécuté jusqu'à ce que l'IRQL soit abaissé. Puisque l'IRQL est un attribut du processeur, une simple augmentation d'IRQL ne suffit pas avec les multiprocesseurs, puisqu'elle ne bloque pas les threads qui sont exécutés par d'autres processeurs. Par conséquent le blocage de spin est légèrement plus complexe à réaliser dans un HAL multiprocesseur et il s'agit bien d'un blocage de rotation dans le vrai sens du terme (to spin - tourner, lancer une toupie, un tourniquet). C'est-à-dire, si le blocage est occupé, le thread tourne en boucle infinie, essayant de le saisir à chaque tour. Dans ce cas, puisque la boucle s'effectue à l'IRQL = DISPATCH_LEVEL, la planification des threads sur ce processeur ne se produit pas et le processeur ne peut pas effectuer de travail utile. Pour cette raison même, les blocages de spin sont beaucoup plus critiques en temps. C'est-à-dire, il est nécessaire de libérer le blocage de spin aussi rapidement que possible. Le DDK lui-même détermine l'intervalle maximum de temps pendant lequel le thread peut retenir le blocage de spin à 25 micro-secondes. Dans ce sens, les mutex et les autres objets d'attente sont moins exigeants, puisque le thread qui attend l'objet occupé est simplement exclu de la planification, et le processeur obtient d'autres threads, prêts à être exécutés.

Et puisque nous avons dû aborder le blocage de spin, profitez-en pour retenir quelques règles classiques. Premièrement, la saisie d'un blocage de spin, comme nous venons de l'expliquer, augmente l'IRQL à DISPATCH_LEVEL, et cela signifie que nous ne pouvons accéder qu'à la mémoire non-paginable et que le code lui-même doit également se trouver dans la mémoire non-paginable. Deuxièmement, la saisie répétée d'un même blocage de spin, naturellement, amènera à un blocage complet (deadlock). Troisièmement, il est impossible de saisir un blocage de spin à un IRQL plus haut que DISPATCH_LEVEL, puisque ceci entraînera en fait une réduction explicite de l'IRQL qui inévitablement mènera à l'écran bleu. Quatrièmement, s'il est nécessaire de prendre deux (ou plus) blocages de spin, alors tous les threads doivent le faire dans le même ordre. Sinon, le blocage mutuel est possible. Par exemple, deux threads doivent prendre les blocages A et B. S'ils le font dans un ordre différent, alors, il est possible qu'en même temps que le premier thread prend le blocage A, le deuxième prend le blocage B et qu'ensuite tous les deux attendent infiniment : le premier que soit libéré le blocage B, et le second que soit libéré le blocage A.

```

LOCK_ACQUIRE MACRO lck:REQ
    mov ecx, lck
    fastcall KfAcquireSpinLock, ecx
ENDM

LOCK_RELEASE MACRO lck:REQ, NewIrql:REQ

    mov ecx, lck
    mov dl, NewIrql

    .if dl == DISPATCH_LEVEL
        fastcall KefReleaseSpinLockFromDpcLevel, ecx
    .endif

```

```

    .else
        and edx, 0FFh
        fastcall KfReleaseSpinLock, ecx, edx
    .endif

```

```

ENDM

```

Pour saisir le blocage de spin, nous utiliserons des macros. Ce sont des versions simplifiées. Dans le macro `LOCK_RELEASE`, nous utilisons une petite optimisation : si nous devons saisir le blocage de spin à `IRQL = DISPATCH_LEVEL`, alors il est plus avantageux d'appeler `KefReleaseSpinLockFromDpcLevel` au lieu de `KfReleaseSpinLock`, puisqu'il n'est pas nécessaire de changer d'IRQL. Egalement, sur une machine monoprocesseur, `KefReleaseSpinLockFromDpcLevel` est une fonction "vide".

```

KeReleaseSpinLockFromDpcLevel proc
    retn 4
KeReleaseSpinLockFromDpcLevel endp

```

Il est aussi possible de faire quelque chose de similaire (comme optimisation), pour la macro `LOCK_ACQUIRE`. Il suffit de savoir si l'IRQL courant est égal à `DISPATCH_LEVEL`, et dans ce cas appeler `KeAcquireSpinLockAtDpcLevel`, ce qu'effectue également (dans la machine monoprocesseur) l'instruction `ret`.

```

KeAcquireSpinLockAtDpcLevel proc
    retn 4
KeAcquireSpinLockAtDpcLevel endp

```

Je n'ai pas commencé à optimiser la macro `LOCK_ACQUIRE`, car j'ai écrit ces macros il y a bien longtemps et je les ai souvent utilisées sans problème et en outre ce qui est plus rapide n'est pas certain : appeler simplement `KfAcquireSpinLock` ou obtenir l'IRQL et selon sa valeur éventuellement appeler `KeAcquireSpinLockAtDpcLevel`. Par conséquent, je ne me suis pas posé d'autres questions et j'ai tout laissé en l'état. Si vous avez un désir infatigable d'optimisation, étudiez `hal.dll/halmps.dll` et `ntoskrnl.exe/ntkrnlmp.exe` et optimisez tant que vous voulez.

Pour compléter le tableau, j'ajoute encore la fonction `KeAcquireSpinLockRaiseToSynch`, qui augmente l'IRQL pendant la capture du blocage au niveau `CLOCK2_LEVEL` (28).

16.3. La procédure `DriverEntry`

Maintenant examinons notre propre filtre. Comme je l'ai déjà dit, c'est un pilote non-PnP. Il y a pas mal de code aussi, je ne le donnerai pas complètement (voir les sources de l'article).

```

    invoke IoCreateDevice, pDriverObject, 0, addr g_usControlDeviceName, \
        FILE_DEVICE_UNKNOWN, 0, TRUE, addr g_pControlDeviceObject

```

Cette fois, notre pilote gèrera deux objets : l'objet "périphérique filtre" (filter device object) et l'objet "périphérique de commande" (control device object). L'objet "périphérique filtre" sera associé à la pile du clavier, et à travers lui passeront tous les IRP qui gèrent le clavier. Au moyen de l'objet "périphérique de commande" le programme de contrôle enverra au pilote les commandes nécessaires : "connecter le filtre", "ouvrir le filtre", "transmettre les données interceptées". Pour l'instant, nous n'avons besoin que du périphérique de commande. Cet objet sera nommé, de sorte que le programme de contrôle puisse y obtenir l'accès. Nous ne voulons pas travailler simultanément avec plusieurs clients. Par conséquent, créons un objet exclusif, en spécifiant `TRUE` dans le paramètre `Exclusive`. Alors, le gestionnaire d'objets ne permettra de créer qu'une handle d'objet. Malheureusement, cette méthode simple n'est pas très fiable, et il est possible d'ouvrir quand même l'objet via le chemin relatif, c'est-à-dire, après ouverture du dossier `"\Device"` et après transmission de sa handle dans le paramètre `RootDirectory` de la macro `InitializeObjectAttributes`. Le DDK indique d'une manière générale que le paramètre `Exclusive` est réservé. Par conséquent, nous ajouterons un traitement supplémentaire aux requêtes `IRP_MJ_CREATE` et `IRP_MJ_CLOSE`.

```

    invoke ExAllocatePool, NonPagedPool, sizeof NPAGED_LOOKASIDE_LIST
    .if eax != NULL

        mov g_pKeyDataLookaside, eax

        invoke ExInitializeNPagedLookasideList, \
            g_pKeyDataLookaside, NULL, NULL, 0, \
            sizeof KEY_DATA_ENTRY, 'ypSK', 0
    .endif

```

Nous allouons de la mémoire pour une liste liée et nous l'initialisons. Depuis cette liste, nous allouerons de la mémoire pour les copies de la structure `KEY_DATA_ENTRY` que nous définissons.

```
KEY_DATA STRUCT
    dwScanCode  DWORD    ?
    Flags        DWORD    ?
KEY_DATA ENDS
PKEY_DATA typedef ptr KEY_DATA

KEY_DATA_ENTRY STRUCT
    ListEntry    LIST_ENTRY <>
    KeyData      KEY_DATA    <>
KEY_DATA_ENTRY ENDS
```

