Debreceni Egyetem
Természettudományi Kar

Diplomamunka

OS/2 rendszerprogramozás

Készítette: Kocsis Péter
Témavezető: Dr. Bölcskei András

Debrecen, 2002.

Tartalomjegyzék

1. Bevezetés **4**
1.1. Az operációs rendszer 4
1.2. Probléma a hosszú fájlnevekkel 5

2. Az OS/2 rendszer felépítése **6**
2.1. Fájlrendszer a VDM-ben és a Supervisor Call (SVC) 6
2.2. Eszközmeghajtók 7
2.2.1. A Physical Device Driver (PDD) 7
2.2.2. A Virtual Device Driver (VDD) 7
2.3. Egyéb kiszolgáló programok 8

3. A VLFN felépítése **10**
3.1. A memóriát adó PDD (VLFNMEM.SYS) 12
3.1.1. A fejléc és a változók 12
3.1.2. A kódszegmens 13
3.2. A híd VDD (VLFN.SYS) 16
3.2.1. A szegmensek 16
3.2.2. A VDD inicializálása 17
3.2.3. VDM események kezelése 18
3.2.4. Információcsere a daemonnal 18
3.2.5. Információcsere a VDM-mel 20
3.2.6. A VDM kéréseinek kiszolgálása 22
3.2.7. Egyéb segédeljárások 29
3.3. A DOS-os átirányító alkalmazás (LFN.COM) 33
3.3.1. Az inicializálás 33
3.3.2. A DOS kernel megpatkolása 35
3.3.3. Az AAh megszakítás 36
3.3.4. Az egyes rutinok 37
3.3.5. Az ABh megszakítás 38
3.4. A daemon (VLFND.EXE) 40
3.4.1. A névátalakítás 41
3.4.2. A főprogram 41
3.4.3. LFN_Unit 47

3.4.4. Az LFN_EA unit 50
3.4.5. Az LFN_Cache unit 51

4. A VLFN üzembehelyezése 55

5. Összefoglaló 56
5.1. Felhasznált programok, felhasznált irodalom 57
5.2. Köszönetnyilvánítás 57

1. fejezet

Bevezetés

1.1. Az operációs rendszer

Manapság ha valaki azt hallja, hogy OS/2, vagy nem tudja mi az, vagy legyint egyet, mondván, az már a régmúlt elavult operációs rendszere. Nagyon kevesen vannak (főleg kishazánkban) akik tudják, ez nem így van. Még ma is igen sok helyen, döntően bankokban és egyéb, nagy biztonságot és stabilitást igénylő helyeken használják, habár az OS/2 mindig is az IBM mostohagyerekének számított: az átlagos felhasználóknak (egy-két apró kivételtől eltekintve) sohasem ajánlották, habár minden tudása megvan az átlagos igények kielégítésére is.

Már a 2.0-s verzió is preemptív multitaszkkal rendelkezett és 32 bites rendszer volt, de az igazi sikert a 3.0-s, ú.n. OS/2 Warp hozta meg, amely a maga idejében bámulatra méltó képességekkel rendelkezett: amellett hogy már egy 4 MByte RAM-mal rendelkező 386-os elég volt a futtatásához (csak összehasonlításképp: azidőben például a Microsoft Windows NT-nek 16 MByte-ra volt szüksége ugyanehhez), képes volt arra, hogy egyszerre futtasson OS/2, DOS és Windows 3.1-es programokat egymás mellett.

Habár később megjelent a Microsoft Windows NT 4.0, az OS/2 stabilabb volt, és a DOS-os programok sokkal nagyobb százalékát volt képes futtatni mint a vetélytársai, így hamar elterjedt, főleg a BBS-eket üzemeltetők körében, mivel amellett, hogy a BBS üzemelt, a gépet másra is lehetett használni.

Az elmúlt évben jelent meg az OS/2 legújabb, 4.51-es verziószámot viselő változata, mely komoly változást hozott az OS/2 felhasználók életében, ugyanis végre sikerült egy kis cégnek megállapodni az IBM-mel, és a saját neve alatt adhatja el a licenszelt OS/2-t. Eddig az OS/2 felhasználók az IBM-re voltak hagyatkozva, amely köztudottan keveset törődik az otthoni felhasználókkal, inkább csak a nagy cégek kívánságait lesi, valljuk be, érthető módon. Most azonban egy sokkal kisebb cég, a Serenity Systems Inc. felhasználva a „nagyoknál” már bevált OS/2 technológiát, egy kifejezetten az otthoni felhasználók és a kis cégek gépeit megcélzó operációs rendszert dobott piacra eComStation néven.

Mivel az eComStation maga is OS/2, csak sok-sok kiegészítő programmal és néhány, főleg vizuális és kényelmi változtatással, ez a változat is megőrizte elődei jó tulajdonságait, és kiválóan képes DOS-os alkalmazásokat (sőt, különböző DOS-okat!) futtatni. Megörökölte azonban azok egy kényelmetlen megszorítását is, melyre a DOS ablakokban figyelhetünk fel.

1.2. Probléma a hosszú fájlnevekkel

Amikor az OS/2 2.0-t és vele a DOS emulációt megírták, még nem léteztek a PC-s világban olyan operációs rendszerek, melyek hosszú fájlneveket tudtak kezelni. Az OS/2 volt az első, amely ezt a szolgáltatást nyújtotta, amennyiben az (akkor még) új HPFS fájlrendszert választja a felhasználó. Mivel a lehető legnagyobb kompatibilitást szerették volna elérni a DOS emulációnál, és a DOS azidőtájt még nem ismerte a hosszú fájlneveket, a bonyodalmak elkerülése érdekében úgy döntöttek, hogy azokat a fájlokat és könyvtárakat amelyeknek neve nem fér bele a 8+3 alakba (azaz amelyeket a DOS egyébként sem tudna kezelni), egyszerűen eltüntetik a DOS elől, azok nem látszanak a DOS ablakból.

Ez a megoldás nagyon jól működött egészen addig, amíg meg nem jelentek az olyan fájlok, amelyek neve túl hosszú volt, de feldolgozásukra csak DOS-os alkalmazás volt elérhető. Ezeknél az egyetlen megoldás az volt, ha a fájlokat átnevezték valami rövidebb formába, amely már a DOS ablakban is látható. Ez azonban nem megoldható, ha például az ominózus fájl egy CD-n van, vagy éppen van belőle körülbelül 2000 darab.

Ilyen fájlok voltak például a kezdeti időkben az MP3 kiterjesztésű zenefájlok, melyeket csak DOS-os lejátszóval lehetett lejátszani amíg a natív OS/2-es MP3 lejátszóprogramok meg nem születtek, vagy például néhány videó formátum, melyre máig nincs stabil, megbízható OS/2-es lejátszóprogram, így a DOS-os, vagy a DOS emulációban futó Windows 3.1-es programokra van szükség, melyek viszont nem látják a túl hosszú nevű fájlokat.

Szükség volt hát valamiféle megoldásra, amellyel valami módon ezek a fájlok is elérhetővé válnak a DOS emuláció (Virtual DOS Machine, továbbiakban VDM) számára.

2. fejezet

Az OS/2 rendszer felépítése

Mielőtt a probléma megoldásához fognánk, tudnunk kell, hogyan folyik a kommunikácio a VDM és az OS/2 között, milyen lehetőségek vannak arra, hogy információt cseréljen egy DOS és egy OS/2 alkalmazás.

2.1. Fájlrendszer a VDM-ben és a Supervisor Call (SVC)

Amennyiben a felhasználó a beépített DOS emulációt választja, akkor az IBM PC-DOS egy módosított változata indul el (az MDOS), amely az alacsonyszintű fájlműveleteket az OS/2 kerneléhez irányítja át. A kernel dolgozza fel ezeket a kéréseket, és a kernel az, ami úgymond „megszűri" az eredményt, és csak a rövid fájlneveket szolgáltatja. Megvan viszont az az előnye ennek a módszernek, hogy ha egy bizonyos fájlrendszert ismer az OS/2, akkor azt a DOS ablak is elérheti és használhatja, még akkor is, ha azt az eredeti PC-DOS nem ismerte.

Nagyon érdekes, máig nem dokumentált dolog, hogy hogyan juttatja el kéréseit az OS/2 kerneléhez a beépített DOS. Van egy speciális Assembly utasítás, a *HLT*, amely normál esetben leállítja a processzor futását egészen addig, amíg valamiféle megszakítás nem érkezik. Nos, az OS/2 egy három byte-os kódsorozatot használ arra, hogy a VDM számára speciális kéréseket szolgálhasson ki. Ez a kódsorozat a következőképpen néz ki:

```
HLT
DB Utasításkód
DB NOT Utasításkód
```

Amikor a VDM a *HLT* végrehajtásához ér, azt az OS/2 kernel elfogja, és megvizsgálja, vajon neki szóló kérésről van-e szó. Ha a *HLT*-ot követő két byte egymás negáltja, akkor a kernel ezen három byte helyett egy, az `Utasításkód` által meghatározott dolgot végez el, például létrehoz egy könyvtárat, olvas egy fájlból, vagy akár bezárja a hívó DOS ablakot.

Ezt a hívási módot nevezik *Supervisor Call*-nak, vagy röviden SVC-nek. Nagyon sok ilyen hívás létezik, melyek legtöbbje a VDM-ek fájl- és könyvtárkezelésével foglalkozik.

2.2. Eszközmeghajtók

Az OS/2 a DOS-hoz hasonlóan eszközmeghajtó programokat használ arra, hogy a különféle hardverekkel kommunikáljon, illetve hogy az alap operációs rendszer funkcióit bővítse. Két fő eszközmeghajtótípus létezik, a Physical Device Driver és a Virtual Device Driver.

2.2.1. A Physical Device Driver (PDD)

A PDD-k a legtöbb esetben az operációs rendszer és alkalmazásainak eszközfüggetlen I/O kéréseit fordítják le az adott eszköz számára érthető kérésekké. Ilyen eszközmeghajtó például a CD-ROM, az egér vagy a videókártya meghajtóprogramja. A PDD a rendszer indításakor töltődik be, és Ring 3-ban (a 80386-os Intel processzorok legkevésbé privilégizált szintjén) inicializálódik, majd Ring 0-ban fut tovább (a legprivilégizáltabb szinten). Két fő típusa létezik:

- A **Character Device Driver** általában valamilyen byte- vagy stream-orientált eszközt támogat. Az eszköznek neve van, mint például *SCREEN$*, *KBD$*, *LPT1* vagy *COM2*. Az alkalmazások úgy kommunikálhatnak az eszközzel, ha a megadott eszköznévre hivatkozva meghívják a `DosOpen`, majd a `DosRead`, `DosWrite` vagy a `DosDevIOCtl` API-t.

- A **Block Device Driver** általában valamilyen blokk- vagy rekord-orientált eszközt támogat. Az ilyen meghajtóknak nincs nevük, hanem egy vagy több meghajtó-betűjelet kapnak az OS/2 kernelétől, egyet minden egyes támogatott eszközhöz vagy egységhez.

2.2.2. A Virtual Device Driver (VDD)

A VDD feladata, hogy egy bizonyos hardvert vagy ROM BIOS-t virtualizáljon a DOS alkalmazások számára. Ezt a virtualizációt az I/O portok és/vagy az eszköz memóriájának emulálásával éri el. A Virutal Device Driverek 32 bites eszközmeghajtók, melyek a processzor legprivilégizáltabb szintjén futnak (Ring 0). Ahhoz, hogy egy bizonyos hardverfüggetlenséget érhessen el, a Virtual Device Driver gyakran egy Physical Device Driverrel kommunikál annak érdekében, hogy a hardvert elérje.

Röviden összefoglalva tehát a Virtual Device Driver egy olyan eszközmeghajtó, ami igazából nem is eszközmeghajtó, hiszen (általában) nem kommunikál közvetlenül fizikai eszközökkel, csak más eszközmeghajtó programokkal (legyen az PDD

vagy egy másik VDD), de a VDM-ben futó DOS számára annak látszik, hiszen rajta keresztül érhető el valamilyen hardver. Innen hát a *Virtual* név.

Az OS/2 annak köszönheti kitűnő DOS-kompatibilitását, illetve azt, hogy szinte minden DOS-os program futtatható a DOS ablakában, hogy minden főbb hardverhez létezik egy ilyen VDD, így a VDM-ben futó DOS-t nagyszerűen be tudja csapni amellett, hogy ezeket a hardvereket mégiscsak a saját kezében tartja. Íme néhány az alapállapotban betöltődő VDD-k közül:

- VBIOS.SYS - A BIOS virtualizálására
- VCDROM.SYS - A CD-ROM elérhetővé tételére. Ezáltal amelyik CD-ROM-ot az OS/2 kezelni tudja, az a VDM-ben is látszani fog.
- VCOM.SYS - A soros portok virtualizálására
- VEMM.SYS - EMS memória szolgáltatása a VDM számára

2.3. Egyéb kiszolgáló programok

Az eszközmeghajtóknak nagyobb szabadságuk van, mint az általános alkalmazásoknak, hiszen Ring 0-ban futnak, így szabadon írhatnak és olvashatnak a portokról, szabadon elérhetik a memóriát, stb. Vannak azonban dolgok, amiket lehetetlen, vagy csak nagyon körülményes módon lehet megoldani az eszközmeghajtókból, de nagyon egyszerűek egy átlagos alkalmazás számára. Ilyen dolog például a fájlok kezelése, melyet se a PDD-kből se a VDD-kből nem lehet elvégezni.

Ezen problémák megoldására szokás speciális kiszolgáló programokat indítani, ú.n. *daemonokat*, melyek egyetlen feladata, hogy az adott meghajtóprogram ilyen irányú kéréseit lessék, és elvégezzék azokat a PDD vagy VDD helyett.

Az ilyen daemonok a következő módszert alkalmazzák arra, hogy a meghajtóprogram kérései elérhessenek hozzájuk:

- `DosOpen`-t vagy `DosOpenVDD`-t használva megnyitják a meghajtóprogramot
- `DosDevIOCtl` vagy `DosRequestVDD` segítségével „behívnak" a meghajtóba, mely ezáltal tudomásul veszi, hogy a kiszolgáló program megérkezett, készen áll. Az eszközmeghajtó blokkolja a program futását, azaz addig nem tér vissza, amíg valamit el nem akar végeztetni a daemonnal.
- Amennyiben az alkalmazás visszakapja a vezérlést, tudja, hogy valamit el kell végeznie az eszközmeghajtó számára, így megvizsgál valamilyen változót, osztott memóriatartományt vagy egyebet, melyben az eszközmeghajtó megüzente neki, hogy mit kell csinálnia.

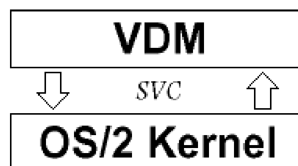
- Miután a kívánt dolgot elvégezte, az eredménnyel újra behív a meghajtóprogramba (mint a második pontban), amelynek ezzel nyugtázza a dolog elvégzését. Ilyen módon az alkalmazás újra blokkolt állapotba kerül, egészen a következő kérésig.

3. fejezet

A VLFN felépítése

Mindezen áttekintés után tehát lássunk hozzá a probléma megoldásához, tegyük valahogyan láthatóvá a hosszú fájlneveket a VDM-ekben! A megoldáshoz szükség lesz egy PDD-re, egy VDD-re, egy Daemon-ra, sőt, még egy DOS-os programot is kell írni.

Mint azt már említettem, a VDM eredetileg közvetlenül az OS/2 kernelt hívja meg ha valamilyen fájlműveletre van szüksége, amihez a Supervisor Call-t használja (lásd 3.1 ábra).