Les copies de cette structure stockeront les données sur les enfoncements/relâchements de touches interceptés et nous les stockerons (les copies de la structure) dans la liste doublement liée. Dans la septième partie du cycle - "Technologie de base : Travail avec la mémoire. L'utilisation des listes Look-Aside (préallouées)", nous sommes rentrés suffisamment dans le détail des listes préallouées (lookaside list) et des listes doublement liées (doubly linked list). Je suis certain que beaucoup d'entre vous ont simplement sauté cet article ;) S'il en est ainsi, alors il est nécessaire de le lire maintenant, puisque je ne le répéterai pas, et sans cette base, quelque chose peut naturellement vous échapper. La seule différence est que, maintenant, nous utiliserons la liste lookaside en mémoire non-paginable. Les fonctions `ExAllocateFromNPagedLookasideList` et `ExFreeToNPagedLookasideList` pour le traitement des listes lookaside non-paginées sont réalisées dans le DDK en tant que macros, et non en tant que fonctions comme pour les listes lookaside paginées. Malheureusement, en raison des limitations du macrolangage MASM, j'ai dû les réaliser sous la forme des fonctions `_ExAllocateFromNPagedLookasideList` et `_ExFreeToNPagedLookasideList`. Des listes lookaside non-paginées sont exigées, comme vous pouvez l'imaginer, parce que nous travaillerons avec elles à l'IRQL = DISPATCH_LEVEL.

```
InitializeListHead addr g_KeyDataListHead
```

La variable globale `g_KeyDataListHead` est la tête de la liste doublement liée des structures `KEY_DATA_ENTRY`.

```
invoke KeInitializeSpinLock, addr g_KeyDataSpinLock
```

Le blocage de spin est obligatoire pour mettre en place un accès exclusif à la liste des structures `KEY_DATA_ENTRY`. L'usage d'objets de synchronisation, par exemple, des mutex, nous est interdit, puisque nous accéderons aux listes à l'IRQL = DISPATCH_LEVEL.

```
invoke KeInitializeSpinLock, addr g_EventSpinLock
```

Ce blocage de spin nous aidera à mettre en place l'accès exclusif à la variable `g_pEventObject`, dans laquelle sera stocké le pointeur sur l'objet événement. Cet objet sera employé pour informer le programme de contrôle de l'arrivée de nouvelles données (pour plus de détails, voir la partie 14 "Technologie de base. Synchronisation : Utilisation des objets").

```
MUTEX_INIT g_mtxCDO_State
```

A l'aide de ce mutex nous pourrons exécuter exclusivement quelques sections de code.

```
mov ecx, IRP_MJ_MAXIMUM_FUNCTION + 1
    .while ecx
        dec ecx
        mov [eax].MajorFunction[ecx*(sizeof PVOID)], \
            offset DriverDispatch
    .endw
```

Nous remplissons tous les éléments du tableau de pointeurs sur les procédures de traitement du pilote, avec l'adresse de l'unique procédure `DriverDispatch`. Cette procédure distribuera les requêtes entre le filtre et le périphérique de commande. Le périphérique de commande n'obtiendra du programme de contrôle que trois requêtes : `IRP_MJ_CREATE`, `IRP_MJ_CLOSE` et `IRP_MJ_DEVICE_CONTROL`. Mais le périphérique filtre, lui, peut obtenir n'importe quelle requête, puisqu'il est associé à la pile déjà existante, sur laquelle peuvent circuler des IRP de n'importe quel type. Souvent, le spectre entier des IRP qui passent le long de la pile du filtre n'est pas du tout connu. Il est nécessaire de filtrer seulement quelques types d'IRP, mais si le filtre obtient une requête qui ne l'intéresse pas, il est obligé de la rediriger au-dessous dans la pile. Pour cette raison-même, nous devons remplir le tableau `MajorFunction` en entier. Sinon, dans les éléments non remplis, se trouvera le pointeur sur la fonction système

IopInvalidDeviceRequest, qui exécutera l'IRP avec le code STATUS_INVALID_DEVICE_REQUEST, et nous bloquerons le traitement de telles requêtes.

16.4. La procédure DriverDispatch

Dans notre pilote, les requêtes sont dirigées sur deux objets : le périphérique de commande et le filtre (s'il est connecté). Tous les IRP aboutissent dans la procédure générale de traitement **DriverDispatch**.

```
IoGetCurrentIrpStackLocation pIrp

movzx eax, (IO_STACK_LOCATION PTR [eax]).MajorFunction
mov dwMajorFunction, eax

mov eax, pDeviceObject
.if eax == g_pFilterDeviceObject

    mov eax, dwMajorFunction
    .if eax == IRP_MJ_READ
        invoke FiDO_DispatchRead, pDeviceObject, pIrp
        mov status, eax
    .elseif eax == IRP_MJ_POWER
        invoke FiDO_DispatchPower, pDeviceObject, pIrp
        mov status, eax
    .else
        invoke FiDO_DispatchPassthrough, pDeviceObject, pIrp
        mov status, eax
    .endif

.elseif eax == g_pControlDeviceObject

    mov eax, dwMajorFunction
    .if eax == IRP_MJ_CREATE
        invoke CDO_DispatchCreate, pDeviceObject, pIrp
        mov status, eax
    .elseif eax == IRP_MJ_CLOSE
        invoke CDO_DispatchClose, pDeviceObject, pIrp
        mov status, eax
    .elseif eax == IRP_MJ_DEVICE_CONTROL
        invoke CDO_DispatchDeviceControl, pDeviceObject, pIrp
        mov status, eax
    .else

        mov ecx, pIrp
        mov (_IRP PTR [ecx]).IoStatus.Status, \
            STATUS_INVALID_DEVICE_REQUEST
        and (_IRP PTR [ecx]).IoStatus.Information, 0

        fastcall IofCompleteRequest, ecx, IO_NO_INCREMENT

        mov status, STATUS_INVALID_DEVICE_REQUEST

    .endif

.else

    mov ecx, pIrp
    mov (_IRP PTR [ecx]).IoStatus.Status, STATUS_INVALID_DEVICE_REQUEST
    and (_IRP PTR [ecx]).IoStatus.Information, 0

    fastcall IofCompleteRequest, ecx, IO_NO_INCREMENT
```

```

        mov status, STATUS_INVALID_DEVICE_REQUEST

    .endif

    mov eax, status
    ret

```

A l'aide des pointeurs globaux `g_pFilterDeviceObject` et `g_pControlDeviceObject`, nous déterminons à quel objet est arrivé la requête et, selon le type de requête, nous appelons la procédure appropriée. Notre périphérique de commande ne traite que trois types de requêtes : `IRP_MJ_CREATE`, `IRP_MJ_CLOSE` et `IRP_MJ_DEVICE_CONTROL`. Mais nous sommes obligés de traiter toutes les requêtes dans le filtre. Le traitement consistera en un simple transfert de l'IRP au pilote subalterne dans la procédure **FiDO_DispatchPassThrough**. Les requêtes de type `IRP_MJ_READ` contiennent les codes des touches ; donc il y aura un traitement spécial pour ce type de requêtes. Les IRP de type `IRP_MJ_POWER` exigent un traitement spécifique ; donc ils sont isolés dans une procédure séparée. Si nous obtenions tout d'un coup (quoique ce ne soit pas possible), une requête pour des périphériques inconnus de nous, nous la finaliserions avec un code d'erreur, car on ne saurait pas quoi faire avec cet IRP.

Décortiquons d'abord les requêtes de traitement du périphérique de commande.

16.5.1. La procédure `CDO_DispatchCreate`

```

    .while TRUE

        invoke RemoveEntry, addr KeyData
        .break .if eax == 0

    .endw

```

Le pilote et le programme de contrôle sont conçus de telle manière que le programme de contrôle puisse être stoppé et relancé à plusieurs reprises alors que le pilote est déjà chargé et le filtre connecté. Il peut ainsi se produire que la liste `g_KeyDataListHead` ne soit pas vide. Si vous analysez attentivement le cours des événements possibles après avoir lu entièrement l'article, alors il sera clair que dans la liste peut se trouver une structure `KEY_DATA_ENTRY`, qui correspond au code de la touche, appuyée juste après une finalisation incorrecte du travail du programme de contrôle. La boucle ci-dessus vide une liste `g_KeyDataListHead` éventuellement non vide.

```

    MUTEX_ACQUIRE g_mtxCDO_State

    .if g_fCDO_Opened

        mov status, STATUS_DEVICE_BUSY

    .else

        mov g_fCDO_Opened, TRUE

        mov status, STATUS_SUCCESS

    .endif

    MUTEX_RELEASE g_mtxCDO_State

```

Si la handle de l'objet "périphérique de commande" est déjà ouverte, nous ne permettons pas une nouvelle ouverture. Ceci nous garantit la présence d'un seul client (les autres obtiendront le code `STATUS_DEVICE_BUSY`), et la saisie du mutex, elle, garantit que la procédure `CDO_DispatchClose` ne fermera pas en même temps la handle et ne remettra pas à zéro le drapeau `g_fCDO_Opened`.

16.5.2. La procédure `CDO_DispatchClose`

```

    and g_fSpy, FALSE

```

Si le client est déconnecté, alors il est inutile de tracer le clavier - **FiDO_DispatchRead** ne doit pas non plus installer de procédure de finalisation.

```

MUTEX_ACQUIRE g_mtxCDO_State

.if ( g_pFilterDeviceObject == NULL )

    .if g_dwPendingRequests == 0

        mov eax, g_pDriverObject
        mov (DRIVER_OBJECT PTR [eax]).DriverUnload, offset DriverUnload

    .endif

.endif

```

Si la variable `g_pFilterDeviceObject` est vide, alors, il est évident qu'il n'y a pas de filtre. Si, par ailleurs, nous n'avons pas d'IRP non finalisés, dont la finalisation amènerait à appeler notre procédure de finalisation **ReadComplete**, qui est située dans le corps du pilote, alors, il est possible de le laisser décharger. Si le filtre existe toujours, le pilote reste chargé. Avant la finalisation du travail, le programme de contrôle invite le pilote à ouvrir et enlever le filtre. Mais on peut rencontrer des situations où le pilote ne peut pas le faire. Par exemple, si quelqu'un s'est connecté à la pile au-dessus de nous, fermer le filtre "détruit la pile". Le programme de contrôle peut simplement oublier d'ouvrir le filtre ou une exception peut s'y produire et la handle du périphérique est automatiquement fermée par le système. La discussion, cela va de soi, ne concerne pas notre programme de commande, dans lequel (je l'espère !) tout est fait comme il faut. Je parle d'une manière générale des programmes de contrôle, c'est-à-dire, de leur principe général. Enfin, l'utilisateur peut terminer la session de travail du système, et tous les processus utilisateur sont forcés de se terminer. Quoi qu'il arrive, comme je l'ai déjà dit, le pilote et le programme de contrôle ont été conçus de telle manière que le programme de contrôle puisse être relancé à plusieurs reprises.

```

and g_fCDO_Opened, FALSE

MUTEX_RELEASE g_mtxCDO_State

```

Puisque notre seul client vient de "partir", nous invalidons le drapeau `g_fccDO_Opened`.

16.5.3. La procédure `CDO_DispatchDeviceControl`

```

MUTEX_ACQUIRE g_mtxCDO_State

mov edx, [esi].AssociatedIrp.SystemBuffer
mov edx, [edx]

mov ecx, ExEventObjectType
mov ecx, [ecx]
mov ecx, [ecx]

invoke ObReferenceObjectByHandle, edx, EVENT_MODIFY_STATE, ecx, \
    UserMode, addr pEventObject, NULL

.if eax == STATUS_SUCCESS

```

En obtenant du programme de contrôle le code de commande `IOCTL_KEYBOARD_ATTACH`, nous saisissons le mutex et nous vérifions la handle de l'objet "événement" qui nous est transmis. Nous avons déjà fait ceci pour le moniteur de processus (voir la partie 14). Si c'est un objet "événement" réel, alors nous avons deux possibilités : nous devons créer le filtre et le connecter à la pile du clavier ou le filtre existe déjà et il est connecté.

```

.if !g_fFiDO_Attached

    invoke KeyboardAttach
    mov [esi].IoStatus.Status, eax

```

Si le filtre n'est pas connecté, nous considérerons qu'il n'est pas encore créé. La procédure **KeyboardAttach** fera tout le nécessaire, en retournant le code approprié.

```

.if eax == STATUS_SUCCESS

```

```

        mov eax, pEventObject
        mov g_pEventObject, eax

        mov g_fFiDO_Attached, TRUE
        mov g_fSpy, TRUE

    .else
        invoke ObDereferenceObject, pEventObject
    .endif

```

Si la connexion s'est bien passée, nous mémorisons le pointeur sur l'objet "événement" dans la variable globale `g_pEventObject` et activons les drapeaux `g_fFiDO_Attached` et `g_fSpy`. Bien que le filtre soit déjà connecté, nous ne sommes pas obligés d'interdire l'accès à la variable `g_pEventObject`, à ce moment, puisque le drapeau `g_fSpy` est activé après l'initialisation de la variable `g_pEventObject`, et à ce moment, la procédure **FiDO_DispatchRead** n'installera pas la procédure de finalisation, et cela signifie que **ReadComplete** ne sera en aucun cas appelée et que personne d'autre que nous ne touchera `g_pEventObject`.

```

    .else

        LOCK_ACQUIRE g_EventSpinLock
        mov bl, al

```

Si le filtre est déjà connecté, il est nécessaire d'interdire l'accès à la variable `g_pEventObject`, puisque notre procédure de finalisation **ReadComplete** peut y accéder. Le blocage de spin est exigé parce que **ReadComplete** fonctionne à l'IRQL = DISPATCH_LEVEL.

```

        mov eax, g_pEventObject
        .if eax != NULL
            and g_pEventObject, NULL
            invoke ObDereferenceObject, eax
        .endif

        mov eax, pEventObject
        mov g_pEventObject, eax

        LOCK_RELEASE g_EventSpinLock, bl

```

Juste pour le cas où `g_pEventObject` contienne le pointeur sur l'objet événement, on décrémente le compteur de références et nous y ajoutons le pointeur sur le nouvel objet événement. Une petite explication est ici de rigueur. Ce code peut sembler, à première vue, idiot. Le fait est que, dans les exemples précédents, nous avons, pour simplifier, pris pour hypothèse un comportement correct du programme de contrôle du pilote. Mais, dans l'idéal, le pilote doit être blindé, même si son propre programme de commande ou quelque autre code effectue des actions imprévisibles. Dans la boîte à outils du DDK, se trouve le programme spécialisé dans les tests pour les pilotes Device Path Exerciser (dc2.exe), qui, entre d'autres tests, envoie au pilote d'énormes quantités de codes de commande avec des paramètres délibérément incorrects. Si le programme de contrôle envoie à deux reprises au pilote IOCTL_KEYBOARD_ATTACH, alors, grâce au code donné ci-dessus, nous pourrions correctement jongler avec deux objets "événement" et le mutex `g_mtxCDO_State` nous garantira de beaucoup de problèmes potentiels.

```

    MUTEX_ACQUIRE g_mtxCDO_State

```

Après avoir obtenu du programme de contrôle le code de commande IOCTL_KEYBOARD_DETACH, nous essayons d'ouvrir le filtre, toujours sous la protection du mutex.

```

    .if g_fFiDO_Attached

        and g_fSpy, FALSE

        invoke KeyboardDetach
        mov [esi].IoStatus.Status, eax

```

Si le filtre est connecté, nous invalidons le drapeau `g_fSpy` de sorte que **FiDO_DispatchRead** n'installe plus la procédure de finalisation, et nous essayons d'ouvrir et d'enlever le filtre.

```

    .if eax == STATUS_SUCCESS

```

```

        mov g_fFidoAttached, FALSE
    .endif

```

Si le filtre est ouvert avec succès, nous invalidons le drapeau approprié. Si l'ouverture du filtre a été impossible, ce drapeau restera dans l'état activé, ce qui nous donnera la possibilité de faire les choses correctement si on obtient de nouveaux (éventuels) IOCTL_KEYBOARD_ATTACH.

```

    LOCK_ACQUIRE g_EventSpinLock
    mov bl, al

    mov eax, g_pEventObject
    .if eax != NULL
        and g_pEventObject, NULL
        invoke ObDereferenceObject, eax
    .endif

    LOCK_RELEASE g_EventSpinLock, bl

```

Sous la protection du blocage de spin, nous enlevons la référence à l'objet "événement".

```

        invoke FillKeyData, [esi].AssociatedIrp.SystemBuffer, \
            [edi].Parameters.DeviceIoControl.OutputBufferLength

```

Après avoir obtenu du programme de contrôle le code de commande IOCTL_GET_KEY_DATA, nous copions dans le tampon utilisateur les structures KEY_DATA en notre possession jusqu'à ce moment. Je n'étudierai pas la procédure **FillKeyData**, pas plus qu'**AddEntry** et **RemoveEntry**, car si vous avez lu la partie sept du cycle "Utilisation des listes Lookaside", leur contenu ne doit pas présenter de complexité, mais vous trouverez les détails relatifs à la structure KEYBOARD_INPUT_DATA dans le DDK.