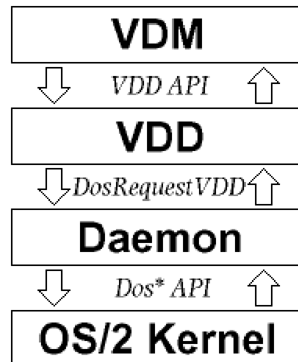
3.1. ábra. Eredeti útvonal

A célunk eléréséhez azonban arra van szükség, hogy ne közvetlenül az OS/2 kernel szolgálja ki a VDM-et, mert akkor az csak a rövid fájlneveket fogja megmutatni. Csakis az OS/2 alkalmazások látják és kezelik a hosszú fájlneveket, tehát a feladat adott: valami módon át kell irányítani a fájlműveleteket egy OS/2 alkalmazásnak, ami majd mindent elvégez a VDM helyett.

Sajnos a DOS alkalmazások nem tudnak közvetlenül az OS/2 alkalmazásokkal kommunikálni, csak nevesített pipe-okon keresztül, mivel azok a fájlrendszerben vannak implementálva, így ha egy ilyen pipe-ot létrehoz egy alkalmazás, az mindenhonnan látszik, mindenhonnan elérhető.

Megtehetnénk tehát, hogy futtatunk egy OS/2 alkalmazást, ami létrehoz egy nevesített pipe-ot, amin keresztül parancsokat kapna a DOS ablakoktól, és amin keresztül vissza is juttathatna adatokat. Ez azonban nem lenne optimális megoldás, hiszen például egy fájl olvasását is ennek az OS/2 alkalmazásnak kell elvégeznie a VDM helyett, és egy nagy fájlt átküldeni egy pipe-on elég lassú folyamat.

Szerencsére lehetőségünk van arra is, hogy egy VDD-t írjunk, amely úgynevezett *VDD API*-n keresztül elérhető a DOS ablakok számára, míg a `DosRequestVDD` függvény alkalmazásával elérhető az OS/2 alkalmazások számára is. Ilyen módon egy hidat képezhet a VDM és egy OS/2 alkalmazás közt, és a 3.2 ábrán látható módon felhasználhatnánk a probléma megoldására.



3.2. ábra. Új útvonal

Ezzel a módszerrel az adatok nagy sebességgel és egyszerre nagy darabokban áramolhatnak a VDM és a daemon alkalmazás közt, ami nem mondható el a nevesített pipe-os megoldásról. Míg ott 64 KByte adat átküldése számottevő időt venne igénybe, addig itt ez egy hívással megoldható, hiszen ezt a 64 KByte-ot a daemon egyszerre elküldheti a VDD-n keresztül.

Mivel valamiféle VDM-es alkalmazást fogunk írni, és az OS/2 világában már megszokottnak mondható, hogy a VDM-hez tartozó meghajtóprogramok a „V" (mint *Virtual*) betűvel kezdődnek, legyen a meghajtóprogram neve VLFN, a *Virtual Long Filename* rövidítéseként.

Mint az már ezen fejezet elején említésre került, a VLFN négy részből fog állni, melyek a következők:

- Szükség van egy, a VDM-ben futó **DOS-os alkalmazás**ra, mely az éppen futó MDOS-t módosítja a memóriában úgy, hogy az ne az OS/2 Kernel felé küldje kéréseit, hanem egy VDD-hez. Ezt egy egyszerű rezidens programmal könnyen el lehet végeztetni.

- Kell továbbá az előbb említett **VDD**, mely fogadja a VDM kéréseit, majd továbbítja azokat egy OS/2 alkalmazásnak, a daemonnak, illetve a daemon által szolgáltatott adatokat visszajuttatja a VDM-nek.

- Mindehhez szükség van egy memóriaterületre, amin keresztül a VDD és a daemon kommunikálhat, azaz egy olyan terület, amit mind a kettő elér. A VDD részéről nincs probléma, hiszen az Ring 0-ban fut, azaz szinte bármit elérhet

ha akar, illetve létre is hozhat magának egy ilyen munkaterületet. A problémát a daemon jelenti, akinek a memória-címtartományába ,,be kell lapozni" ezt a memóriaterületet, amit csak egy PDD tud elvégezni. Ehhez a munkához tehát szükség van egy **PDD**-re is.

- Végül, de nem utolsó sorban kell egy OS/2 alkalmazás, a **daemon**, amely fogadja a VDD (és végső soron a VDM) felől érkező kéréseket, végrehajtja azokat, és mindeközben a hosszú fájlnevekből valamilyen módon rövidet csinál, hogy azok az MDOS-ban is használhatóvá váljanak.

A következőkben lássuk ezeket a részeket egyenként!

3.1. A memóriát adó PDD (VLFNMEM.SYS)

Kezdjük talán a legegyszerűbb résszel, a PDD-vel, amelynek egyetlen dolga, hogy lefoglaljon valamennyi memóriát az adatcsere számára, és ennek a memóriaterületnek a címét odaadja a hívó alkalmazásnak vagy VDD-nek.

3.1.1. A fejléc és a változók

A PDD-k az OS/2-ben hasonló felépítésűek, mint a DOS-os eszközmeghajtó programok. Kötelezően egy adatszegmenssel kell kezdődniük, melyben az első struktúra egy meghatározott fejléc kell hogy legyen. Ez az adatszegmens a mi esetünkben a következőképpen néz ki Assembly nyelven:

```
_DATA       segment word public 'DATA'

; ------ Device Driver Header: ------

devhdr      dd    -1                    ; Chain to next DD
devatt      dw    DAW_CHAR or DAW_LEVEL ; Attributes
strat       dw    offset strategy       ; Strategy procedure
idc         dw    0                     ; No IDC
devname     db    'VLFNMEM$'            ; Device name
reserved    db    8 DUP (0)

; ---- Private variables ----

devhlp      dd    ?             ; DEVHLP entry
MEM128K     dd    ?             ; Address of 128K shared area
PDD_DevName db    'VLFNMEM$',0

; ---- End of Data segment, remaining in memory! ------
```

```
startmsg     db    LF,'VLFN Memory-support device driver v1.0',
                   CR,LF
startlen     equ   $-startmsg

initmsg     db    'Driver successfully installed',CR,LF
initlen     equ   $-initmsg

failmsg     db    'Driver failed to install!',CR,LF
faillen     equ   $-failmsg

_DATA       ends
```

Az első duplaszót, a *devhdr*-t maga az operációs rendszer fogja kitölteni, és a következő betöltött eszközmeghajtó fejlécére fog mutatni. A *devatt* az eszközmeghajtónk típusát adja meg: egy Character Device Driver lesz (lásd 2.2.1 a 7. oldalon), mely az újabb szintű rendszerhívásokat is támogatja.

Minden PDD rendelkezhet egy ú.n. *Strategy* és egy *IDC* belépési ponttal. A *Strategy*-n keresztül kommunikál az operációs rendszer a meghajtóprogrammal, míg az *IDC* szolgál az eszközmeghajtók közötti közvetlen kommunikációra. A mi programunktól más PDD-k nem fognak kérni semmit, így az *IDC*-t üresen hagyhatjuk.

Szükség van még az eszköz nevének megadására, amellyel az alkalmazások majd erre a meghajtóprogramra hivatkozhatnak, ez került a meghajtóprogram fejlécének következő helyére.

A további adatok különböző globális változók, róluk majd a felhasználás helyén esik szó.

3.1.2. A kódszegmens

Mint azt az előbbiekben láthattuk, szükség van egy Strategy eljárásra, amin keresztül az operációs rendszer kommunikálhat a meghajtóprogrammal. Nagyon sok kérés és értesítés fut be ebbe az eljárásba, de mi csak kettőt használunk fel, az **INIT** parancsot és az **IOCTL** parancsot. Az **INIT** parancsot kapja a Strategy rutin, amikor a meghajtóprogram betöltődött a memóriába, és a futtatásra kész. Ezt használjuk fel arra, hogy inicializáljuk a programot. Az **IOCTL** parancs akkor érkezik, ha egy alkalmazás a *DosDevIOCTL* API használatával valami szolgáltatást kér az eszközmeghajtónktól. Minden más parancsra az „ismeretlen parancs” hibakóddal (8103h) válaszolunk. A mi Strategy rutinunk, amely ezeket végzi el, tehát így néz ki:

```
strategy    proc far
```

```
            mov     di,es:[bx+2]      ; get command
            cmp     di,INIT_CMD       ; init command?
            jne     ioctl             ;
            call    init              ; yes, do initialize
            jmp     short sdone       ; and go to end

ioctl:      cmp     di,IOCTL_CMD      ; no. Is it IOCTL cmd?
            jne     short serr        ;
            call    IOCTL_Proc        ; yes, do the job
            jmp     short sdone       ; and go to end

serr:       mov     ax,8103h          ; not init, not IOCTL...
                                      ; we don't support more.
                                      ; bad command:
                                      ; return error
sdone:      mov     es:[bx+3],ax      ; store error code
            ret
strategy    endp
```

Mint az látható, az inicializálást az *init* rutin végzi, míg az IOCTL hívásokat az *IOCTL_Proc* kezeli. Az inicializáló rutin nem sok különös dolgot végez: amellett, hogy üzeneteket ír a képernyőre, a kernel úgynevezett *DevHlp* függvényei használatával lefoglal 128 KByte memóriát, majd regisztrálja magát, így a VDD-k számára is hívható lesz. Ezeket a rutin következő kódrészlete végzi el:

```
 xor     eax,eax                    ; Try to allocate 128K
 mov     MEM128K,eax                ; for shared memory

 mov     ecx,_128K                  ; Size
 mov     edi,-1                     ; Physical address
 mov     eax,0                      ; Alloc. flags
 mov     dl,DevHlp_VMAlloc
 call    [DevHlp]
 jc      short fail

 mov     MEM128K,eax                ; Save the linear address

 push    es
 lea     si, PDD_DevName            ; Register ourselves to be
 mov     di, cs                     ; callable from eg. a VDD
 mov     es, di                     ; DS:SI = Ptr to name of PDD
 lea     di, VDDEntry               ; ES:DI = Pointer to IDC
 mov     dl,DevHlp_RegisterPDD
```

```
call    [DevHlp]                    ; if the function fails,
pop     es                          ; system halt occurs.
```

Az *IOCTL_Proc*-on keresztül tehát az alkalmazások, míg az előző kódrészlet eredményeként a *VDDEntry*-n keresztül a VDD-k számára is elérhető a meghajtóprogram, és azon keresztül a lefoglalt 128 KByte memóriaterület címe.

Lássuk először mi történik az IOCTL hívások esetén! Feltételezzük, hogy amennyiben egy OS/2 alkalmazás meghív minket, akkor az a memória címére kíváncsi, azt szeretné visszakapni egy változójába. Az első dolgunk tehát az, hogy leellenőrizzük, olyan címre várja-e az alkalmazás az eredményt, amire van joga írni, vagy nem. Ezt a *DevHlp_VerifyAccess* segítségével tehetjük meg a következő módon:

```
mov     di, word ptr es:[bx+GIODataPack]    ; (di) = offset
mov     ax, word ptr es:[bx+GIODataPack+2]  ; (ax) = seg/sel
mov     cx, 4                       ; length: 4 byte
mov     dh, 1                       ; read/write
mov     dl, DevHlp_VerifyAccess

call    [DevHlp]                    ; Verify if the caller can
                                    ; access the area where it
                                    ; waits for our result.

jc      short error                 ; No, quit with error.
```

Amennyiben az alkalmazás megfelelő címre várja az eredményt, a 128 KByte-os memóriaterületet be kell lapozni az alkalmazás címtartományába, azaz elérhetővé kell tenni a számára azt, majd az így kapott (csak az alkalmazás számára használható) címet vissza kell adni neki az előbb már leellenőrzött memóriacímen. Ezt végzi a következő kódrészlet:

```
les     di, es:[bx+GIODataPack]     ; Target = User Buffer
movzx   edi,di
lea     esi, MEM128K                ; Source = MEM128K

mov     ebx, dword ptr ds:[esi]
mov     ecx,_128K                   ; Map 128K memory
mov     eax,1                       ; with R/W access
mov     dl,DevHlp_VMGlobalToProcess
call    [DevHlp]                    ; please...

stosd                               ; Store the result (EAX)
                                    ; in the caller's variable

mov     ax,0100h                    ; No error, Done.
```

A hívó VDD számára sokkal könnyebb a címet átadni, hiszen az is Ring 0-ban fut, így nincs szükség különféle jogok ellenőrzésére, sőt, a memóriaterületet sem kell belapozni a hívó címtartományába, hiszen a lineáris címet a VDD közvetlenül kezelni tudja. A *VDDEntry* rutin tehát egyszerűen átadja a *MEM128K* változó tartalmát a hívó VDD számára. Egyszerűsége miatt ennek közlését most kihagyom.

3.2. A híd VDD (VLFN.SYS)

3.2.1. A szegmensek

A VDD-k formájukat tekintve speciális DLL-ek, melyek megadott sorrendben, megadott típusú szegmensekből állnak. Ezeket a definíciós fájlban kell megadni, ami a VDD-k esetében általában így néz ki:

```
VIRTUAL DEVICE
PROTMODE

SEGMENTS
  _TEXT        CLASS 'CODE'      SHARED    NONDISCARDABLE  RESIDENT
  CINIT_TEXT   CLASS 'CODE'      SHARED    DISCARDABLE     RESIDENT
  CSWAP_TEXT   CLASS 'CODE'      SHARED    NONDISCARDABLE
  CINIT_DATA   CLASS 'CINITDATA' SHARED    DISCARDABLE     RESIDENT
  CSWAP_DATA   CLASS 'CSWAPDATA' SHARED    NONDISCARDABLE
  MVDMINSTDATA CLASS 'MIDATA'    NONSHARED NONDISCARDABLE  RESIDENT
  SWAPINSTDATA CLASS 'SIDATA'    NONSHARED NONDISCARDABLE
  _DATA        CLASS 'DATA'      SHARED    NONDISCARDABLE  RESIDENT
```

Mint látható, nyolc különféle szegmensből áll a VDD, melyek a következők:

- *_TEXT*: Ez az a szegmens, amelybe azokat a kódrészleteket kell elhelyezni, amelyeknek folyamatosan a memóriában kell lenniük, nem szabad, hogy a tárcserefájlba kilapozódjanak.
- *CINIT_TEXT*: Ez a szegmens a VDD inicializálása után ,,el lesz dobva”, azaz nem marad tovább a memóriában (lásd DISCARDABLE attribútum), azaz ide érdemes azokat a kódrészleteket helyezni, amelyek csak az inicializáláshoz szükségesek.
- *CSWAP_TEXT*: Ide helyezendő az a kód, amely szükség esetén kilapozható a tárcserefájlba.
- *CINIT_DATA*: Ugyanúgy, mint a *CINIT_TEXT* esetén, ez a szegmens is el lesz dobva az inicializálás után. A különbség az, hogy ez nem kódszegmens, hanem adatszegmens.

- *CSWAP_DATA*: Ez egy olyan adatszegmens, amely kilapozható a tárcserefájlba, és minden egyes VDM számára elérhető adatokat tárol.
- *MVDMINSTDATA*: Ez az adatszegmens nem lapozható ki a memóriából, azaz ami ide kerül, az folyamatosan a memóriában lesz. Egyedisége, hogy ebből a szegmensből új készül minden egyes VDM számára, azaz a különböző, VDM-enként egyedi adatok tárolására kiválóan alkalmas.
- *SWAPINSTDATA*: Ez az *MVDMINSTDATA*-hoz hasonlóan minden egyes VDM számára egyedi, de szükség esetén a tárcserefájlba kilapozható adatszegmens.
- *_DATA*: Ez egy nem-kilapozható (RESIDENT), minden VDM számára elérhető (SHARED) adatszegmens. Az általános változók tárolására használatos.

3.2.2. A VDD inicializálása

Mivel a VDD-k felépítése a DLL-ekhez hasonló, így itt is meg lehet adni egy eljárást, amely az inicializáláskor fog lefutni, és amelyet természetesen a *CINIT_TEXT* szegmensben kell elhelyezni. A mi eljárásunk, melynek neve *VLFN_Init*, a következőket végzi el:

- Bejegyzi magát a különféle VDM események kezelésére, mint például új VDM létrehozása, régi megszűnése, VDM-beli program indítása illetve leállása.
- Megnyitja a VLFNMEM$-t (tulajdonképpen a VLFNMEM.SYS meghajtóprogramunkat), és lekérdezi a 128 KByte-os memóriaterület címét, elmenti azt későbbi használatra.
- Létrehoz néhány szemafort, melyeket a daemonnal való kommunikáció során használunk majd a szinkronizációhoz.
- Regisztrálja a VDD-t a rendszerben, így az OS/2-es alkalmazások is el tujják érni a VDD-t, például a daemon is képes lesz behívni a VDD-be.