16.6. La procédure KeyboardAttach

```

    .if ( g_pFilterDeviceObject != NULL )

        mov status, STATUS_SUCCESS

```

Si la variable `g_pFilterDeviceObject` n'est pas égale à zéro, de toute évidence, elle contient le pointeur sur l'objet "périphérique filtre" et il est déjà probablement connecté à la pile.

```

    .else

```

S'il n'y a aucun filtre, créons en un.

```

        mov eax, g_pControlDeviceObject
        mov ecx, (DEVICE_OBJECT_PTR [eax]).DriverObject

        invoke IoCreateDevice, ecx, sizeof Fido_Device_Extension, NULL, \
            FILE_DEVICE_UNKNOWN, 0, FALSE, addr g_pFilterDeviceObject
        .if eax == STATUS_SUCCESS

```

L'objet "périphérique filtre" ne doit pas être nommé, de sorte qu'il ne puisse être ouvert directement par son nom. Puisque le filtre appartient à la pile, mais que c'est un objet qui y est introduit, alors clairement ce n'est pas à lui de décider, si une handle a été découverte ou non. C'est aux pilotes subalternes de s'en charger. Ce n'est pas toujours correct en réalité. Dans le cas de la pile du clavier, le pilote filtre de haut niveau Kbdclass possède l'objet nommé "périphérique filtre" KeyboardClassX" et précisément, il traite la requête IRP_MJ_CREATE. Le deuxième paramètre de la fonction **IoCreateDevice** détermine la taille de la zone de mémoire supplémentaire de l'objet "périphérique" (device extension), qui est décrite par la structure hypothétique DEVICE_EXTENSION. Hypothétique dans le sens qu'une telle structure n'existe pas. Vous déterminez vous-mêmes l'espace nécessaire dans la zone additionnelle de mémoires de l'objet "périphérique" et vous déterminez vous-mêmes la structure. L'extension de périphérique doit suivre immédiatement la structure DEVICE_OBJECT et être initialisée à zéro. Dans notre cas, c'est la structure Fido_Device_Extension. L'utilisation de l'extension de périphérique permet au pilote de créer commodément autant d'objets "périphérique" qu'il lui plait et de stocker toutes les données le concernant dans ces objets.

```

        invoke IoGetDeviceObjectPointer, addr g_usTargetDeviceName, \
            FILE_READ_DATA, addr pTargetFileObject, \

```

```

        addr pTargetDeviceObject
    .if eax == STATUS_SUCCESS

```

Vous vous rappelez sans doute que la fonction **IoGetAttachedDevice** renvoie toujours le pointeur sur l'objet "périphérique", qui est situé sur au sommet de la pile. Nous avons utilisé la fonction **IoGetDeviceObjectPointer** pour obtenir le pointeur sur le sommet de la pile grâce au nom préalablement connu de nous d'un des objets "périphérique" qui appartenait à la pile. Pour les pilotes PnP, en ce sens, c'est plus simple, puisque le gestionnaire PnP leur donne un pointeur sur l'objet racine de la pile – l'objet "périphérique physique". Ce que je veux dire, c'est que pour vous connecter à la pile, vous avez besoin d'un pointeur sur n'importe quel objet de la pile. Comment vous l'obtenez importe peu.

```

    mov eax, g_pDriverObject
    and (DRIVER_OBJECT PTR [eax]).DriverUnload, NULL

```

Comme maintenant nous devrions être connectés à la pile, alors les IRP peuvent passer par nous. Après avoir activé le drapeau `g_fSpy`, nous y établirons la procédure de finalisation. Quand ces IRP seront finalisés, nous ne le savons pas, mais nous savons que le pilote ne peut pas être déchargé avant ce moment, puisque la procédure de finalisation est située dans le corps même du pilote. Par conséquent le plus simple est de faire un pilote non déchargeable.

```

PDEVICE_OBJECT
IoGetAttachedDevice(
    IN PDEVICE_OBJECT pDeviceObject
)
{
    while pDeviceObject->AttachedDevice {
        pDeviceObject = pDeviceObject->AttachedDevice
    }
    return pDeviceObject
}

PDEVICE_OBJECT
IoAttachDeviceToDeviceStack(
    IN PDEVICE_OBJECT pSourceDevice,
    IN PDEVICE_OBJECT pTargetDevice
)
{
    PDEVICE_OBJECT pTopMostDeviceObject
    PDEVICE_EXTENSION pSourceDeviceExtension

    pSourceDeviceExtension = pSourceDevice->DeviceObjectExtension

    ExAcquireSpinLock( &IopDatabaseLock, ... )

    pTopMostDeviceObject = IoGetAttachedDevice( pTargetDevice )

    if pTopMostDeviceObject->Flags & DO_DEVICE_INITIALIZING
        ||
        pTopMostDeviceObject->DeviceObjectExtension->ExtensionFlags &
        (DOE_UNLOAD_PENDING | DOE_DELETE_PENDING | DOE_REMOVE_PENDING |
        DOE_REMOVE_PROCESSED) {
        pTopMostDeviceObject = NULL
    } else {
        pTopMostDeviceObject->AttachedDevice = pSourceDevice
    }
}

```

```

    pSourceDevice->AlignmentRequirement =
        pTopMostDeviceObject->AlignmentRequirement
    pSourceDevice->SectorSize = pTopMostDeviceObject->SectorSize
    pSourceDevice->StackSize = pTopMostDeviceObject->StackSize + 1

    if pTopMostDeviceObject ->DeviceObjectExtension->ExtensionFlags & \
        DOE_START_PENDING {

        pSourceDevice->DeviceObjectExtension->ExtensionFlags |= \
            DOE_START_PENDING
    }

    pSourceDeviceExtension->AttachedTo = pTopMostDeviceObject
}

ExReleaseSpinLock( &IopDatabaseLock, ... )

return pTopMostDeviceObject
}