Mindezt különféle kernel helperekkel, VDH-kkal éri el, például az események kezelésének bejegyzésére a *VDHInstallUserHook* használható:

```
mov  eax,VDM_CREATE        ; Hook for VDM Creation Event
mov  ebx,offset FLAT:VLFN_VDMCreation    ; Proc. to call

push eax
push ebx
Call VDHInstallUserHook

cmp eax,VDH_FAILURE        ; If fails, then exit
je Init_Fail
```

3.2.3. VDM események kezelése

Négy olyan esemény van, amely bekövetkeztekor valamit csinálni szeretnénk. Ezekre az eseményekre a VDD inicializálásakor kell regisztráltatni az eseménykezelő eljárásokat a fentebb bemutatott *VDHInstallUserHook* segítségével. A VLFN a következő négy eseményt kezeli le:

- **VDM_CREATE** Ez akkor történik, amikor például egy DOS ablakot nyit a felhasználó. Ekkor inicializálni kell a VDM-enként egyedi adatstruktúrákat, illetve létre kell hozni azt a ,,kaput" a VDM-ben, amin keresztül az majd kommunikálni tud velünk.
- **VDM_TERMINATE** Ez akkor történik, amikor valami miatt (például a felhasználó kérésére) a VDM bezáródik. Ekkor kell elvégezni a ,,nagytakarítást", legalábbis annak azt a részét, ami ránk tartozik. Ilyenkor kell bezárni a VDM által nyitva hagyott fájlokat illetve félbehagyott fájlkereséseket.
- **VDM_PDB_CHANGE** Amennyiben a VDM-ben egy DOS-os alkalmazás elindul, akkor megváltozik a *Program Description Block*, és erről kapunk egy értesítést. Ezt arra használjuk fel, hogy letároljuk az új program azonosítóját, az új **PDB**-t, amit minden megnyitott fájl mellé le fogunk tárolni, így minden fájlról tudni fogjuk, melyik alkalmazás nyitotta meg.
- **VDM_PDB_DESTROY** Ez az esemény jelzi, ha egy DOS-os alkalmazás befejeződött. Ekkor végignézzük a nyitott fájlokat és fájl-kereséseket, és a hozzájuk rendelt **PDB** alapján bezárjuk azokat, amelyeket az éppen befejeződött alkalmazás esetleg nyitva hagyott. Olyan tehát ez mint a *VDM_TERMINATE*, csak itt nem kell mindent bezárnunk.

Ezen események kezelését természetesen külön-külön eljárások végzik, melyek a *CSWAP_TEXT* szegmensben helyezkednek el, mivel ezek olyan eljárások, melyek megszakításból sohasem lesznek meghívva, így szükség esetén nyugodtan kilapozhatóak a fizikai memóriából (lásd 16. oldal).

3.2.4. Információcsere a daemonnal

A VLFN.SYS mint híd egyik oldalán a szolgáltatásokat kérő VDM áll, míg a másik oldalon a kiszolgáló daemon. Lássuk, hogyan tudjuk a VDM-től érkező kéréseket eljuttatni a daemonhoz!

Mint azt már a 2.3 részben (a 8. oldalon) láthattuk, a kiszolgáló alkalmazás egy `DosRequestVDD` hívással behív a VDD-be, és itt blokkolt állapotban marad egészen addig, amíg valamit el nem kell végeznie. Ehhez szükség van arra, hogy a VDD regisztrálva legyen a rendszer számára, amit el is végeztünk a VLFN inicializálásakor (lásd a 17. oldalon).

A regisztrációkor meg kell adni egy eljárásnevet, mely akkor fog meghívódni, amikor egy OS/2 alkalmazás behív a VDD-be. A VLFN esetében az eljárás neve, mely ezt végzi a *VLFN_Communicate_With_Daemon*. Ez különféle paramétereket (szám szerint hat darabot) kap illetve kaphat, amin keresztül az alkalmazás megmondhatja, hogy mit szeretne. A mi esetünkben azonban csak egy valamit akarhat a behívó alkalmazás: várni, amíg valamit csinálni kell. Így mi a hat paramétert teljes mértékben figyelmen kívül fogjuk hagyni, ezáltal is leegyszerűsítve az eljárást. Minden egyéb kommunikáció a PDD által lefoglalt 128 KByte-os memóriaterületen keresztül folyik.

Ennek az eljárásnak tehát csak annyit kell csinálnia, hogy valahogyan ,,szól" a VDD többi részének, hogy a daemon megérkezett (ezáltal valószínűleg kiszolgált egy előző kérést!), és addig vár, amíg újabb kérés nem érkezik. Az ilyen várakozásokra nagyszerűen használhatók a szemaforok. A VDD inicializálásakor már létrehoztunk néhány szemafort, ezek közül kettő itt kerül felhasználásra.

Ezek a szemaforok úgynevezett event– vagy ,,*esemény-szemaforok*", azaz arra valók, hogy egy esemény bekövetkeztét jelezhessük vele. Az ilyen szemaforok kezelésére a VDD-k a következő segédfüggvényeket használhatják:

- A `VDHWaitEventSem` használatával egy megadott ideig (de akár végtelen ideig is) várakozhatunk egy szemaforra. Ez azt jelenti, hogy az aktuális szál futása addig felfüggesztődik, amíg a szemafor ,,tilos" állásban van, és rögtön újraindul, amint ,,szabad" állásba kerül.
- A `VDHResetEventSem` használatával a szemafor ,,tilos" állásba kapcsolható
- A `VDHPostEventSem` használatával a szemafor ,,szabad" állásba kapcsolható

Ezek után lássuk az eljárást, amely az előbb említett kommunikációt végzi!

```
VLFN_Communicate_With_Daemon proc near
;Parameters on stack:
;  scrgrp, ulfunct, nbin,pin,nbout,pout

  push ebp
  mov ebp,esp

  mov byte ptr bDaemonWaiting,1

  push hsemDataIsReady
  call VDHResetEventSem   ; Clear Data_Ready_Semaphore

  push hsemAnswerIsReady  ; Post Answer_Ready_Semaphore
  call VDHPostEventSem    ;
  push hsemDataIsReady    ; And wait for new job,
```

```
  push -1                  ; new data to be sent to the daemon
  call VDHWaitEventSem

  mov byte ptr bDaemonWaiting,0

  mov eax,0                ; rc = 0

  pop ebp
  ret 22                   ; Clear stack
VLFN_Communicate_With_Daemon endp
```

Mint az az eljárás közepéből látható, ha egy alkalmazás (feltehetően a daemon) behív a VLFN-be, akkor az jelzi az *AnswerIsReady* nevű szemaforon keresztül, hogy a válasz megérkezett a daemontól (ezáltal a VLFN azon részei, amik erre a válaszra vártak elindulhatnak), majd várakozik a *DataIsReady* nevű szemaforra, azaz arra, hogy valahol valaki elkészítsen egy kérést az osztott, 128 KByte-os memóriaterületen, majd jelezze a kérés elküldhetőségét a szemafor szabadra állításával.

A figyelmes olvasó észrevehette, hogy mindenek előtt ez utóbbi szemafort állítja tilosba az eljárás. Erre azért van szükség, hogy a szemafor biztosan tilos állapotban legyen akkor, amikor arra kerül a sor, hogy várakozni kell rá.

Van még a kódban egy *bDaemonWaiting* nevű változó. Ennek szerepe az, hogy jelezze, ha a daemon készen áll a kiszolgálásra. Ez a fentebbi szemaforokkal együtt a *CSWAP_DATA* szegmensben lett deklarálva, így a VLFN minden részéről látható, és csak egy példányban fog szerepelni, azaz nem lesz belőlük külön példány minden egyes VDM számára (a szegmens leírása megtalálható a 17. oldalon), hanem mindenki ugyanazon változót éri el.

3.2.5. Információcsere a VDM-mel

Miután láttuk a híd egyik oldalát, nézzük meg, milyen a másik oldal, hogyan jutnak el a kérések a VDM-ből a VDD-hez, és hogyan készülnek el a daemonnak szóló kérések az osztott memóriaterületen!

Mint az a 3.2.3-ban leírásra került a 18. oldalon, egy VDM létrehozásakor a VLFN egy *VDM_CREATE* eseményről kap üzenetet, melyben többek közt létrehoz egy ,,kaput", melyen keresztül az éppen létrehozott VDM képes lesz kommunikálni a VDD-vel. Ez a kapu az úgynevezett *VDD API*, melyet a `VDHRegisterAPI` függvénnyel regisztráltathat a VDD egy adott VDM számára. E függvény számára meg kell adni egy nevet, melyre majd egy DOS-os alkalmazás hivatkozhat ha velünk szeretne kommunikálni, és meg kell adni egy eljárás nevét, mely akkor kerül meghívásra ha a DOS-os alkalmazás el is kezdi a kommunikációt.

A VLFN esetében a név `VLFN` lesz, míg az eljárás, mely kiszolgálja a hívó DOS-os alkalmazást a *VLFN_Communicate_With_VDM*.

A dokumentációk szerint ez a kiszolgáló eljárás két paramétert kap, melyből számunkra a második tarthat érdeklődésre számot, hiszen az egy mutató, mely a hívó VDM regisztereit leíró rekordra (*Client Register Frame*, röviden *CRF*) mutat. Ezt felhasználva lehetőségünk nyílik egy olyan VDD API létrehozására, mely a hívó bizonyos regisztereitől függően más-és-más dolgot végez el. Erre szükség is lesz, hiszen sok különféle dolgot kell elvégezni a VDM számára.

A VLFN a hívó `AH` regiszterében várja, hogy mit kell csinálni, ami a gyakorlatban azt jelenti, hogy a VDM `AH` regisztere fogja tartalmazni annak az SVC-nek a kódját, melyet a kernel helyett a daemonnak kell elvégeznie. Arról hogy ez a kód az `AH` regiszterbe kerüljön majd a DOS-os alkalmazásunk fog gondoskodni.

Tudjuk azonban, hogy elég sok ilyen SVC létezik, így a hosszú programkóddal járó If-Then-Else-If-Then-Else-... konstrukció helyett az elegánsabb, és sokkal könnyebben bővíthető, módosítható illetve átlátható ugró-tábla került implementálásra. Ez egy olyan táblázat, melyben minden sor öt byte-os, ami tartalmazza a kódot (1 byte) és a hívandó eljárás címét (4 byte). A táblázat végét egy csupa -1-ből álló sor jelzi (lásd a 3.1. táblázatot).

SVC Byte (1 byte)	***Eljárás címe (4 byte)***
db 0FFh	dd offset VDM_Install_Check
db 0FEh	dd offset VDM_Log_This_State
db SVC_demDup	dd offset VDM_demDup
db SVC_demClose	dd offset VDM_demClose
...	...
db SVC_demLockOper	dd offset VDM_demLockOper
db -1	dd -1

3.1. táblázat. Ugró-tábla

A következő kódrészlet a hívó VDM `AH` regisztere alapján az ugró-táblából (melynek neve `VDM_Func_Table`) kikeresi a meghívandó eljárás címét, majd meghívja azt:

```
  mov     ebx,_pcrf            ; EBX = pointer to client regs
  mov     eax,[ebx].crf_eax    ; Get AX value

  mov esi,offset FLAT:VDM_Func_Table

SearchMoreFunctions:
  cmp dword ptr [esi+1],-1     ; Function to call is -1 ?
  je short NotOurJob           ; If so, it means End of Table.
  cmp byte ptr [esi],ah
  je short FunctionFound
  add esi,5
  jmp short SearchMoreFunctions
```

```
FunctionFound:
  inc esi
  mov esi,dword ptr [esi]
  call esi
  jnc short OurJobDone

NotOurJob:                      ; If STC (error) occured,
  xor eax,eax                   ; report it to VDM too.
  mov ax,0FFFFh
  mov [ebx].crf_eax,eax
  or [ebx].crf_eflag,1          ; Set Carry of VDM
  stc

OurJobDone:                     ; We mustn't manipulate the
  mov eax,[ebx].crf_eflag       ; EFlags of VDM directly,
  push eax                      ; just under the control of
  call VDHSetFlags              ; 8086 emulation, not to change
  mov eax,ebx                   ; I/O privilege and interrupt
  add eax,crf_eflag             ; flags etc...
  push eax                      ; Look at VDHSetFlags->Purpose
  call VDHGetFlags              ; for more information...
```

A meghívott eljárás feladata, hogy elvégezze az érdemi munkát. Amennyiben az eljárás a VDM regisztereit szeretné módosítani vagy felhasználni, azt megteheti, hiszen hívásakor az `EBX` regiszter a CRF-re mutat. Oda kell azonban figyelni arra, hogy amíg a VDM minden regiszterét szabadon változtathatjuk, az `eflags`-t a maga speciális jelentése miatt csak a DOS emuláció felügyelete alatt szabad módosítani, ezáltal biztosak lehetünk benne, hogy a VDM nem fogja lefagyasztani a gépet, illetve nem tud túlságosan nagy jogokat kapni.

Miután tehát az eljárás visszatért és esetleg a CRF `eflag` mezőjét is módosította, meghívjuk a `VDHSetFlags` helpert, hogy az állítsa be a VDM flag-jeit. Ezt a fenti kódrészlet `OurJobDone` cimke utáni része végzi.

3.2.6. A VDM kéréseinek kiszolgálása

Amennyiben a VDM valamilyen kéréssel fordul a VLFN-hez, a 3.1 táblázat alapján egy eljárásra kerül a vezérlés, mely valamilyen módon megpróbálja kiszolgálni azt. Lássunk most néhány példát ezek közül!

VDM_Install_Check

Szükség van egy függvényre, amin keresztül a VDM ellenőrizni tudja, hogy fut-e a VLFN minden része (a daemon is), illetve azok verziószáma megegyezik-e azzal,

amit vár. Erre való az `AH=0ffh` kódú VLFN hívás.

Ez egy egyszerű eljárás, mely csak annyit csinál, hogy ha a daemon fut, továbbítja neki a kérést. A daemon erre a VDM regisztereiben visszaadja a verziószámát. Amellett, hogy ez csak ennyit végez, az alapvető kommunikáció bemutatására nagyon alkalmas, hiszen szinte az egész eljárás feladata csak az, hogy egy az egyben továbbítsa a daemon felé a beérkező kérést. Lássuk először magát a kódot, majd a magyarázatot!

```
public VDM_Install_Check
VDM_Install_Check proc

  mov dword ptr [ebx].crf_ecx,eax

  cmp byte ptr bDaemonWaiting,0
  jne short VDM_IC_DaemonIsHere
  mov dword ptr [ebx].crf_eax,'NODM'
  or  [ebx].crf_eflag,1
  jmp short VDM_IC_SemaphoreIsNotOurs

 VDM_IC_DaemonIsHere:

  call Request_Daemon_Usage
  or eax,eax
  jz short VDM_IC_SemaphoreIsNotOurs

  call CopyCRFtoDataArea

  call Exchange_Data_With_Daemon

  call CopyDataAreatoCRF

  call Release_Daemon_Usage

 VDM_IC_SemaphoreIsNotOurs:
  clc                ; Clear carry bit (Consume interrupt)
  ret
VDM_Install_Check endp
```

Mindenek előtt a kód ellenőrzi, hogy fut-e a daemon, és ha nem, visszatér egy speciális hibakóddal az `EAX`-ban, és beállítja a kliens Carry flagjét is, jelezvén, hogy hiba történt. A daemon futásának ellenőrzésére a **bDaemonWaiting** változó értékét vizsgálja meg, mellyel már a 19. oldalon találkozhattunk a példakódban.

Ha úgy látja, hogy a daemon fut, megpróbál az osztott memóriaterületen egy kérést összeállítani neki, majd azt elküldeni. Ehhez először is biztosítani kell, hogy

ugyanebben az időben más (illetve a program más része) nem akar ugyanarra a memóriaterületre írni, hiszen az összezavarhatná a mi munkánkat. Ezt egy *MutEx* szemafor (*Mutual Exclusive*, azaz kölcsönösen kizáró) alkalmazásával tökéletesen meg lehet oldani, hiszen ezek pont arra valók, hogy egyszerre csak egy szerezhesse meg a szemafort, a többi igénylő addig várakozik, amíg sorra nem kerül.

A kód tehát meghív egy `Request_Daemon_Usage` nevű eljárást, mely megpróbálja megszerezni a *UseDaemon* nevű mutex szemafort, illetve maximum egy másodpercig vár arra, hogy megszerezze. Amennyiben egy másodpercig nem sikerül megszerezni, a kód az érdemi munka elvégzése nélkül kilép (a `VDM_IC_SemaphoreIsNotOurs` cimkére kerül át a vezérlés).

Ha megszerezte a szemafort, akkor nyugodtan használhatjuk az osztott memóriaterületet, hiszen biztosak lehetünk benne, hogy más nem használja velünk egy időben. Rámásoljuk tehát a VDM fontosabb regisztereit a memóriaterület elejére (innen fogja tudni a daemon, hogy mit is kell csinálnia). Ezek után a kérés kész, lehet szólni a daemonnak, hogy hajtsa végre azt. Az `Exchange_Data_With_Daemon` feladata az, hogy a *DataIsReady* esemény-szemafort beállítva jelezze ezt, és az *AnswerIsReady* szemaforra várva megvárja a válasz megérkeztét (ezekről a szemaforokról lásd még a 19 oldalt). Az ezt elvégző kód a következő:

```
push hsemAnswerIsReady
Call VDHResetEventSem

push hsemDataIsReady
Call VDHPostEventSem

push hsemAnswerIsReady
push -1                    ; Wait forever
Call VDHWaitEventSem
```

Ha ez a kód visszatér, akkor a daemon elvégezte a feladatát, és az osztott 128 KByte-os memóriaterület tartalmazza a választ, ami esetünkben nem más, mint a VDM daemon által módosított regiszterei. Vissza kell hát másolni a memóriaterületről a CRF-be a byte-okat, és elengedni a *UseDaemon* mutex szemafort, jelezve ezzel, hogy befejeztük a daemon és vele a memóriaterület használatát.

Így működik tehát egy egyszerű adatcsere a VDM és a daemon között. Lássunk ezek után egy kissé bonyolultabb feladatot, amikor már bonyolultabb feldolgozásra is szükség van!

VDM_demCreateDir

Amennyiben egy VDM-ben futó alkalmazás egy könyvtárat szeretne létrehozni, akkor kerül a vezérlés erre az eljárásra. A létrehozandó könyvtár nevére a VDM-beli `DS:DX` mutat.

Itt már nem elég egyszerűen továbbítani a kérést a daemon felé, hiszen az nem képes a `DS:DX` ismeretében a létrehozandó könyvtár nevét elérni, így nekünk kell az osztott memóriaterületre másolni azt. Az előző példát követve lássuk először a kódot, majd jöjjön a magyarázat!

```
public VDM_demCreateDir
VDM_demCreateDir proc

  call Request_Daemon_Usage
  or eax,eax
  jz short VDM_CD_SemaphoreIsNotOurs

  call CopyCRFtoDataArea

  push edx
  push esi
  mov dx,word ptr [ebx].crf_ds
  mov esi,[ebx].crf_edx         ; ds:(e)dx into ESI
  call VDMPtr2FlatPtr
  push ESI                      ; Check for valid rights
  push 0                        ; Max size = autodetect
  push VDHLM_READ
  call CheckAccessRights
  jnc short VDM_CD_RightsOK
  pop esi
  pop edx
  call Release_Daemon_Usage
  jmp VDM_Access_Violation
  VDM_CD_RightsOK:              ; Access rights OK, let's go
  call _Canonize_               ; Input: ESI-> AsciiZ
  call CopyCanonizedToDataArea
  pop esi
  pop edx

  call Exchange_Data_With_Daemon
  call CopyDataAreatoCRF
  call Release_Daemon_Usage
 VDM_CD_SemaphoreIsNotOurs:
  clc
  ret
VDM_demCreateDir endp
```

Mivel a daemonnal szeretnénk kommunikálni, az első dolog amit meghívunk itt

is a *Request_Daemon_Usage*. Ez az az eljárás, amely az *UseDaemon* nevű mutex szemafort megpróbálja megszerezni.

Amennyiben ez sikerült, a CRF regisztereit (mondhatni szokás szerint) az osztott memóriaterületre másoljuk a *CopyCRFtoDataArea* meghívásával. Innen fogja tudni a daemon is, hogy mi a teendője. Ez esetben azonban szükség van még a létrehozandó könyvtár nevére is. Ezt is az osztott memóriaterületre másoljuk majd a CRF után, ez viszont nem ennyire egyszerű dolog.

Az első teendőnk az, hogy a VDM-beli `DS:DX` mutatóból egy, a VDD számára használható mutatót hozzunk létre, vagy más szóval a `DS:DX` szegmentált címből lineáris (ú.n. *FLAT*) címet készítsünk. Ezt a konverziót végzi el a *VDMPtr2FlatPtr* eljárás, melynek megvalósítása egy kicsit később kerül bemutatásra a 3.2.7. fejezetben.

Mielőtt ezt a címet felhasználnánk, le kell ellenőrizni, hogy a hívó VDM-nek van-e joga azt a memóriaterületet elérni. Megeshet ugyanis, hogy egy hibás, védett módban futó DOS-os alkalmazás egy véletlenszerű `DS` értékkel (azaz hibás szelektorral) hívja meg a rutinunkat. Mivel ez a rutin egy VDD része, így ez Ring 0-ban futva bármilyen memóriaterülethez hozzáfér, ilyen módon egy hibás DOS-os alkalmazás akár az OS/2 kernelének részeit is olvashatná vagy legrosszabb esetben felülírhatná.

A cím felhasználhatóságának vizsgálatát a *CheckAccessRights* eljárás végzi. Amennyiben ennek eredménye az, hogy a VDM-nek nincs joga az adott memóriaterületet elérni, az eljárás elengedi a daemont, majd egy speciális kódra kerül a vezérlés, a *VDM_Access_Violation*-re, mely tipikusan az ilyen esetek lekezelésére lett kifejlesztve.

Természetesen normális esetben a vizsgálat eredménye az, hogy a cím használható, azaz remélhetőleg ott egy könyvtárnév szerepel. Megeshet viszont, hogy az alkalmazás azt mondja, létre kell hozni egy ,,`alkonyvtar`” nevű alkönyvtárat. Kérdés, hogy hol, melyik meghajtón, melyik könyvtárban?

Mivel a daemonhoz elméletileg egymás után több VDM felől is érkezhetnek kérések, így a VDD az, amely minden egyes VDM-hez letárolja az aktuális meghajtót, illetve az aktív könyvtárat minden egyes meghajtón. Ezt úgy tudja megtenni, hogy a meghajtó- és könyvtárváltó parancsok is rajta mennek keresztül.

Amennyiben kap egy valamilyen könyvtár- vagy fájlnevet, azt mindenekelőtt ,,kanonizálni” kell, azaz az aktuális meghajtó és az aktív könyvtárak ismeretében a relatív névből abszolút nevet csinálni. Ez magában foglalja a `..\` jellegű hivatkozások feldolgozását is. Erre jó a *_Canonize_* nevű eljárás, mely az ,,`alkonyvtar`”-ból például ,,`C:\konyvtar\ban\alkonyvtar`”-t csinál.

Ezt a kanonizált alakot már átadhatjuk a daemonnak, azaz az így kapott nevet az osztott memóriaterületre másolhatjuk. A `CopyCanonizedToDataArea` az eljárás, amely ezt a másolást elvégzi.

Mindezen előkészületek után a daemonnak szánt kérés összeállt, az előbbiekben már ismertetett módon elvégeztethetjük azt vele.

VDM_demOpen

Az előbbiekben már láttuk, hogyan megy keresztül a daemonhoz egy olyan kérés, amelynél csak a regiszterek értékei számítanak. Láthattunk olyat is, amelynél a regiszterek értékein kívül egy nevet is át kellett adni a daemonnak. Lássuk most hogyan működik a fájl megnyitás művelete, ahol az előbbieken kívül már a nyitott fájlok táblázatát is kezelni kell! Az eljárás hosszát figyelembe véve a magyarázatot a kód különböző részei közé beszúrva helyeztem el.

```
public VDM_demOpen
VDM_demOpen proc

  push edi
  mov edi,offset FLAT:OpenFileTable
  call Search_For_Empty_Table_Entry
  pop edi
  cmp eax,-1
  jne short VDM_O_There_Is_Handle
  mov [ebx].crf_eax,4        ; 4 = error_Too_Many_Open_Files
  or  [ebx].crf_eflag,1      ; set Carry Flag
  jmp VDM_O_Exit
 VDM_O_There_Is_Handle:

  mov dword ptr ulTempHandle,eax
  call Request_Daemon_Usage
  or eax,eax
  jz VDM_O_SemaphoreIsNotOurs

  call CopyCRFtoDataArea
```

Mint az látható, a kód most nem a daemon megszerzésével kezdődik, hiszen ha a nyitott fájlok táblázatában már nincs több hely egy újabb fájlnak, akkor felesleges a daemonhoz fordulni, hanem rögtön vissza is térhetünk a 4-es hibával, melynek jelentése: túl sok nyitott fájl.

A táblázatban a keresést a `Search_For_Empty_Table_Entry` végzi, melynek e-gyetlen paramétere a táblázat címe. Ilyen módon ugyanez az eljárás használható a nyitott keresések táblázatában való keresésre is.

Amennyiben van még hely a táblázatban, az `ulTempHandle` nevű változóban letároljuk ezt a helyet, majd a már szokásosnak mondható módon a daemon használatát elkezdjük.

A következő lépések az előző példából már ismerősek lehetnek, hiszen ugyanazt a feladatot látják el. A hívó VDM `DS:SI` regiszterpárosa a megnyitandó fájl nevére mutat, amit a mutató-konverzió és a kötelező ellenőrzések után kanonizált formában bemásolunk az osztott memóriaterületre:

```
  push edx
  push esi
  mov dx,word ptr [ebx].crf_ds  ; DX <- segment/selector
  mov esi,[ebx].crf_esi         ; ds:(e)si into ESI
  call VDMPtr2FlatPtr
  push ESI                      ; Check for valid rights
  push 0                        ; Max size = autodetect
  push VDHLM_READ
  call CheckAccessRights
  jnc short VDM_O_RightsOK
  pop esi
  pop edx
  call Release_Daemon_Usage
  jmp VDM_Access_Violation

VDM_O_RightsOK:        ; Access rights OK, let's go
  call _Canonize_        ; Input: ESI-> AsciiZ
  call CopyCanonizedToDataArea
  pop esi
  pop edx

  Call Exchange_Data_With_Daemon
  call CopyDataAreatoCRF
  Call Release_Daemon_Usage
```

A lényeges különbség ezek után kezdődik. Meg kell vizsgálni, hogy a daemon meg tudta-e nyitni a kívánt fájlt vagy nem! Az egyszerűbb eset, ha nem tudta, hiszen akkor már beállította a hibakódot is, és csak vissza kell térni a módosított regiszterekkel a hívó VDM-hez.

```
  test [ebx].crf_eflag,1        ; If CF set, error occured!
  jnz short VDM_O_Exit
```

Ha sikerült a fájlt megnyitni, akkor ezt regisztrálni kell a nyitott fájl-ok táblázatában. Az `ulTempHandle`-ben már letároltuk a táblázat azon pozícióját, amit felhasználhatunk erre, így ezt könnyű megtenni. A táblázat minden egyes sorában négy byte-ot kell tárolnunk, melyek a következők:

- A daemon által szolgáltatott handle [1 byte]
- A megnyitások száma (mely a fájlszám-duplikációval növekedhet) [1 byte]
- A megnyitó program azonosítója, azaz az aktuális PDB [2 byte]

A táblázat kitöltése után még egy tennivaló van. A VDM-nek nem a daemon által adott fájl-azonosítót adjuk tovább, hanem egy olyan értéket, amelyről később biztosan meg tudjuk mondani, hogy mi szolgáltattuk-e azt. Ezt úgy tehetjük meg, hogy bebiztosítjuk, hogy az általunk szolgáltatott azonosítók mindig egy adott érték fölöttiek lesznek. Így ha egy olyan fájlt kellene bezárni amit nem mi nyitottunk meg, azt azonnal tudni fogjuk, és annak bezárását az eredeti eljárásra bízhatjuk.

Lássuk hát, hogyan lehet kitölteni a táblázatot, és kicserélni a fájl-azonosítót egy speciálisra!

```
  mov edi,ulTempHandle
  mov edx,edi                     ; edx=ulTempHandle
  shl edi,2
  add edi,offset FLAT:OpenFileTable
  ; edi=OpenFileTable[ulTempHandle]

  mov ax,word ptr [ebx].crf_eax   ; load Handle and
                                  ; OpenNum (should be 0)
  mov word ptr [edi],ax           ; Store
  mov ax,word ptr ipdbCurrent     ; Load current PDB
  mov word ptr [edi+2],ax         ; Store

  add edx,VLFN_Handle_Base        ; Create unique handle, and
  mov [ebx].crf_eax,edx           ; send our handle to VDM

 VDM_O_Exit:
 VDM_O_SemaphoreIsNotOurs:
  clc
  ret
VDM_demOpen endp
```

3.2.7. Egyéb segédeljárások

Az előbbi részből megismerhettük a legfontosabb eljárásokat, megtudhattuk, milyen elven működik a VLFN VDD része. Ezek az eljárások használtak néhány más eljárást, melyekről akkor nem esett szó. Következzen hát most ezeknek a bemutatása!

VDMPtr2FlatPtr

Talán még emlékszünk rá, hogy a VDM-től kapott szegmentált címet először lineáris címmé kell alakítani ahhoz, hogy felhasználhassuk. Ezt végzi el ez az eljárás. A kód a `DX:ESI` regiszterpárosban várja a VDM-beli szegmentált címet, és az `ESI`-ben szolgáltatja az átalakított, lineáris címet.

A cím átalakításához mindenekelőtt tudni kell, hogy a VDM-beli program védett módban fut, vagy valós módban, hiszen a védett módú szegmensértékek (szelektorok) más módon határozzák meg a memóriatartomány címét mint a valós módúak.