```

D'abord, la fonction **IoAttachDeviceToDeviceStack** obtient un pointeur sur la structure `DEVOBJ_EXTENSION` (à ne pas confondre avec la structure optionnelle `DEVICE_EXTENSION`). Dans le champ `ExtensionFlags` de cette structure, se trouvent quelques drapeaux intéressant la fonction **IoAttachDeviceToDeviceStack**. Ensuite la base de données du gestionnaire d'entrées/sorties est bloquée et dans `pTopMostDeviceObject` sera placé le pointeur sur l'objet "périphérique" qui est situé au sommet de la pile. Comme la base de données du gestionnaire d'entrées/sorties est bloquée, l'état de la pile ne changera pas jusqu'au déblocage. Si l'objet "périphérique" n'est pas encore initialisé ou si le périphérique ou son pilote sont signalés pour fermeture ou sont déjà dans le processus de fermeture, la fonction **IoAttachDeviceToDeviceStack** refuse d'attacher un nouvel objet à la pile et retourne `NULL`. Sinon, dans les champs `AttachedDevice` et `AttachedTo`, seront stockés les pointeurs sur les objets "périphérique" correspondants (c'est en cela que consiste le processus de connexion d'un nouvel objet à la pile) et dans l'objet connecté, les champs `AlignmentRequirement`, `SectorSize` et `StackSize` seront mis à jour. `AlignmentRequirement` et `SectorSize` sont importants pour les périphériques de stockage, et `StackSize` doit être incrémenté de toutes façons, puisque la profondeur de la pile a augmenté d'un objet (voir les détails dans l'article précédent). Vous porterez votre attention sur le fait que la connexion ne se fait pas avec l'objet dont le pointeur est transmis dans le paramètre `pTargetDevice`, mais qui n'était pas situé au sommet de la pile. Si dans l'espace de temps entre l'obtention du pointeur `pTargetDevice` et l'appel à **IoAttachDeviceToDeviceStack**, quelqu'un a le temps de connecter son objet à la pile, les valeurs retournées `pTargetDevice` et `pTopMostDeviceObject` seront différentes. Quoi qu'il arrive, la valeur retournée par **IoAttachDeviceToDeviceStack**, en cas de succès, est un pointeur sur l'objet "périphérique", auquel a été connecté le filtre. Mais le filtre est maintenant le sommet de la pile et obtient le premier tous les IRP destinés à cette pile. Au commencement de l'article, nous avons expliqué que le thread d'entrées brutes ouvre un des objets de la pile du clavier et, en utilisant sa handle (plus précisément la handle de l'objet "fichier", qui correspond à l'objet "périphérique"), il dirige sur lui les requêtes de lecture. Si des IRP sont prévus pour un périphérique donné en-dessous dans la pile, alors comment passent-ils dans le filtre ? Le sous-système d'entrées-sorties se comporte de façon analogue avec les fonctions **IoGetAttachedDevice** et **IoAttachDeviceToDeviceStack**, dans le sens où pour le destinataire des IRP, il utilise un pointeur sur le sommet de la pile. Ici, par exemple, voilà ce que fait la fonction **ZwReadFile**.

```

NTSTATUS
ZwReadFile(
    IN HANDLE hFile,
    . . .
)
{

    PFILE_OBJECT    pFileObject
    PDEVICE_OBJECT  pDeviceObject

    ObReferenceObjectByHandle( hFile, ... &pFileObject ... )
}

```

```

    pDeviceObject = IoGetRelatedDeviceObject( pFileObject )
    . . .
}

```

La fonction **IoGetRelatedDeviceObject** (voir son code source dans l'article précédent) renvoie un pointeur sur l'objet "périphérique" le plus élevé dans la pile. Mais si l'IRP est construit par le pilote, pour ainsi dire à la main (voir le code source de la procédure **QueryPnpDeviceState** dans l'article précédent), alors il sera envoyé directement au périphérique cible et il sera, bien sûr, impossible d'intercepter cette requête à l'aide du filtre, si le filtre n'est pas situé en dessous dans la pile.

```

    invoke IoAttachDeviceToDeviceStack, g_pFilterDeviceObject, \
        pTargetDeviceObject
    .if eax != NULL
        mov edx, eax

        mov ecx, g_pFilterDeviceObject
        mov eax, (DEVICE_OBJECT ptr [ecx]).DeviceExtension
        assume eax:ptr FiDO_DEVICE_EXTENSION
        mov [eax].pNextLowerDeviceObject, edx
        push pTargetFileObject
        pop [eax].pTargetFileObject
        assume eax:nothing
    .endif

```

Si **IoAttachDeviceToDeviceStack** nous a connecté à la pile, nous remplissons la structure **FiDO_DEVICE_EXTENSION**. Nous y plaçons le pointeur sur l'objet, auquel nous avons été connectés et le pointeur sur l'objet "dossier" associé à l'objet périphérique concerné (voir les détails dans l'article précédent). En le fermant, nous avons dû appeler **ObDereferenceObject** pour cet objet "fichier".

```

    assume edx:ptr DEVICE_OBJECT
    assume ecx:ptr DEVICE_OBJECT

    mov eax, [edx].DeviceType
    mov [ecx].DeviceType, eax

    mov eax, [edx].Flags
    and eax, DO_DIRECT_IO + DO_BUFFERED_IO + DO_POWER_PAGABLE
    or [ecx].Flags, eax

```

De notre côté, il est nécessaire de mettre à jour plusieurs drapeaux dans notre objet filtre. Le fait est que pour le gestionnaire d'entrées/sorties, notre objet doit apparaître également comme l'objet, auquel nous avons été connectés. Par exemple, le drapeau **DO_BUFFERED_IO** indique au gestionnaire d'entrées/sorties qu'avec des opérations de lecture/écriture, il doit copier les tampons utilisateur dans l'espace d'adressage du système, c'est-à-dire, employer une méthode d'entrée-sortie **METHOD_BUFFERED**. Les drapeaux **DO_DIRECT_IO** et **DO_BUFFERED_IO**, naturellement, sont mutuellement exclusifs. Bien que les drapeaux qu'utilise le périphérique **KeyboardClass0**, à savoir **DO_BUFFERED_IO** et **DO_POWER_PAGEABLE**, soient déjà connus de nous, nous utilisons un mécanisme plus général et plus universel.

```

    and [ecx].Flags, not DO_DEVICE_INITIALIZING

```

La fonction **IoCreateDevice** crée l'objet "périphérique" avec le drapeau **DO_DEVICE_INITIALIZING** activé. Jusqu'ici, nous n'avons pas été concernés par ce détail car les périphériques ont seulement été créés dans la procédure **DriverEntry**. Le fait est qu'à la sortie de **DriverEntry**, le gestionnaire d'entrées/sorties (dans la fonction **IoReadyDeviceObjects**) désactive lui-même ce drapeau dans tous les objets "périphérique", créés par le pilote. Mais si nous ne créons pas le périphérique dans **DriverEntry**, il est nécessaire de désactiver le drapeau **DO_DEVICE_INITIALIZING** indépendamment, sinon personne ne pourra se connecter à l'objet non-initialisé, comme vous l'avez vu récemment dans le code d'**IoAttachDeviceToDeviceStack**. En outre, ce drapeau est vérifié lors d'autres opérations.

16.7. La procédure KeyboardDetach

```
.if g_pFilterDeviceObject != NULL

    mov eax, g_pFilterDeviceObject
    or (DEVICE_OBJECT ptr [eax]).Flags, DO_DEVICE_INITIALIZING
```

Avant de nous déconnecter de la pile, nous devons regarder si nous nous trouvons à son sommet lui-même ou si quelqu'un s'est également connecté à nous auparavant. Nous ne pouvons pas bloquer la base de données du gestionnaire d'entrées/sorties, mais nous pouvons éviter de nouvelles connexions, pendant que nous vérifions s'il y a quelqu'un au-dessus de nous.

```
PDEVICE_OBJECT
IoGetAttachedDeviceReference(
    IN PDEVICE_OBJECT pDeviceObject
)
{
    ExAcquireSpinLock( &IopDatabaseLock, ... )

    pDeviceObject = IoGetAttachedDevice( pDeviceObject )
    ObReferenceObject( pDeviceObject )

    ExReleaseSpinLock( &IopDatabaseLock, ... )

    return pDeviceObject
}
```

Ici tout devrait être clair pour tout le monde.

```
    invoke IoGetAttachedDeviceReference, g_pFilterDeviceObject
    mov pTopmostDeviceObject, eax

    .if eax != g_pFilterDeviceObject

        mov eax, g_pFilterDeviceObject
        and (DEVICE_OBJECT ptr [eax]).Flags, not DO_DEVICE_INITIALIZING
```