Minden VDM-hez tartozik egy állapotjelző duplaszó, az _flVDMStatus, melyből többek közt ezt is megtudhatjuk:

```
public VDMPtr2FlatPtr
VDMPtr2FlatPtr proc
  push eax
  push edx
  test _flVdmStatus,VDM_STATUS_VPM_EXEC
  jz   short VP2FP_not_protect_mode_app
```

Amennyiben védett módban fut a DOS-os alkalmazás, akkor a DX regiszter tartalma egy szelektor, melynek meg kell tudni a bázisát, azaz azt, hogy ez a szelektor melyik memóriatartományra mutat. Ezt a VDHGetSelBase helper segítségével gyorsan és egyszerűen megtudhatjuk, az eredmény az ulVP2FPTemp változóba kerül:

```
  and  edx,0FFFFh           ; Selector is only 16bits long
  push edx
  push offset FLAT:ulVP2FPTemp
  Call VDHGetSelBase
```

Miután a báziscímet megtudtuk, hozzá kell adni az offszetet. Az offset viszont lehet, hogy 16 bites, és az is lehet, hogy 32 bites, attól függően, hogy 16 vagy 32 bites védett módú program fut. Ennek eldöntésére újra az _flVDMStatus-hoz kell fordulni:

```
  test _flVdmStatus,VDM_STATUS_VPM_32
  jnz   short VP2FP_32bit_app
  movzx esi,si       ; 16 bits protected mode
                     ; (Using SI instead of ESI)
VP2FP_32bit_app:
  add   esi,dword ptr ulVP2FPTemp
  jmp   short VP2FP_got_pointer
```

A fenti kód végeredményeként ESI-ben a lineáris cím fog szerepelni. Sokkal egyszerűbb a tennivaló akkor, ha a VDM-beli alkalmazás valós módban fut. Ez esetben ugyanis a szegmensek 16 byte-os átfedéssel követik egymást, azaz a lineáris cím a $Cim = Szegmens * 16 + Offszet$ képlettel egyszerűen kiszámítható:

```
VP2FP_not_protect_mode_app:
  movzx   esi,si    ; Use 16 bit offset! (for RealMode calls)
  movzx   edx,dx    ; Ptr=Segment*16+Offset
```

```
  shl     edx,4
  add     esi,edx

VP2FP_got_pointer:
  pop edx
  pop eax
  ret
VDMPtr2FlatPtr endp
```

CheckAccessRights

Megelőzendő, hogy a VDM a rendszer más részeiben kárt okozzon, illetve hogy illetéktelen memóriatartományokhoz hozzáférjen, minden esetben szükséges a VDM által szolgáltatott memóriacím ellenőrzése. Ha a VDM az adott címen szolgáltat valamilyen adatot, akkor a memóriacím olvashatóságát kell vizsgálni, míg ha oda vár valamilyen eredményt, akkor annak írhatóságát.

A `CheckAccessRights` úgy lett megírva, hogy a veremben várja paramétereit: az ellenőrizendő lineáris címet, a memóriatartomány méretét, illetve a kívánt jogokat. Sokszor előfordul viszont, hogy a memóriatartomány mérete nem ismert, hanem egy nulla byte jelzi annak végét (pl. létrehozandó könyvtár neve). Ekkor a második paramétert nullára állítva az eljárás automatikusan megkeresi a memóriaterület méretét.

Amennyiben a méret nem adott, meg kell keresni a memóriatartomány végét jelző nulla byte-ot, és így kiszámolni a méretet. Ezt úgy tehetjük meg, hogy egyenként lépkedve megvizsgáljuk a memóriacím olvashatóságát, és ha olvasható, akkor megnézzük, hogy ott nulla szerepel-e. Az olvashatóság vizsgálatra a kód meghívja saját magát, hiszen ekkor már a méret adott lesz (igaz, elég kicsi, hiszen csak 1 byte elérhetőségét fogja vizsgálni), azaz biztosan nem fog rekurzív módon a végtelenségig futni.

Ha sikerült megtalálni a tartomány végét jelző nullát, akkor annak címéből és a tartomány kezdetének címéből egyszerű kivonással megkapható a tartomány mérete, és innen már úgy folytatható a kód, mintha ez a méret már eleve adott lett volna.

Egy adott memóriatartomány ellenőrzésére a `VDHLockMem` helper használható, azaz meg kell próbálni a tartományt zárolni bizonyos célokra (olvasás illetve írás). Amennyiben ez nem sikerül, akkor a hívónak (ez esetben az adott VDM-nek) nincs joga a kívánt memóriaterületet a kívánt módon elérni. Ha sikerül a zárolás, akkor erre joga van, így a zárolás megszüntetése után (`VDHUnlockMem`) az eljárás sikeresen térhet vissza.

A következő kódrészlet az eljárás magja, az a rész, amely a tulajdonképpeni ellenőrzést végzi. Mikor a végrehajtás erre a részre ér, már `ESI` tartalmazza a memóriatartomány kezdőcímét, `ECX`-ben van annak mérete, és `EAX`-ben a kívánt jogok. A kód a végeredmény függvényében vagy törli a `Carry` flaget (jelezvén a sikert), vagy beállítja azt (amennyiben a zárolás sikertelen volt).

```
  push esi           ; StartingLinAddr
  push ecx           ; NumBytes
  push eax           ; OptionFlag
  push VDHLM_NO_ADDR ; PageListArrayPtr
  push -1            ; ArrayCountPtr
  call VDHLockMem

  or eax,eax
  jz short CAR_Error
  push eax           ; If no error occured,
  call VDHUnlockMem  ; release the lock handle,
  clc                ; and clear carry.
  jmp short CAR_Cleanup
 CAR_Error:          ; Otherwise set carry.
  stc
 CAR_Cleanup:
```

VDM_Access_Violation

Amennyiben valamilyen kérés feldolgozásakor az derül ki, hogy a VDM-nek nincs joga egy memóriaterületet elérni, a `VDM_Access_Violation` az, amelyre a kérést feldolgozó eljárásból a vezérlés kerül. Ennek a feladata a hibát valamilyen módon lekezelni.

A kód jellegzetessége, hogy nem eljárásként kerül rá a vezérlés, azaz nem `CALL`-lal hívódik meg, hanem `JMP`-vel, hiszen az eredeti kérés végrehajtásakor valamilyen hiba történt, így annak végrehajtása megszakadt, azt már nem kell folytatni.

Kérdés, hogy mit tegyünk akkor, ha a VDM-beli alkalmazás hibás memóriacímre hivatkozott. Nos, megtehetnénk, hogy értesítjük a felhasználót a hibáról (pl. a `VDHPopUp` segítségével), és felajánlhatjuk neki a DOS ablak bezárását (`VDHKillVDM`), vagy a hiba figyelmen kívül hagyását.

Mivel azonban a DOS-os alkalmazásoknál nem szokatlan dolog a hiba (illetve sokkal könnyebb előidézni), talán egy idő után terhessé válhat a sok figyelmeztető ablak, így egy kevésbé látványos módszer került implementálásra.

Mindenek előtt a kód értesíti a daemon-t a hibáról (a teljes regiszterkészlettel együtt). Ez jól jöhet pl. hibakereséskor. Ezek után a VDM számára jelzi a hibát olyan módon, mintha a kívánt funkció elvégzése meghiúsult volna, azaz beállítja a VDM `EAX` regiszterében a hibakódot (9-es hibakód, melynek jelentése érvénytelen memóriablokk) és beállítja a VDM-beli `Carry`-t is, mely jelzi, hogy hiba történt.

A kód végül egy `RET`-tel végződik, azaz az eredeti hívást kiszolgáló eljárás helyett ez a kód fog visszatérni (emlékezzünk rá, hogy erre a kódra minden esetben `JMP`-vel kerül rá a vezérlés!).

3.3. A DOS-os átirányító alkalmazás (LFN.COM)

Amennyiben a rendszer indításakor a VLFN.SYS betöltődött, az készen áll a VDM felől érkező kérések kiszolgálására. Az LFN.COM feladata, hogy a megfelelő rendszerhívások átirányításra kerüljenek.

Több módszer is előtérbe került, ahogyan ezt meg lehetne oldani. Az egyik az volt, hogy egy egyszerű, rezidens DOS-os programot kell írni, amely elfogja a 21h megszakításokat, melyek a DOS rendszerhívásai, és amelyeken keresztül az alkalmazások a fájl- és könyvtárműveleteket végzik. Az elfogott megszakításon keresztül aztán emulálja az összes szükséges rendszerhívást.

Ez azonban a fejlesztés közben zsákutcának bizonyult, mivel sok, egyébként emulálást nem igénylő rendszerhívás a DOS kernelén belül nem a 21h megszakítást használja a fájlműveletek elvégzésére, hanem saját, belső eljárásait, melyek normális esetben egyébként is meghívásra kerülnének a 21h megszakításból.

Más módszert kellett tehát választani. Így merült fel annak lehetősége, hogy a DOS kernelét kellene egy kicsit átírni olyan módon, hogy a kritikus helyeken a vezérlés a saját kódra kerüljön. A vezérlést ott kell átvenni, ahol a DOS kernele az OS/2 kerneléhez fordul a munka elvégzése érdekében az SVC hívások használatával (lásd a 2.1 részt a 6. oldalon). Az átirányítás elvégzésére vagy a *DOSKRNL* nevű fájlt kell módosítani, vagy a már memóriában levő, betöltött változatot. Az első ellen több minden is szól: nem biztos, hogy minden OS/2 változatban ugyanazon a helyen, ugyanolyan névvel szerepel ez a fájl, és egyébként is, egy rendszerfrissítéssel elveszhetne a „javításunk". Ezek tükrében a második módszer maradt az egyetlen járható út.

3.3.1. Az inicializálás

Mielőtt az LFN.COM az átirányításhoz hozzáfogna, meg kell vizsgálni, hogy az átirányítás biztonságosan elvégezhető-e. Ehhez az indulás után a következő lépéseket hajtja végre a programunk:

- Ellenőrzi, hogy tényleg OS/2 alatt fut-e. Ez a 2Fh megszakítás 4010h funkcióhívásával egyszerűen eldönthető:

```
mov ax,4010h              ; Check for OS/2
int 2fh
cmp ax,4010h
jne @OS2_Installed

mov dx,offset s_OS2_Needed
mov ah,09h                ; No OS/2 Found,
int 21h                   ; write error message
jmp @Normal_Exit
```

```
@OS2_Installed:
```

- Megvizsgálja, hogy a VLFN.SYS betöltésre került-e, és ha igen, milyen címen keresztül lehet kommunikálni vele:

  ```
  xor di,di
  mov es,di
  mov eax,4011h              ; Check for VLFN.SYS
  mov si,offset VLFN_DevName
  int 2fh
  mov word ptr VLFN[0],di
  mov word ptr VLFN[2],es
  mov eax,dword ptr VLFN
  or eax,eax
  jz @No_VLFN_Installed
  ```

  Itt a `VLFN_DevName` egy adatszegmensbeli string, tartalma „`VLFN`”, hiszen ezen a néven regisztrálta magát a VDD-nk, ha betöltődött. A `VLFN` nevű duplaszó fog mutatni arra a címre, ahová ugrani kell, ha valamit szeretnénk elvégeztetni a VDD-vel, illetve nulla lesz a tartalma, ha nincs ilyen cím, azaz a VLFN.SYS nem lett betöltve.

- A VDD segítségével ellenőrzi a futó daemon verziószámát:

  ```
  mov eax,0FF00h          ; Get VLFN Version (AH=0FFh)
  call [VLFN]
  cmp eax,'NODM'
  je @No_VLFN_Daemon
  cmp eax,0D00Dh
  jne @Bad_VLFN_Driver
  cmp eax,Daemon_Version_String
  jne @Bad_VLFN_Version
  ```

  A hívás után `EAX` kell, hogy tartalmazza a verziószámot jelző négy karaktert, például 0.91-es verzió esetén a „0”, a pont, a „9” és az „1” karakterkódokat.

Miután a kód megbizonyosodott róla, hogy megfelelő környezetben fut, felkészíti a VDD-t a futásra. Mivel ebben a VDM-ben eddig még semmilyen rendszerhívás nem ment keresztül a VLFN.SYS-en, így az nem tudhat róla, hogy melyik meghajtó aktív, illetve az egyes meghatókon melyek az aktív könyvtárak. Ezt mind meg kell mondani neki mielőtt az ténylegesen munkához láthatna. Le kell tehát kérdezni ezeket az információkat, majd továbbítani a VLFN.SYS-nek. Példának okáért az aktív meghajtót a következőképpen mondhatjuk meg a VDD-nek:

```
mov ah,19h                  ; Get act. drive
int 21h
mov dl,al                   ; Set act. drive
mov ah,SVC_demSetDefaultDrive
call [VLFN]
```

Az aktív könyvtárak beállítása is ezen az elven működik, csak ott szükség van egy kis buffer használatára, ahová lekérdezhetjük az adott meghajtó aktív könyvtárának nevét, illetve ezt végre kell hajtani minden lehetséges meghajtóra.

Ha ezen túl vagyunk, a VDD mindent tudni fog az adott VDM-ről ami a megfelelő működéséhez szükséges, el lehet kezdeni a DOS kernel átalakítását.

3.3.2. A DOS kernel megpatkolása

Ahhoz, hogy a DOS kernelt a memóriában felülírjuk néhány helyen, először meg kell azt keresni, hogy hol is van. Ezt végzi el a következő eljárás:

```
Get_DOSKRNL_Segment proc
  push ds
  mov ax,1203h            ; Get DOS data segment
  int 2fh
  mov ax,ds
  pop ds
  mov word ptr cs:DOSKRNL_Seg,ax
  ret
Get_DOSKRNL_Segment endp
```

A kód tulajdonképpen a DOS adatszegmensét kérdezi le, magától a DOS kernelétől. Ez viszont a gyakorlatban megegyezik a DOS kernelének kódszegmensével is, így már meg is van a cím.

Mielőtt továbbmennénk, tudnunk kell, mit is fogunk átírni a DOS kernelben, és mire. Mint már említettem, az SVC hívásokat kell átirányítanunk ahhoz, hogy sikeresen elvégezhessük feladatunkat. Az SVC hívások azonban csak három byte-osak, így eléggé korlátozott, hogy mit lehet a helyükre írni. A legkézenfekvőbb megoldás talán az, hogy írjuk egy nem használt megszakítás hívását a helyükre (pl. `INT AAh`), hiszen annak utasításkódja csak két byte, és a megszakítás kezelőjében a maradék egy byte-ból megtudhatjuk, hogy milyen SVC volt ott eredetileg.

Nos, ehhez először „rá kell ülnünk” a kívánt megszakításra. Ezt végzi el a `SetSupportInts` eljárás a DOS 35h és 25h funkcióhívásai segítségével (megszakítás lekérdezése és beállítása). Sajnos az előbb említetten kívül egy másik megszakításra is szükség lesz, ezért ez az eljárás két megszakítást fog el, az `AAh` és az `ABh` megszakításokat. A második feladatáról egy kicsit később esik majd szó, egyenlőre elégedjünk meg annyival, hogy szükség lesz rá.

A kernel átírását a `Patch_DOSKRNL` nevű eljlárás végzi, mely az előbbiek után fut le:

```
Patch_DOSKRNL proc
  push es
  xor bx,bx
  mov es,cs:DOSKRNL_Seg
  mov cx,MAX_DOSKRNL_SIZE

 @Patch_More:
  call Is_Anything_Patchable_Here
  jnc @Patch_Next_Address
  mov word ptr es:[bx],0AACDh  ; Patch memory to INT AAh
 @Patch_Next_Address:
  inc bx
  dec cx
  jnz @Patch_More

  pop es
  ret
Patch_DOSKRNL endp
```

A kód tulajdonképpen csak végigmegy a memóriában levő DOS kernelen, és ha ott olyan részt tal ál, amit át kell írnia, oda egy AACDh hexadecimális számot ír, ami tulajdonképpen az `INT AAh` utasítás kódja.

Annak eldöntését, hogy egy adott memóriaterületen átírandó adat szerepel-e, az `Is_Anything_Patchable_Here` végzi. Ez megvizsgálja a memória `ES:BX` által címzett területét, és ha ott olyan byte-hármast talál, amely megfelel egy SVC hívásnak (lásd a 6. oldalon), akkor tovább kutakodik. Megvizsgálja a hívandó SVC kódját, és egy helyi táblázatból kikeresi, hogy azt az SVC hívást át kell-e irányítani. Ez a helyi táblázat a felépítése szempontjából megegyezik a VDD-nél már bemutatott ugró-táblával (lásd a 3.1. táblázatot a 21. oldalon), és szerepe is ugyanaz.

3.3.3. Az AAh megszakítás

Miután az LFN.COM átírta a DOS kernelt, befejezheti futását, de természetesen csak úgy, hogy egy része a memóriában maradjon, és kiszolgálja a beérkező megszakításokat, amiket átirányított.

Mivel az előbbiek eredményeképpen minden, a helyi ugró-táblában szereplő SVC hívás át lett irányítva, így ha egy AAh megszakítás történik, akkor az valószínűleg egy olyan SVC hívás volt eredetileg, amit nekünk kell feldolgozni.