Si le pointeur retourné par la fonction **IoGetAttachedDeviceReference**, n'est pas le pointeur sur notre filtre, cela signifie que quelqu'un s'est connecté à nous. Dans ce cas, nous ne nous déconnecterons pas de la pile et nous activons le drapeau DO_DEVICE_INITIALIZING. Si nous appelons **IoDetachDevice**, alors nous "démolissons" tout simplement la pile, puisqu'**IoDetachDevice** ne fait pas de vérification. Déconnecter l'objet "périphérique" de la pile consiste simplement à remettre à zéro les pointeurs correspondants dans les objets connectés.

```
VOID
IoDetachDevice(
    IN OUT PDEVICE_OBJECT pTargetDeviceObject
)
{
    PDEVICE_OBJECT    pDeviceToDetach
    PDEVOBJ_EXTENSION pDeviceToDetachExtension

    ExAcquireSpinLock( &IopDatabaseLock, ... )

    pDeviceToDetach          = pTargetDeviceObject->AttachedDevice
    pDeviceToDetachExtension = pDeviceToDetach->DeviceObjectExtension

    pDeviceToDetachExtension->AttachedTo = NULL
    pTargetDeviceObject->AttachedDevice = NULL

    if pTargetDeviceObject->DeviceObjectExtension->ExtensionFlags &
        (DOE_UNLOAD_PENDING | DOE_DELETE_PENDING | DOE_REMOVE_PENDING)
        && pTargetDeviceObject->ReferenceCount == 0
```

```

    {
        // Termine la déconnexion ou suppression
    }

    ExReleaseSpinLock( &IopDatabaseLock, ... )
}

```

Après avoir ouvert le périphérique depuis la pile, **IoDetachDevice** vérifie s'il n'attend pas la déconnexion, mais le déchargement de son pilote. Et dans ce cas, si le compteur de références de l'objet est égal à zéro, alors il lance les opérations différées.

```

    .else

        mov eax, g_pFilterDeviceObject
        mov eax, (DEVICE_OBJECT ptr [eax]).DeviceExtension
        mov ecx, (FIDO_DEVICE_EXTENSION ptr [eax]).pTargetFileObject

        fastcall ObfDereferenceObject, ecx

        mov eax, g_pFilterDeviceObject
        mov eax, (DEVICE_OBJECT ptr [eax]).DeviceExtension
        mov eax, (FIDO_DEVICE_EXTENSION ptr [eax]).pNextLowerDeviceObject

        invoke IoDetachDevice, eax

        mov status, STATUS_SUCCESS

```

Si nous sommes au sommet de la pile, le compteur de références de l'objet fichier lié à l'objet périphérique est décrémenté, et nous sommes déconnectés de la pile. Revalider le drapeau DO_DEVICE_INITIALIZING n'a pas de sens, car maintenant nous sommes en train d'enlever le filtre.

```

        mov eax, g_pFilterDeviceObject
        and g_pFilterDeviceObject, NULL
        invoke IoDeleteDevice, eax

```

Nous enlevons l'objet "périphérique filtre", mais le pilote n'est pas encore déchargé, car il se peut qu'il y ait des IRP en attente de finalisation, contenant un pointeur sur notre procédure de finalisation.

```

    .endif

    invoke ObDereferenceObject, pTopmostDeviceObject

```

Contrairement à la fonction **IoGetAttachedDevice**, la fonction **IoGetAttachedDeviceReference** incrémente le compteur de références de l'objet sur lequel elle retourne un pointeur. Ceci garantit que l'objet ne sera pas enlevé. Si nous étions au sommet de la pile, alors le compteur de références de notre objet "périphérique filtre" a augmenté et **IoDeleteDevice** ne sera pas en mesure de l'enlever.

```

VOID
IoDeleteDevice(
    IN PDEVICE_OBJECT pDeviceObject
)
{
    ...

    ExAcquireSpinLock( &IopDatabaseLock, ... )

    pDeviceObject->DeviceObjectExtension->ExtensionFlags |= \
        DOE_DELETE_PENDING

    if pDeviceObject->ReferenceCount == 0 {
        // Termine la déconnexion ou suppression
    }
}

```

```

    ExReleaseSpinLock( &IopDatabaseLock, ... )
}

```

Mais **IoDeleteDevice** ajoutera le drapeau `DO_DELETE_PENDING`, après avoir noté le fait que l'objet "périphérique" est en attente de fermeture. Quand nous appelons **ObDereferenceObject**, le compteur de références deviendra égal à 0, le gestionnaire d'objets verra que l'objet doit être enlevé et entreprendra les démarches appropriées.

Décortiquons maintenant les procédures de traitement des requêtes du filtre.

16.8.1. La procédure `FiDO_DispatchPower`

```

invoke PoStartNextPowerIrp, pIrp

IoSkipCurrentIrpStackLocation pIrp

mov eax, pDeviceObject
mov eax, (DEVICE_OBJECT ptr [eax]).DeviceExtension
mov eax, (FiDO_DEVICE_EXTENSION ptr [eax]).pNextLowerDeviceObject

invoke PoCallDriver, eax, pIrp

```

Les IRP de type `IRP_MJ_POWER` sont traités avec une méthode différente de tous les autres types d'IRP.

La macro **IoCopyCurrentIrpStackLocationToNext** a été décortiquée en détail dans l'article précédent (nous l'utiliserons dans la procédure **FiDO_DispatchRead**). La macro **IoSkipCurrentIrpStackLocation** est beaucoup plus simple.

```

IoSkipCurrentIrpStackLocation MACRO pIrp:REQ
    mov eax, pIrp
    inc (_IRP_PTR [eax]).CurrentLocation
    add (_IRP_PTR [eax]).Tail.Overlay.CurrentStackLocation, \
        sizeof IO_STACK_LOCATION
ENDM

```

Vous devez vous rappeler de l'article précédent que la fonction **IoCallDriver** avant d'appeler la procédure de traitement du pilote, déplace le bloc de pile courant d'une position vers le bas.

```

Irp->CurrentLocation--
pIrp->Tail.Overlay.CurrentStackLocation -= sizeof(IO_STACK_LOCATION)

```

Si nous utilisons auparavant la macro **IoSkipCurrentIrpStackLocation**, alors il s'ensuit que le pointeur sur le bloc de pile ne change pas du tout et que le pilote subalterne obtient le même bloc de pile que le pilote qui a appelé **IoCallDriver** (**PoCallDriver**). L'appel de la macro **IoSkipCurrentIrpStackLocation** est une simple optimisation. En fait, si nous devons installer la procédure de finalisation, alors l'appel de la macro **IoCopyCurrentIrpStackLocationToNext** copiera notre bloc de pile dans le bloc de pile du pilote subalterne (les champs `Control`, `CompletionRoutine` et `Context`, comme vous vous le rappelez, ne sont pas copiés). Ainsi le pilote subalterne obtiendra quand même les mêmes paramètres. En utilisant la macro **IoSkipCurrentIrpStackLocation** au lieu d'**IoCopyCurrentIrpStackLocationToNext**, nous évitons l'opération inutile de copie des blocs de pile. Mais, je le répète, ceci ne peut être fait que s'il n'est pas nécessaire d'installer une procédure de finalisation, ce qui est tout à fait compréhensible.

16.8.2. La procédure `FiDO_DispatchPassThrough`

```

IoSkipCurrentIrpStackLocation pIrp

mov eax, pDeviceObject
mov eax, (DEVICE_OBJECT ptr [eax]).DeviceExtension
mov eax, (FiDO_DEVICE_EXTENSION ptr [eax]).pNextLowerDeviceObject

invoke IoCallDriver, eax, pIrp
ret

```

Ici, nous transférons simplement les IRP au pilote subalterne.

16.8.3. La procédure FiDO_DispatchRead

```
.if g_fSpy
```

Après avoir obtenu une requête de type IRP_MJ_READ adressée au filtre, nous regardons si le drapeau `g_fSpy` est activé. Si oui, alors nous devons installer la procédure de finalisation.

```
lock inc g_dwPendingRequests
```

Nous incrémentons atomiquement la valeur du compteur de requêtes en attente `g_dwPendingRequests`. Quand l'IRP se terminera, le système appellera notre procédure de finalisation **ReadComplete**, lira le code de la touche et décrémentera `g_dwPendingRequests`. "Atomiquement" signifie qu'un seul thread, même sur une machine multi- processeurs, pourra changer la valeur de la variable, et l'IRQL auquel il s'exécute n'a aucune importance. Même si le thread, qui s'exécute sur l'autre processeur, essaye en même temps (littéralement sur une machine MP) d'exécuter le même code, il obtiendra la valeur déjà mise à jour par le premier thread. C'est dû grâce à l'utilisation du préfixe **lock**. Après avoir vu ce préfixe, le processeur bloque le bus de données pendant la période d'exécution de l'instruction. D'autres processeurs ne peuvent pas, à ce moment-là, accéder à cette région de mémoire et la changer. Même si cette région de mémoire est cachée par plusieurs processeurs, l'action entraînera le mécanisme de garantie de cohérence du cache (processor's cache coherency mechanism) et les caches d'autres processeurs seront invalidés, ce qui aura comme conséquence que les processeurs devront charger de nouveau le cache avec le contenu déjà mis à jour de la région de mémoire. Le préfixe **lock** ne peut pas être utilisé avec toutes les instructions, mais certaines (par exemple, `xchg`) sont toujours effectuées avec ce préfixe. Pour plus de détails, voir "Intel Architecture Software Developer's Manual". Le système (tant en mode noyau qu'utilisateur) exporte le jeu complet des fonctions **Interlocked**, qui permettent l'accès atomique, mais nous pouvons utiliser les ressources de l'assembleur.

```
IoCopyCurrentIrpStackLocationToNext pIrp
```

```
IoSetCompletionRoutine pIrp, ReadComplete, NULL, TRUE, TRUE, TRUE
```

Nous installons la procédure de finalisation (voir les détails dans l'article précédent). Quand l'IRP se terminera, nous pourrons connaître le code de la touche.

```
.else
```

```
IoSkipCurrentIrpStackLocation pIrp
```

Si le drapeau `g_fSpy` est nul, la procédure **FiDO_DispatchRead** se comporte de façon analogue à la procédure **FiDO_DispatchPassThrough**.

```
.endif
```

```
mov eax, pDeviceObject
mov eax, (DEVICE_OBJECT ptr [eax]).DeviceExtension
mov eax, (FiDO_DEVICE_EXTENSION ptr [eax]).pNextLowerDeviceObject

invoke IoCallDriver, eax, pIrp

ret
```

Vous noterez que toutes les procédures **FiDO_XXX** renvoient le code qu'a renvoyé la fonction **IoCallDriver** (`PoCallDriver`). De même, **DriverDispatch** le renvoie au système.

16.9. La procédure ReadComplete

Bien, voici en conclusion, la procédure **ReadComplete**, l'endroit même où se produisent les événements principaux, à savoir l'obtention des codes de touches. Nous avons installé l'adresse de cette procédure dans notre bloc de pile par l'appel de la macro **IoSetCompletionRoutine**. Quand l'IRP se termine, la fonction **IoCompleteRequest** appelle consécutivement toutes les procédures de finalisation. L'article précédent était pratiquement entièrement consacré à ceci. L'IRP se termine à la suite d'une interruption matérielle post-opératoire (dans notre cas l'interruption du contrôleur de clavier), et cela signifie dans le contexte d'un thread aléatoire et à un IRQL élevé.

```
.if [esi].IoStatus.Status == STATUS_SUCCESS
```

```
mov edi, [esi].AssociatedIrp.SystemBuffer
```

```
assume edi:ptr KEYBOARD_INPUT_DATA
```

Si l'IRP se termine avec le code de succès, alors son tampon contient au moins une structure KEYBOARD_INPUT_DATA, qui porte en elle le code de touche désiré.

```
mov ebx, [esi].IoStatus.Information
```

Le champ IoStatus.Information contient la taille de la partie utile du tampon et doit être un multiple de la taille de la structure KEYBOARD_INPUT_DATA.

```
and cEntriesLogged, 0
.while sdword ptr ebx >= sizeof KEYBOARD_INPUT_DATA

    movzx eax, [edi].MakeCode
    mov KeyData.dwScanCode, eax

    movzx eax, [edi].Flags
    mov KeyData.Flags, eax

    invoke AddEntry, addr KeyData

    inc cEntriesLogged

    add edi, sizeof KEYBOARD_INPUT_DATA
    sub ebx, sizeof KEYBOARD_INPUT_DATA
.endw

assume edi:nothing
```

Nous faisons passer les champs de la structure KEYBOARD_INPUT_DATA qui nous intéressent dans notre structure KEY_DATA_ENTRY et l'attachons à la liste doublement liée g_KeyDataListHead. Nous faisons ceci dans la fonction **AddEntry** et sous la protection du blocage de spin g_KeyDataSpinLock. Le blocage de la liste est nécessaire, vous le comprenez bien, pour avoir un accès exclusif à la liste, et cela doit être un blocage de spin parce que la procédure **ReadComplete** est exécutée à l'IRQL = DISPATCH_LEVEL. Le DDK affirme que les procédures de la finalisation peuvent être appelées à IRQL <= DISPATCH_LEVEL, mais dans notre cas, nous serons toujours strictement à l'IRQL = DISPATCH_LEVEL. La fonction **KeyboardClassServiceCallback** du pilote Kbdclass, qui exécute l'IRP lui-même, utilise **KeAcquireSpinLockAtDpcLevel** et **KeReleaseSpinLockFromDpcLevel** pour le blocage.

```
VOID
KeyboardClassServiceCallback(
    . . .
)
{
    . . .

    //
    // N.B. Nous pouvons utiliser KeAcquireSpinLockAtDpcLevel, au lieu de
    // KeAcquireSpinLock, parce que cette routine tourne déjà à
    // DISPATCH_IRQL.
    //

    KeAcquireSpinLockAtDpcLevel( &deviceExtension->SpinLock );

    . . .

    //
    // Libère le blocage de spin de la queue des données en entrée.
    //

    KeReleaseSpinLockFromDpcLevel( &deviceExtension->SpinLock );
}
```

```

    . . .

    IoCompleteRequest( irp, IO_KEYBOARD_INCREMENT );

    . . .
}

```

Mais néanmoins, j'utilise les macros LOCK_ACQUIRE et LOCK_RELEASE (elles me plaisent tellement!).

```

    .if cEntriesLogged != 0

        LOCK_ACQUIRE g_EventSpinLock
        mov bl, al

        .if g_pEventObject != NULL
            invoke KeSetEvent, g_pEventObject, 0, FALSE
        .endif

        LOCK_RELEASE g_EventSpinLock, bl

    .endif

```

Si nous avons de nouvelles données, nous en informons le programme de contrôle, en signalant l'objet "événement". J'utilise également un blocage, pour être sûr que g_pEventObject contient toujours un pointeur réel.

```

    .if [esi].PendingReturned
        IoMarkIrpPending esi
    .endif

```

Puisque nous renvoyons un code différent de STATUS_MORE_PROCESSING_REQUIRED depuis la procédure de finalisation, nous devons suivre la règle n°6 (voir la partie 15).

```

    lock dec g_dwPendingRequests

```

La requête est traitée - le compteur g_dwPendingRequests est atomiquement décrémenté.

```

    mov eax, STATUS_SUCCESS

```

La finalisation de l'IRP doit être continuée. Des explications de l'article précédent, vous devez vous rappeler qu'il est possible de renvoyer STATUS_MORE_PROCESSING_REQUIRED ou n'importe quel autre code depuis les procédures de la finalisation. Le code STATUS_SUCCESS est employé pour plus de clarté.