Az AAh megszakítás új kezelője tehát megvizsgálja, hogy milyen byte szerepel azon a memóriacímen, ahová majd vissza kell térnie. Ezt a címet a veremből egyszerűen megtudhatja, hiszen annak tetején ez a cím kell hogy legyen.

Ott elméletileg az SVC utolsó byte-ja szerepel, ami alapján az ugró-táblából megtalálhatja a végrehajtandó eljárás címét. Ha ezt nem találja, akkor valami más hívta meg az AAh megszakítást, így nem csinál semmit, csak végrehajtja a régi megszakításkezelő rutint.

Ha megvan a táblázatból a hívandó rutin címe, akkor azt kell végrehajtani. Van azonban két apróság, amiről nem szabad elfelejtkezni:

- A megszakításból való visszatérés előtt módosítani kell a visszatérési címet a vermen, hiszen az az SVC utolsó byte-jára mutat, amit át kell lépni!
- Megszakítás hívásakor a rendszer nemcsak a visszatérési címet, de a flageket is elmenti, majd a megszakítás befejezésekor visszaállítja azt. Mivel a mi rutinunk a flageken keresztül jelzi ha hiba történt, így a visszatérés előtt az elmentett flageket is át kell írni a vermen az aktuális flagekre.

3.3.4. Az egyes rutinok

Miután az ugró-tábla alapján az egyes rutinokra kerül a vezérlés, azok többféle módon végzik el feladatukat. Sokuk egyszerűen csak meghívja a VLFN.SYS megfelelő rutinját, mint például a könyvtár létrehozása parancs:

```
VLFN_CreateDir proc
  mov dword ptr cs:SaveEAX,eax
  mov ah,SVC_demCreateDir
  call cs:[VLFN]
  jc @CD_RetError
  mov eax,dword ptr cs:SaveEAX
@CD_RetError:
  ret
VLFN_CreateDir endp
```

Az `EAX` regiszter elmentésére azért van szükség, mert a VLFN-nek az `AH`-n keresztül mondjuk meg, hogy mit kell tennie, és amennyiben nem történt hiba, ezt a regisztert vissza kell állítanunk az eredeti állapotába. Ha hiba történt akkor nem, hiszen ez esetben `EAX` tartalmazza a hibakódot.

Van olyan rutin is, ahol az eredeti SVC-t is el kell végezni, hogy az OS/2 kernele is tudjon az eseményről. Ilyen esemény például az aktív meghajtó váltása:

```
VLFN_SetDefaultDrive proc
  push eax
  push edx
  hlt
  db SVC_demSetDefaultDrive
  db not SVC_demSetDefaultDrive
```

```
  pop edx
  pop eax

  mov byte ptr cs:SaveEAX,ah
  mov ah,SVC_demSetDefaultDrive
  call cs:[VLFN]
  mov ah,byte ptr cs:SaveEAX
  ret
VLFN_SetDefaultDrive endp
```

Egy harmadik fajta rutin is gyakori, amely az előbbi kettő keverékének tekinthető. Amennyiben valamilyen fájlműveletre érkezik kérelem, akkor a fájl-azonosítótól függően vagy az eredeti SVC-t kell végrehajtani (ekkor a fájlt még az eredeti kernel nyitotta meg), vagy továbbítani kell a VLFN.SYS-nek (ekkor a fájlt már mi nyitottuk meg). Ez utóbbi akkor lehet, ha a fájl-azonosító beleesik abba a tartományba, amelyben a VLFN.SYS osztogatja az azonosítókat. Ilyen rutin például a fájl bezárása:

```
VLFN_Close proc
  cmp bx,VLFN_Handle_Base
  jb @Close_Original_Handle
  cmp bx,Max_VLFN_Handle
  ja @Close_Original_Handle

@Close_VLFN_Handle:
  mov ah,SVC_demClose
  call cs:[VLFN]
  ret

@Close_Original_Handle:
  hlt            ; Not our handle, call original close process!
  db SVC_demClose
  db not SVC_demClose
  ret
VLFN_Close endp
```

Van azonban egy olyan SVC, amely természete miatt sokkal bonyolultabb kezelést igényel, sőt, átirányításához még egy extra megszakításra is szükség volt.

3.3.5. Az ABh megszakítás

A *demDupJFT* nevű SVC egy igen nehéz problémát vetett fel. Akkor kerül meghívásra, ha egy DOS-os alkalmazás egy másik DOS-os alkalmazást indít. Ekkor az új alkalmazás számára minden eddig már megnyitott fájl-azonosítót duplikálni kell.

Az SVC feltételezi, hogy hívásakor paraméterként megkapja a nyitott fájlok táblázatát, melyben dolgozhat. Ez a táblázat azonban tartalmazza azokat a fájl-azonosítókat is, amelyeket a VLFN nyitott meg, és ezeket nem szabad átadni az eredeti rutinnak, hiszen az nem tud mit kezdeni velük!

Ezt természetesen csak úgy lehet megoldani, hogy a nyitott fájlok táblázatából ideiglenesen kitöröljük a VLFN által megnyitott fájl-azonosítókat, a ,,megtisztított" táblázatra meghívjuk az eredeti SVC-t, majd visszaírjuk a táblázatba a VLFN által megnyitott fájlok azonosítóit, miközben duplikáljuk azokat.

Az igazi problémát az jelenti, hogy az eredeti SVC feltételezi, hogy a táblázatra a `CS:DI` mutat. Az eredeti SVC tehát csak akkor hajtható végre helyesen (márpedig végre kell hajtani), ha az eredeti helyéről, az eredeti kódszegmensből hívódik meg.

A probléma tehát adott: valami módon az eredeti helyéről kell végrehajtani ezt az SVC-t! Ez csak úgy oldható meg, ha amikor a mi kódunk az `AAh` megszakítás kezelőjében észreveszi, hogy ezt a problémás SVC-t kell lekezelnie, akkor az eredeti helyre visszaírja az SVC-t, plusz utána ír egy `INT 0ABh` utasítást, majd a vermen levő visszatérési cím módosítása által a megszakításból visszatérve erre ráadja a vezérlést. Ekkor az SVC az eredeti helyéről fog meghívódni, és utána az `ABh` megszakítás kerül sorra, ahol mindezt visszacsinálhatjuk.

Ezzel a módszerrel az eredeti, `AAh` megszakításból lehetőségünk nyílik az SVC előtt a nyitott fájlok táblázatát ,,megtisztítatni", illetve lehetőségünk van az `ABh` megszakításból az SVC után elvégzendő munkák végrehajtására. Ezért van tehát szükség két megszakítás elfogására.

Lássuk most szemléletes módon, táblázatok segítségével, hogyan működik ez a módszer!

Kód	***Mnemonik***
...	
f4	hlt
2a	db 2ah
d5	db 0d5h
??	
??	
??	
...	

Eredeti kód

Kód	***Mnemonik***
...	
cd	int aah
aa	
d5	db 0d5h
??	
??	
??	
...	

Módosított kód

Kód	***Mnemonik***
...	
f4	hlt
2a	db 2ah
d5	db 0d5h
cd	int abh
ab	
??	
...	

Ideiglenes kód

3.3. ábra. Ugyanazon kódrészlet három időpillanatban

A 3.3. ábra első táblázata mutatja a DOS kernel egyik kódrészletének eredeti állapotát, azaz magát az SVC-t. Ezt az LFN.COM az elindulása után megtalálja, és átírja a második táblázatnak megfelelő módon. Ha erre a módosított kódra kerül a vezérlés, akkor az `AAh` megszakítás fog meghívódni. Ebben észrevesszük a maradék

`0d5h` byte-ból, hogy a problémás *demDupJFT*-vel van dolgunk, így visszaírjuk az eredeti kódot az `int 0aah` helyére és lecseréljük az így visszaállított SVC utáni két byte-ot egy `int 0abh`-ra (természetesen gondosan lementjük az ott levő eredeti byte-okat, így később azt visszaállíthatjuk). Ekkor kapjuk a harmadik táblázatbeli ideiglenes kódot.

Mivel a visszatérési cím a vermen keresztül elérhető, így megtalálhatjuk azt a táblázatot, amire eredetileg a `CS:DI` mutat. Erről a táblázatról egy másolatot készítünk, miközben a táblázat azon fájl-azonosítóit, melyeket mi készítettünk, kitöröljük a táblázatból.

Újra a vermen levő visszatérési címhez kell nyúlni: csökkenteni kell azt kettővel, hogy a megszakításból visszatérve a nemrég odaírt SVC-re kerüljön a vezérlés, majd visszatérni.

Ezek hatására az eredeti helyén, az eredeti SVC fog végrehajtódni, és az operációs rendszer duplikálni fogja azokat a fájl-azonosítókat, melyeket ő hozott létre. A duplikálás után kiváltódik az `ABh` megszakítás, ahol a következő a dolgunk:

Vissza kell állítani a kódrészletet a 3.3. ábra második táblázatának megfelelő módon, majd feltölteni a nyitott fájlok táblázatát a duplikált VLFN-féle fájl-azonosítókkal. A visszatérési cím újbóli módosításával a megszakítás után a visszaállított kódrészletre adható a vezérlés.

Nos, talán egy kicsit komplikált módon, de ezt a nehéz problémát is sikerült megoldani. Most már tudjuk, hogyan fogjuk el az SVC-ket, hogyan kerülnek azok a VDD-hez, illetve onnan hogyan jutnak el a daemonhoz. Már csak az a kérdés, hogyan dolgozik a daemon.

3.4. A daemon (VLFND.EXE)

A VLFN eddigi részei vagy eszközmeghajtó programok voltak, vagy feltétel volt a lehető legkisebb méret. Emiatt az eddigi programkódok mind Assembly nyelven íródtak.

A kiszolgáló programnál a méret kevésbé fontos, itt inkább az kerül előtérbe, hogy bonyolult feladatokat kell elvégeznie, így ez egy magas szintű nyelven került megírásra. A program írásakor vagy valamilyen C/C++ fordító (Watcom C++, IBM C Set Compiler, Borland C++ for OS/2, Microsoft C stb.), vagy valamilyen Pascal fordító (Speedsoft Sibyl, Virtual Pascal, TMT Pascal stb.) jöhetett szóba. Végül a választás a Virtual Pascalra esett, főleg mert ez egy ingyenesen használható, kitűnő OS/2 támogatással rendelkező fordítóprogram.

A daemon feladata, hogy a VDD felől érkező kéréseket kiszolgálja. Mindezt bonyolítja, hogy képessé kell tenni a daemont arra, hogy a hosszú nevű fájlokhoz rövid nevet generáljon, és később erről a rövid névről azonosítani tudja, melyik fájlról vagy könyvtárról van szó.

3.4.1. A névátalakítás

Gyakorlatilag két különféle módon kell működnie a daemonnak, bár mint később látni fogjuk, a kettő bizonyos szempontból hasonlít is egymásra.

Az egyik mód az, amikor az adott meghajtón a hosszú fájlnevek tárolása lehetséges, és ezekhez kell megjegyezni a hozzájuk társított, egyedi rövid fájlnevet. Ez a leggyakoribb eset, mivel az OS/2 alatt a FAT fájlrendszer-típuson kívül minden más fájlrendszer támogatja a hosszú fájlneveket. A fájlokhoz társított rövid fájlnév tárolására kihasználhatjuk az úgynevezett kibővített attribútumokat (*Extended Attributes*, EA), amelyek tulajdonképpen minden egyes fájlhoz vagy könyvtárhoz csatolható meta-adatok. A módszer tehát az, hogy a hosszú fájlnevekhez generálunk egy egyedi rövid fájlnevet, és ezt letároljuk az adott fájl vagy könyvtár `.SHORTNAME` nevű EA-jában.

A másik működési mód az, amikor az adott meghajtó csak a rövid fájlneveket támogatja (pl. FAT). Ilyenkor az előző fordítottja a feladat: a hosszú fájlneveket kell letárolni az eredeti, rövid nevű fájlokhoz. Alapesetben az OS/2 is ezt csinálja, amennyiben a WPS-ből (Workplace Shell, az OS/2 grafikus felületének neve) egy hosszú nevű fájlt hozunk létre a FAT meghajtón: generál egy rövid fájlnevet, mellyel létrehozza a fájlt, majd a fájl `.LONGNAME` EA-jaként hozzácsatolja a hosszú fájlnevet.

Ehhez szükség van arra, hogy az OS/2 CPAPI-ját (Control Program API) „körülvegyük" egy héjjal, amely a megkapott fájlnévhez megkeresi az igazi, fizikai fájlnevet, azaz például a VDM felől kapott rövid fájlnévhez megkeresi a hozzá tartozó fájlt, és ezt adja át a kívánt OS/2-es API-nak.

Ezeket figyelembe véve a daemon a következő fő részekből áll:

- Főprogram, mely a VDD kéréseit fogadja, és azokat végrehajtja.
- LFN_Unit, mely a héj szerepét tölti be, azaz amely elvégzi a fájlnév átalakítást.
- LFN_EA, melyet az előző unit használ arra, hogy a társított rövid illetve hosszú fájlneveket a fájlok és könyvtárak EA-jában lementhesse illetve azokat visszaolvashassa.
- LFN_Cache, mely a memóriában tárolja a használt EA-kat, ezzel gyorsítva a keresést és használatot.

3.4.2. A főprogram

A VDD-vel való kommunikációhoz mindenekelőtt szükség van a 128 KByte-os memóriaterületre, amin keresztül a kommunikáció történik. Mivel ezt a PDD (a VLFNMEM.SYS) foglalta le, tőle kell a címet kérni, mégpedig a megnyitás után egy egyszerű IOCTL hívással (a PDD felőli részről a 15. oldalon volt szó). Ezt a daemon `PDD_Init` nevű eljárása végzi, mely így néz ki:

```
procedure PDD_Init;
Var ulAction, rc : ApiRet;
    hPDD : os2base.HFile;
    l : longint;
begin
  if DosOpen( 'VLFNMEM$',
              hPDD,
              ulAction,
              0, 0,
              OPEN_ACTION_OPEN_IF_EXISTS,
              OPEN_ACCESS_READWRITE or
              OPEN_SHARE_DENYREADWRITE,
              nil ) <> 0 then
  begin
    LogLN(llAlwaysScreen,'Cannot open VLFNMEM$!');
    LogLN(llAlwaysScreen,'Please install VLFNMEM.SYS first!');
    Halt(1);
  end;
  rc := DosDevIOCtl( hPDD, $ff, $ff,
                     nil, 0, nil,
                     @pBuffer,
                     4,
                     nil );
  if rc=0 then
    DosClose(hPDD);
  else
  begin
    LogLN(llAlwaysScreen,'Error querying '
          'Memory-frame address from VLFNMEM$!');
    LogLN(llAlwaysScreen,'rc = '+hex(rc));
    DosClose(hPDD);
    halt(2);
  end;
end;
```

A kód törzsét tulajdonképpen a `DosOpen` – `DosDevIOCtl` – `DosClose` hármas képezi. Az első az, amely megnyitja a PDD-t. A sikeres megnyitás után jöhet a kommunikáció, ami a jelen helyzetben egy kissé rendhagyó, hiszen a VLFNMEM.SYS nem követi az IOCTL hívási konvencióit, figyelmen kívül hagyja a *kategória* és *funkció* paramétereket (a második és harmadik paraméter), hiszen feltételezi, hogy csak mi fogjuk meghívni. A sikeres IOCTL hívás után tehát a `pBuffer` nevű globális változó fogja tartalmazni a kommunkációs memóriaterület címét. Ezután a megnyitott azonosítót a `DosClose`-zal be kell zárni, és a feladatot elvégeztük.

A VDD-vel való kommunikáció hasonló. Ott is meg kell nyitni a meghajtóprogramot (`DosOpenVDD`), be kell hívni oda (itt IOCTL helyett *„kérni”* kell tőle valamit a `DosRequestVDD` API segítségével), majd a kommunkáció befejeztével (a daemon bezárásakor) bezárni azt (`DosCloseVDD`).

Ha mind a két meghajtóprogramot sikerült megnyitni, akkor valószínűsíthetjük, hogy a kommunikáció működni fog, így felkészíthetjük magunkat a kiszolgálói futásra. Ez azt jelenti, hogy a programunk prioritását megemeljük, hiszen ez egy kiszolgáló program, aminek lehetőleg több időszeletet kell kapnia, mint egyéb programoknak, vagy akár a kiszolgálandó DOS ablakoknak. A prioritás állítására a `DosSetPriority` API használható, melynek formátuma a következő:

```
DosSetPriority(scope, ulClass, delta, PorTid);
```

A *scope* tartalmazza azt, hogy minek a prioritását akarjuk állítani, egy szálét, az egész processzét, vagy az egész processz-fáét. Esetünkben biztosra megyünk, ennek értéke `prtys_ProcessTree`. A *PorTid* tartalmazza a módosítandó processz vagy szál azonosítóját. Ez esetünkben nulla, jelezvén hogy a hívó processzt szeretnénk módosítani.

Az *ulClass* és *delta* paraméterek állítják be a tulajdonképpeni prioritást. Az OS/2 egy igen kifinomult prioritás-rendszert alkalmaz, melyben minden szál négy különböző prioritás-osztály közül választhat, és mindegyik prioritás-osztály tovább finomítható 63 szinten (ez a *delta*, mely -31 és 31 közt vehet fel értéket). A négy prioritás-osztály a következő:

- IdleTime: Ekkor az adott szál csak akkor kerül végrehajtásra, ha a processzornak már semmi más dolga nincs.
- Regular: Ez az alap beállítás, minden program (hacsak nem módosítja a prioritását) ezen osztály nulla deltájú prioritásával indul.
- TimeCritical: Ez a különféle kiszolgáló programok osztálya, például Web szerver vagy FTP szerver.
- ForegroundServer: Az elérhető legmagasabb prioritás. Használata csak nagyon indokolt esetben javasolt, mert egy hibás *ForegroundServer* prioritású szál nagyon lefoghatja a rendszert, akár percekig válaszképtelenné is teheti azt.

A VLFN Daemon alapesetben TimeCritical+0 prioritást állít be magának, de ezt a felhasználó parancssori kapcsolókkal felülbírálhatja.

Mivel a kommunikáció a VDD-vel egy 128 KByte-os közös memóriaterületen keresztül történik, így nincs szükség a kéréskor bármilyen paramétert átadni, a VDD úgyis a memóriaterületet fogja figyelni. A `DosRequestVDD` tehát minden paramétere figyelmen kívül hagyható:

```
rc:=DosRequestVDD(hvlfn,0,0,0,Nil,0,Nil);
```

Amennyiben ez az API visszatér, a VDD összeállított egy kérést a számunkra, tehát a `pBuffer` által mutatott memóriaterületet megvizsgálva a kérésnek megfelelő kiszolgáló kódra kell adni a vezérlést. Mivel az osztott memóriaterület elején mindig a hívó VDM regiszterkészlete szerepel, az ott levő `AH` regiszter megvizsgálásával a kérés kideríthető, és egy egyszerű `CASE` szerkezettel lekezelhető.

Lássunk most néhány kérést, hogyan lehet teljesíteni őket!

VLFN_Install_Check

Talán a legegyszerűbb feladat válaszolni az installáltság ellenőrzésére, hiszen csak két regisztert, az `EAX`-et és az `EBX`-et kell beállítani egy előre megadott értékre:

```
procedure VLFN_Install_Check;
var ulVersion:ulong;
begin
  LogLN(ModuleLogLevel,' Sending answer to Install Check');
  SetDataL(crf_EAX,$D00D);        { EAX = $D00D}
  ulVersion:=ulong(Daemon_Version_String[1])+
             ulong(Daemon_Version_String[2]) shl 8+
             ulong(Daemon_Version_String[3]) shl 16+
             ulong(Daemon_Version_String[4]) shl 24;
  SetDataL(crf_EBX,ulVersion);    { EBX = Daemon's version}
end;
```

Mint láthatjuk, a beállítást a `SetDataL` nevű eljárás végzi. Ez a `pBuffer` változó által mutatott memóriaterületen állít be egy `longint` típusú adatot, mégpedig az első paraméter által mutatott címen. Ennek segítségére a leggyakrabban használt regiszterek helyei deklarálásra kerültek (pl. `crf_EAX`), így egy jól olvasható kódot sikerült létrehozni.

Ennek a segédeljárásnak létezik több változata is, melyet a program egyéb részei használnak, melyek `byte` vagy `word` típusú adatokat állítanak be, sőt, lekérdező változatuk is van (pl. `GetDataB`).

VLFN_demCreateDir

Az előbbinél bonyolultabb eljárás a könyvtár létrehozását elvégző eljárás.

```
Procedure VLFN_demCreateDir;
var s:string;
    rc:apiret;
    flag:byte;
begin
  s:=GetAsciiZ(data);
  LogLN(ModuleLogLevel,'  MKDIR ['+s+'] ');
```

```
  rc:=LFN_CreateDir(s);
  flag:=GetDataB(crf_eflag);
  SetDataB(crf_eflag,flag and 254);
  if rc<>0 then
  begin
    SetDataB(crf_eflag,flag or 1);
    SetDataW(crf_eax,rc);
    LogLN(ModuleLogLevel,
          '    Error: rc='+Hex(rc)+' (CF set)');
  end;
end;
```

Itt ismeretlen a `GetAsciiZ` és az `LFN_CreateDir` lehet. Az első az osztott memóriaterületre elhelyezett különféle fájl- és könyvtárneveket adja vissza mint Pascal-kompatibilis `string`. Ezt a stringet adjuk tovább a könyvtár létrehozását elvégző `LFN_CreateDir`-nek, amely, mint minden „`LFN_`”-nel kezdődő eljárás és függvény, a későbbiekben bemutatásra kerülő *LFN_Unit* része.

Amennyiben a létrehozáskor hiba történik, ezt jelezni kell a VDM részére a `Carry` flag beállításával (ez az `eflags` első bitje), illetve az `AX` regiszter hibakódnak megfelelő beállításával.

VLFN_demOpen

A fájlok megnyitásakor kerül végrehajtásra a következő eljárás. Ennek jellegzetessége, hogy a fájl megnyitása OS/2 alatt nem teljesen kompatibilis a DOS alatti fájl megnyitással. Van például olyan fájl-megnyitási mód, amely DOS alatt engedélyezett, de OS/2 alatt nem, illetve a DOS a fájl megnyitása után a megnyitásra került fájl típusát is szolgáltatja, amit OS/2 alatt csak egy extra API hívásával tudhatunk meg.

Az előző eljáráshoz hasonlóan ez is AsciiZ stringként kapja meg a megnyitandó fájl nevét, de a fájl megnyitásán kívül ennek a következőket is el kell végeznie:

- A fájl-megnyitási módot OS/2 kompatibilissé kell alakítani. Ezt néhány bit vizsgálatával és átállításával meg lehet oldani.
- Ha sikerült a fájlt megnyitni, le kell kérdezni a fájl-azonosító típusát. Az operációs rendszer által szolgáltatott fájl-azonosítók esetén ezt a `DosQueryHType` segítségével lehet megtenni, míg a mi esetünkben az *LFN_Unit*-beli megfelelőjét, az `LFN_DosQueryHType`-ot kell használnunk.

Mindezt a következő eljárás és a benne foglalt segédeljárás végzi el:

```
Procedure VLFN_demOpen;
var s:string;
```

```
    flag:byte;
    rc:apiret;
    Attribs:ulong;
    OpenFlags,OpenMode,Action:ulong;
    DOSOpenMode:SmallWord;
    filehandle:lfn_File;
    Handle_Flags:ulong;
    hType,hAttr:ulong;

  function ConvertOpenMode(DOM:SmallWord):ULong;
  begin
    Result:=DOM;
    if (Result and 7)=3 then        // Bits 2-0 :  011
    begin // Allowed in DOS, not in OS/2!
      Result:=Result and (not 1);   // Let it be 010 instead.
    end;                            // (Read/Write)
    if (Result and 112)=0 then      // Bits 6-4 :  000
    begin // Compatibility mode. Allowed in DOS, not in OS/2!
      Result:=Result or Open_Share_DenyNone;
    end;
  end;

begin
  flag:=GetDataB(crf_eflag);
  SetDataB(crf_Eflag,flag and 254); {Clear CF}
  DOSOpenMode:=GetDataW(crf_ebx);  {BX = Open Mode}
  OpenMode:=ConvertOpenMode(DOSOpenMode);
  Attribs:=GetDataW(crf_ecx) and $3f;
                                {CX = File Attrib., max 3fh!}
  OpenFlags:=GetDataW(crf_edx); {DX = Open Flags}
  s:=GetAsciiZ(data);
  LogLN(ModuleLogLevel,'  Open/Create ['+s+'] ');

  rc:=LFN_DosOpen(s, filehandle, Attribs,
                OpenFlags, OpenMode);
  if rc<>no_Error then
  begin
    SetDataL(crf_EAX,rc);
    SetDataB(crf_Eflag,flag or 1); {Set CF}
    LogLN(ModuleLogLevel,
         '    Error: rc='+Hex(rc)+' (CF set)');
  end else
```

```
  begin
    SetDataL(crf_EAX,filehandle);
    LogLN(ModuleLogLevel,'     Handle: '+
          LongToStr(filehandle));
    SetDataW(crf_ecx,LFN_DOSOPEN_Action_Taken);
    hAttr:=0;
    LFN_DosQueryHType(filehandle,hType,hAttr);
    Handle_Flags:=hType and $8000;  // SF_ISNET
    Handle_Flags:=Handle_Flags or (hAttr and $FF);
    SetDataW(crf_ebx,Handle_Flags); { DOS-style flags... }
  end;
end;
```

Nos, az eddig bemutatott eljárások nagyvonalakban bemutatják a daemon működését, illetve a módszer ismeretében a többi eljárás is könnyen elképzelhető. Mint látható, szinte csak az osztott memóriaterület kezelése folyik itt, a lényeges dolgokat, azaz például egy fájl megnyitását vagy egy könyvtár létrehozását és az ezzel járó fájlnév-átakítást az *LFN_Unit* függvényei végzik. Következzen most ennek a unitnak a bemutatása!

3.4.3. LFN_Unit

Az *LFN_Unit* feladata, hogy olyan fájl- és könyvtárkezelő eljárásokat és függvényeket szolgáltasson, melyek a megkapott fájlnévből megállapítják a valós fájlnevet (FAT meghajtó esetén a rövid fájlnevet, egyéb meghajtók esetén a hosszú fájlnevet), és erre a valós fájlnévre hívják meg a megfelelő `Dos*` függvényeket.

Meghajtótípusok lekérdezése

Mivel minden fájlnévről pontosan el kell tudni dönteni, hogy hol van, az eljárások csak teljes fájlnevet fogadnak el, azaz kötelezően szerepelnie kell benne a meghajtó betűjelének és a teljes elérési útnak is. Ez természetesen a VLFN szempontjából nem probléma, hiszen itt már a VDD-ben kanonizált alakra hozunk minden fájlnevet, tehát azok onnan már csak teljes elérési úttal kerülhetnek ide.

A meghajtó betűjele alapján megállapítható a működési mód, hogy a valós név egy rövid vagy egy hosszú fájlnév lesz-e, illetve hogy a `.LONGNAME` vagy a `.SHORTNAME` EA-t kell-e használni azok tárolására.

A meghajtók típusait, és ezek alapján a hozzájuk kapcsolódó működési módot a unit `Query_DriveInfo` eljárása kérdezi le. Ebben megpróbáljuk kihasználni, hogy a `DosQueryFSAttach` API segítségével lekérdezhető a felcsatolt fájlrendszer neve. Ezt az API-t minden lehetséges meghajtóra lefuttatva (A-tól Z-ig), ha sikeresen lefut, egy stringet ad vissza, amely a meghajtó fájlrendszerének nevét tartalmazza, mint például FAT, HPFS, JFS, CDFS vagy FAT32. Ez alapján egy tömbben beállíthatjuk az adott meghajtóhoz a kívánt működési módot.

Megeshet, hogy ez az API nem tud lefutni sikeresen. Ez viszont még nem jelenti azt, hogy nem tudjuk meghatározni a meghajtó típusát. Amennyiben a hibakód `error_Not_Ready` vagy `error_Sector_Not_Found`, akkor megeshet hogy egy CD-meghajtóval találkoztunk, ami épp nem tartalmaz CD-t, vagy egy audio-CD van berakva. Ekkor egy egyszerű IOCTL hívással (8-as kategória, 63h funkció) lekérdezhetjük, hogy az adott meghajtó egy CD-ROM-e. Ha igen, akkor feltételezzük, hogy a meghajtóhoz tartozó fájlrendszer CDFS lesz.

Valós fájlnév meghatározása

Mivel minden *LFN_Unit*-beli eljárás első feladata, hogy a kapott fájlnévből meghatározza a valós fájlnevet, amin dolgoznia kell, szükséges volt írni egy általános függvényt, amely ezt elvégzi. Ez a `GetRealName` nevet kapta, és a paraméterül kapott fájl- vagy könyvtárnévhez határozza meg a valós elérési utat és nevet. Mivel a meghajtó típusától függően más-más módon kell ezt elvégezni, ezért ennek első dolga, hogy megvizsgálja a meghajtóhoz rendelt működési típust:

```
Function GetRealName(s:string):string;
var drv:string;
begin
  if length(s)=0 then
  begin
    result:='';
    exit;
  end;
  drv:=s[1]; drv:=AnsiUpperCase(drv);
  Case DriveInfo[drv[1]].UsageMode of
    umFAT:  GetRealName:=GetRealName_For_FAT(s);
    umHPFS: GetRealName:=GetRealName_For_HPFS(s);
    umCDFS: GetRealName:=GetRealName_For_HPFS(s);
  else
    GetRealName:=GetRealName_For_GOTHROUGH(s);
  end;
end;
```

A meghívott függvények nagyon hasonló módon működnek, csak az a különbség, hogy az egyik azt feltételezi, hogy a hosszú fájlneveket kell az EA-ban keresni, míg a másik a rövideket keresi ott. A módszer egyébként az, hogy a kapott elérési utat elejétől a végéig végig kell nézni, és minden egyes köztes könyvtárnevet is meg kell vizsgálni, hogy olyan könyvtár az adott helyen létezik-e, vagy esetleg egy másik nevű könyvtár van ott, melyhez olyan `.LONGNAME` vagy `.SHORTNAME` EA van csatolva, ami a megadott könyvtárnevet tartalmazza. Ez alapján minden könyvtárnevet plusz a string végén szereplő fájlnevet is ki lehet cserélni a valódi névre.

Fájlnevek generálása

Miközben a kód egy adott könyvtárban azt keresi, hogy egy adott `.SHORTNAME` EA-val rendelkező fájl létezik-e, kénytelen minden ott levő fájlt végignézni, és lekérdezni a hozzá tartozó EA-t. Amennyiben a fájl még nem tartalmaz ilyet, generál is hozzá, hogy legközelebb már ilyen probléma ne legyen.

A rövid fájlnév generálása a legproblémásabb feladat. Mindenekelőtt a generált rövid névnek meg kell felelnie azoknak a követelményeknek, amelyeket a FAT meghajtó támaszt, azaz 8+3 karakter hosszú lehet, és van egy sor olyan karakter, amely nem szerepelhet benne. Az első feladat tehát a nem használható karakterek kicserélése valami másra, illetve a fájlnév levágása, hogy 8+3 karakternél ne legyen hosszabb. Például a már ránézésre hosszú „`Hosszu filenev.mai.txt`" névből „`Hosszu_f.txt`" lesz.

Természetesen ettől még sok hosszú nevű fájlhoz ugyanazt a nevet kapnánk, tehát ez még nem elég. A más operációs rendszerekből már megszokott módon tehát sorszámot adhatunk a rövid fájlneveknek. Számozzuk hát meg az eddig kapott, 8+3 karakter hosszúra levágott fájlnevet, mint első változat. Valami ilyet kapunk tehát: „`Hosszu~1.txt`".