16.10. Le programme de contrôle

Vous examinerez vous-même le code du programme de contrôle. Il n'y a rien de fondamentalement nouveau là-dedans. J'expliquerai seulement plusieurs points. D'abord, statistiquement, le taux de frappe de texte à approximativement 10 signes par seconde semble un bon ratio. Ainsi, au maximum, nous pourrions obtenir environ 20 structures KEY_DATA par seconde, et en même temps on peut ne pas toucher au clavier pendant très longtemps. Par conséquent, sauf en cas d'excès de requêtes au pilote, nous recueillons les informations accumulées toutes les secondes. Et s'il n'y a rien à recueillir, nous ne nous posons pas d'autres questions. Cette logique de travail est obtenue grâce à la mise en sommeil du thread pendant un certain temps et son attente de l'événement que signale le pilote. Deuxièmement, comme 20 structures KEY_DATA font considérablement moins d'une page, nous avons utilisé les entrée-sortie avec tampon et les informations sont recueillies en appelant **DeviceIoControl**. Si on voulait passer de grands volumes de données (disons, par exemple, plusieurs pages), alors il vaut mieux employer la méthode METHOD_NEITHER, et au lieu de **DeviceIoControl - ReadFile**. Troisièmement, l'utilisateur peut fermer le programme de contrôle à l'aide de la souris ou à l'aide du clavier. S'il utilise la souris, alors le pilote ne sera pas déchargé, puisque le dernier IRP passé par le pilote contient le pointeur sur notre procédure de finalisation et se trouve maintenant dans la queue du pilote Kbdclass. Pour que le programme de contrôle puisse décharger le pilote, l'utilisateur doit appuyer sur une touche. Pour cela, en affichant un message approprié, nous donnons les instructions nécessaires à l'utilisateur. Et pour finir, l'annulation d'un IRP. S'il était possible de supprimer cet IRP de mauvais augure en attente de finalisation, alors il ne serait pas nécessaire d'effrayer l'utilisateur avec des messages étranges. Nous supprimerions simplement cette requête et déchargerions le pilote. Et ce mécanisme existe. Le pilote Kbdclass ne supporte l'annulation que d'un type IRP et il s'agit précisément de IRP_MJ_READ. Le problème est que

supprimer un IRP situé dans la queue d'un autre pilote n'est pas aussi simple. Dans son livre, "Programming The Windows Driver Model" 2nd Edition, Walter Oney donne deux méthodes d'annulation des IRP qui ne sont pas à nous. La première ne correspond pas exactement, mais la deuxième... Si la deuxième ne correspond pas, alors il ne reste qu'à organiser sa propre queue et y placer tous les IRP arrivés de type IRP_MJ_READ, puis de demander au pilote subalterne de traiter leurs copies. Interceptor la finalisation de ces IRP dédoublés, extraire leurs originaux de la queue et en extraire les données nécessaires. Si sa queue se trouve dans le filtre, alors l'annulation de l'IRP devient un problème technique. Comment un tel scénario est possible pratiquement, je ne le sais pas, car des complexités imprévues peuvent surgir dans sa réalisation.

Bien, et pour terminer. Dans les sources, vous trouverez en fait deux filtres. Le second - MouSpy, est obtenu en remplaçant les mots "keyboard", "kbd" et autres dans les objets parents par leurs analogues "mouse". Et bien sûr, je n'ai pas pu m'empêcher d'ajouter encore quelque chose. Par conséquent, ce filtre ne fait pas que suivre passivement les événements "souris", mais il peut y faire quelques ajustements. Mais, si vous avez un clavier/souris USB, alors vous ne réussirez pas, très probablement, à vous connecter aux filtres. De toutes façons, je n'ai pas réussi à connecter le filtre à la pile pour la souris USB, par contre, je n'ai pas de clavier USB. La raison en est qu'intérieurement la fonction **IoGetDeviceObjectPointer** appelle la fonction **ZwOpenFile**, et celle-ci, à son tour, crée la requête IRP_MJ_CREATE et l'envoie à la pile (voir l'article précédent). Voici trois piles de souris sur une de mes machines - la première pour la souris PS/2 classique, la seconde pour la session terminal et la troisième pour la souris USB.

```
kd> !drvobj mouclass
Driver object (816a68e8) is for:
  \Driver\Mouclass
Driver Extension List: (id , addr)

Device Object list:
812b5a20  8169e820  816a3030

kd> !devstack 816a3030
!DevObj  !DrvObj          !DevExt  ObjectName
> 816a3030  \Driver\Mouclass  816a30e8  PointerClass0
  816a63a8  \Driver\nmfilter  816a6460  0000006c
  816a6530  \Driver\i8042prt  816a65e8
  8192f3e8  \Driver\ACPI      81969008  00000051
!DevNode 818685e8 :
  DeviceInst is "ACPI\PNP0F13\3&13c0b0c5&0"
  ServiceName is "i8042prt"

kd> !devstack 8169e820
!DevObj  !DrvObj          !DevExt  ObjectName
> 8169e820  \Driver\Mouclass  8169e8d8  PointerClass1
  8169ea08  \Driver\TermDD    8169eac0  RDP_CONSOLE1
  8197f970  \Driver\PnpManager 8197fa28  00000038
!DevNode 8197f828 :
  DeviceInst is "Root\RDP_MOU\0000"
  ServiceName is "TermDD"

kd> !devstack 812b5a20
!DevObj  !DrvObj          !DevExt  ObjectName
> 812b5a20  \Driver\Mouclass  812b5ad8  PointerClass2
  813c1e20  \Driver\mouhid    813c1ed8
  815f2a90  \Driver\HidUsb    815f2b48  00000074
!DevNode 81361008 :
  DeviceInst is "HID\Vid_09da&Pid_000a\6&3a964113&0&0000"
  ServiceName is "mouhid"
```

Apparemment, un des pilotes dans la pile (mouhid probablement) refuse de traiter une requête IRP_MJ_CREATE, en renvoyant le code STATUS_SHARING_VIOLATION, c'est-à-dire, le partage de fichier est interdit (sous forme de l'objet "fichier" associé à l'objet "périphérique"). De toutes façons, je ne rentrerai pas dans plus de détails. Je n'ai

pas réussi à connecter l'USB à la pile... un point c'est tout!, car "on ne va pas couper les cheveux en quatre"... Comment traiterons-nous IRP_MN_QUERY_REMOVE_DEVICE, IRP_MN_REMOVE_DEVICE et IRP_MN_SURPRISE_REMOVAL, si l'utilisateur branche/débranche le clavier/souris au port d'USB ? Aussi, cherchez ces réponses ailleurs ou attendez l'article suivant (si jamais je l'écris;).

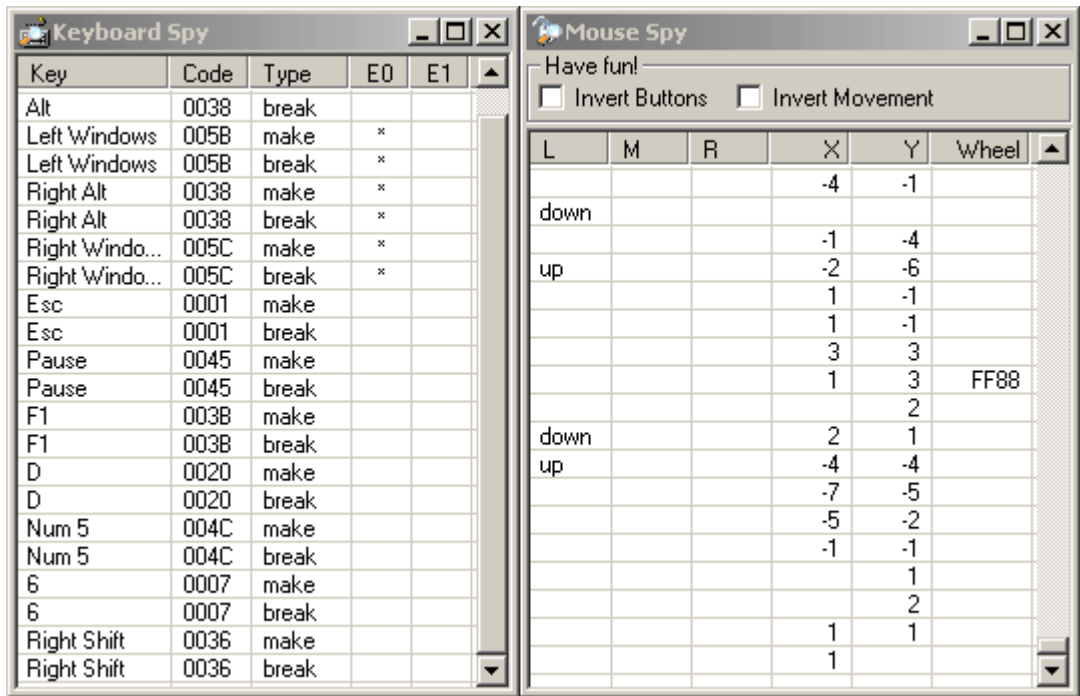


Fig. 16-1 KbdSpy et MouSpy en action.