Ezek után mindig megvizsgáljuk, hogy az adott rövid név szerepel-e már valamelyik fájlnál az adott könyvtárban, és amíg igen, eggyel növeljük a verziószámot.

Ezzel a módszerrel tehát egyedi rövid fájlnevek generálhatóak a hosszú fájlnevekből, de figyeljük meg, hogy sajnos nagyon sokszor kell a fájlok EA-it végigvizsgálni. A fájlnevek generálásakor például minden egyes verzióhoz végig kell nézni a teljes könyvtártartalmat, hogy nincs-e valamelyik fájlhoz a keresett EA csatolva, de a valós fájlnév meghatározásához is ugyanezt kell tenni, sokszor ugyanazokon a fájlokon egymás után.

Logikus tehát, hogy a lassú fájl-elérést valami módon felgyorsítsuk, és puffereljük a már megismert EA-kat, így ismételt kereséskor már nem kell újra a merevlemezhez nyúlni. Ezt fogja bemutatni a 3.4.5. rész az 51. oldalon, de előtte még lássuk az *LFN_Unit* egy másik érdekes részét!

FindFirst / FindNext

A fájlok és könyvtárak kezelésén kívül egy könyvtár tartalmának lekérdezése, illetve adott fájl(ok) keresése az, amit a VDM-ek helyett el kell végeznünk. Erre a DOS is ugyanazt a módszert használja, amit az OS/2, azaz a *FindFirst* – *FindNext* párost alkalmazza. A *FindFirst* az, amelynek meg kell adni a keresendő fájl specifikációját (például „**.EXE*") és egyéb feltételeit, mire az visszaadja az első, a keresés feltételeinek megfelelő fájlt. A többi fájlt a *FindNext* sorozatos meghívásával lehet megkapni.

Megbonyolítja a helyzetet, hogy a mi esetünkben az operációs rendszer eredeti rutinjai közvetlenül nem használhatóak, hiszen azok nem tudják a `.SHORTNAME` vagy a `.LONGNAME` EA-kat fájlnévként kezelni, és azokban keresni. Szükséges volt tehát

ezeknek a rutinoknak a megírása is.

Amennyiben a VDM egy könyvtár tartalmára kíváncsi, a fájlokhoz rendelt rövid fájlneveket kell szolgáltatnunk a részére, illetve azokat kell vizsgálnunk, hogy megfelelnek-e a keresési kritériumoknak. Sajnos ezt csak úgy tehetjük meg, ha az adott könyvtár minden egyes állományához rendelt rövid nevet megvizsgáljuk, azaz bármilyen szűk is a keresési kritérium, mindig az adott könyvtár összes bejegyzését meg kell vizsgálni.

Ez nagyon lelassítja a rutint, főleg ha egy könyvtár sok fájlt tartalmaz. Gondoljunk csak bele, megeshet, hogy egy 10 000 fájlt tartalmazó könyvtárban két EXE fájl van. Ekkor normális esetben ha a keresési kritérium „**.EXE*”, az eredményt pillanatok alatt megkapjuk. A mi esetünkben azonban mind a 10 000 fájlhoz rendelt, EA-ban letárolt nevet le kell kérdezni és meg kell vizsgálni (legyen az a FAT meghajtók esetén `.LONGNAME`, vagy más esetben `.SHORTNAME`), hogy vajon megfelel-e a „**.EXE*” kritériumnak. A keresési idő nagyságrendekkel nagyobb lesz.

Ennek megoldására, vagy legalábbis felgyorsítására került implementálásra a most bemutatásra kerülő módszer. Kihasználandó, hogy az OS/2-es alkalmazások többszálúak is lehetnek, illetve azt, hogy megeshet, hogy a *FindFirst* és *FindNext* meghívása közt sok idő is eltelhet, a VLFN daemonja a *FindFirst* beérkezése után egy új szálat indít, amely a háttérben elkezdi keresni az összes, a kritériumoknak megfelelő bejegyzést, és ezekből egy listát készít. Így a később érkező *FindNext* összes feladata az, hogy a lista elejéről egy eredményt levegyen, és az visszaadja a hívónak, feltéve hogy már van ott valami.

Azáltal, hogy a tulajdonképpeni keresést párhuzamosítjuk, a *FindNext*-ek szinte rögtön eredménnyel térnek vissza, csak a *FindFirst* marad érezhetően lassabb, hiszen annak az első, a kritériumoknak megfelelő fájllal kell visszatérnie, azaz mindenképpen meg kell várnia, hogy a párhuzamosan futó szál legalább egy eredményt megtaláljon.

3.4.4. Az LFN_EA unit

Mint az már említésre került, a VLFN erősen alkalmazza a kibővített attribútumokat a fájlokhoz és könyvtárakhoz rendelt másodlagos fájlnevek eltárolására.

Az OS/2 alatt minden fájlhoz vagy könyvtárhoz csatolható egy vagy több kibővített attribútum, összesen maximum 64 KByte-os méretben. Minden attribútumnak egyedi neve van, és bármilyen adatot tartalmazhat.

Vannak a rendszer által használt, foglalt nevek, mint például a `.ICON`, melyben az alkalmazás ikonját szokás eltárolni, vagy például a `.COMMENTS`, melyben bármilyen megjegyzés elhelyezhető.

A kibővített attribútumoknak típus is adható. A leggyakrabban használt típusok a következők:

- `EAT_BINARY`: Bináris adat, bármi lehet.
- `EAT_ASCII`: Valamilyen szöveg, melynek hosszát az első duplaszó adja meg.

- EAT_ICON: A fájl vagy könyvtár ikonja.
- EAT_ICON1: A könyvtár azon ikonja, mely a megnyitott állapotot mutatja.
- EAT_BITMAP: Bármilyen egyéb kép.

Mivel a VLFN csak fájlneveket fog eltárolni, kizárólag az EAT_ASCII típust használjuk majd. Az állományhoz vagy könyvtárhoz társított EA-kat csak egyben lehet megkapni, az így kapott adathalmazból az operációs rendszerben definiált struktúrák alapján kell megtalálni a kívánt attribútumot.

Az EA-kat két módon lehet lekérdezni. Az egyik a DosQueryFileInfo, amellyel egy már megnyitott fájl EA-jait lehet lekérdezni, míg a másik a DosQueryPathInfo, mely az előbbivel ellentétben nem egy fájl-azonosítót vár, hanem egy fájl- vagy könyvtárnevet, azaz ennek nem feltétele hogy a fájl meg legyen nyitva.

Habár első ránézésre a második alkalmasabbnak látszik (hiszen az könyvtárokra is alkalmazható), mind a kettőt használni kell. A DosQueryPathInfo ugyanis nem tudja lekérdezni az olyan fájlok EA-ját, amelyik egy DOS-os alkalmazás által épp meg van nyitva. Ilyenkor nekünk kell megnyitni a fájlt a DosOpen segítségével, a lehető legkevesebb megszorítással, azaz csak olvasásra, minden más megszorítás nélkül. Ha a megnyitás így sikerül, akkor az így kapott fájl-azonosítóra már meghívható a DosQueryFileInfo.

A két API együttes alkalmazásával tehát szinte minden esetben megkapható a letárolt EA. Ezek kezelésére készült az LFN_EA unit, ami egyenesen a VLFN kiszolgálására készült, így nem is kezel más EA típust, csak az EAT_ASCII-t. Ennél fogva összesen két eljárása van, ami kívülről használható:

```
Function Get_ASCII_EA(pszName:string; EAname:string):String;
Function Set_ASCII_EA(pszName:string; EAname:string;
                           EAValue:string):Boolean;
```

Az első az, amely a fentebb említett két API segítségével megpróbálja lekérdezni az első paraméterben megadott fájl második paraméterben megadott névvel rendelkező EAT_ASCII típusú attribútumát.

A második az EA beállítására használható, és a fentiekben bemutatotthoz hasonló módon használja a DosSetFileInfo és DosSetPathInfo API-kat.

3.4.5. Az LFN_Cache unit

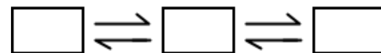
Természetesen ha sok-sok EA-t kell lekérdezni vagy beállítani egymás után, az eltarthat egy ideig. A VLFN pedig szinte minden fájl-eléréshez lekérdezi legalább egy fájl EA-ját, nem is beszélve arról, amikor fájlok keresése folyik.

Szükséges volt tehát az EA-k lekérdezésének gyorsítása, amihez kézenfekvő módszer az EA-k pufferelése, cache-elése. Ez azt jelenti, hogy minden már megismert EA-t eltárolunk a memóriában, így ha legközelebb újra arra kíváncsi a program,

nem kell megint beolvasni azt a merevlemezről, gyorsan előkereshető a memóriából is.

Az LFN_EA unit használja az LFN_Cache unitot, így minden EA lekérdezés előtt megvizsgálja, hogy vajon szerepel-e az adott fájl vagy könyvtár EA-ja a cacheben. Ha szerepel, el sem végzi a lekérdezést, ezzel gyorsítva a feldolgozást. Ha nem szerepelt még, akkor a lekérdezés elvégzése után kibővíti a cache-t az eredménnyel.

Az első implementáció egy egyszerű kétirányba láncolt rendezett lista volt, melyben a fájlnevek és a hozzájuk rendelt EA-k szerepeltek (lásd a 3.4. ábrát). Ez azonban nem bizonyult hatékonynak, sőt, kb. 150 fájl után az EA-k elérése még lassabb lett, mint a cache nélkül.



3.4. ábra. Kétirányba láncolt lista

A végső megoldás egy sokkal bonyolultabb, de nagyon hatékony faszerkezet lett (lásd a 3.5. ábrát és a struktúra-definíciókat az 54. oldalon). Minden lehetséges meghajtóhoz egy ilyen faszerkezetet rendelünk (lásd az `EACache` nevű tömböt). A faszerkezet csomópontjaiban `EACacheTree` típusú struktúrák szerepelnek, melyek végső soron egy láncolt listát és egy a szülőre mutató mutatót tartalmaznak. A láncolt lista tartalmazza az adott könyvtárban szereplő bejegyzéseket.

Minden egyes könyvtárbejegyzés a listában `EACacheNames` típusú struktúra, mely az adminisztratív mezőkön kívül tartalmazza a bejegyzés nevét és a hozzá tartozó EA-kat leíró struktúrára mutató mutatókat. Az egyes EA-kat az `EACacheValueList` struktúra tartalmazza.

Ez a megoldás igen sok pozitívumot hordoz magával, mint például:

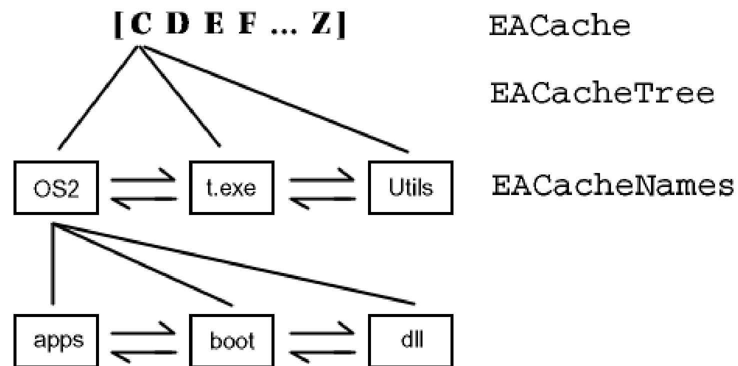
- Az adatok ugyanúgy vannak szervezve, mint a könyvtárszerkezetek a háttértáron, azaz egy elem megkeresése ugyanannyi lépést igényel, mint a merevlemezen, de itt mindez a memóriában történik, így nagyságrendekkel gyorsabb.

- Az EA-k értékei nemcsak a faszerkezetbe vannak beláncolva (melynek gyökerei az `EACache` tömb egyes elemei), hanem egy külön láncolt listának is részei (`CachedItemsH` fejjel és `CachedItemsE` véggel). Ez a lista arra használható, hogy a cache méretét limitáljuk. Amennyiben az éppen használt EA-t mindig a `CachedItems` lista elejére mozgatjuk (figyeljük meg, hogy ez nem befolyásolja a faszerkezetet!), ha a cache túl nagyra nőne (az aktuális méretet mindig a `CacheSize` változó tartalmazza), elég a lista végéről törölni néhány elemet, és ezzel a legrégebben használt elemeket fogjuk a cache-ből kitörölni. Természetesen ekkor a faszerkezetet is módosítani kell, de ezt könnyen megtehetjük, hiszen minden `CachedItems`-beli elem tartalmaz egy mutatót a szülőjére, és ezeket követve szükség esetén a faszerkezetben egészen a gyökérig is

eljuthatunk. A módszerrel tehát megtehetjük, hogy mindig csak a leggyakrabban használt elemeket tároljuk a listában.

- A faszerkezet egyes csomópontjai egy-egy könyvtárat ábrázolnak. A csomópontban levő láncolt lista (az `EACacheNames`) az adott könyvtár tartalmát jelzi. Ezt a láncolt listát is használhatjuk olyan módon, hogy az utoljára használt elemet mindig a lista elejére mozgatjuk. Ezzel a módszerrel a láncolt listák mindig használati gyakoriság szerint lesznek rendezve, ami felgyorsítja a gyakran használt fájlok és könyvtárak megtalálását.

Mindezen pozitívumoknak azonban a bemutatott bonyolult adatszerkezet az ára, és vele egy bonyolultabb programkód is, mely képes minden adminisztrációt elvégezni.



3.5. ábra. A cache szerkezete

```
Type pEACacheValueList=^EACacheValueList;
     pEACacheNames = ^EACacheNames;
     pEACacheTree = ^EACacheTree;

     EACacheNames = record
       Name : string;
       EA : array[0..1] of pEACacheValueList;
       Owner, Child : pEACacheTree;       { Vertical   }
       Prev,Next : pEACacheNames;         { Horizontal }
     end;

     EACacheTree  = record
       Parent: pEACacheNames;   { Up level name }
       Drive : Byte;            { Valid if Parent=Nil}
       NamesH : pEACacheNames;  {Head of list}
       NamesE : pEACacheNames;  {End of list}
     end;

     EACacheValueList=record
       Value : string;
       Status : FileStatus4;
       Parent  : pEACacheNames;
       EAType : Byte;  { 0 = ShortName, 1 = LongName }
       Prev,Next:pEACacheValueList;
     end;

var EACache : array [0..25] of pEACacheTree;
    CachedItemsH,     { linked list for cached values }
    CachedItemsE : pEACacheValueList;
    CacheSize : ULong;
```

4. fejezet

A VLFN üzembehelyezése

Amennyiben az előző fejezetben ismertetett kódokat lefordítjuk, megkapjuk a VLFN négy összetevőjét, melyek a VLFN.SYS, a VLFNMEM.SYS, a VLFND.EXE és az LFN.COM. Az üzembehelyezéshez a következőket kell elvégezni:

- A *CONFIG.SYS* fájlt ki kell bővíteni a következő két sorral (feltételezzük, hogy a fájlok a `C:\VLFN` könyvtárban vannak):

  ```
  device=C:\VLFN\VLFNMEM.SYS
  device=C:\VLFN\VLFN.SYS
  ```

- A *STARTUP.CMD* fájlt is ki kell egészíteni, mégpedig a következő sorral:

  ```
  start /min C:\VLFN\VLFND.EXE
  ```

  Ez minden indításkor minimalizált állapotban fogja elindítani a VLFN daemont.

Ezek után a gépet újra kell indítani, hogy a SYS fájlok betöltődjenek és elinduljanak. Ha minden jól ment, a rendszer újraindítása problémamentesen megtörténik, és a VLFN készen áll a kiszolgálásra.

Egy DOS ablak elindítása után, abban az LFN.COM-ot elindítva láthatóvá válnak a hosszú fájlneves könyvtárak és fájlok egyedi rövid fájlnevekkel!

A VLFN felépítése miatt még az is megoldható, hogy csak egyes DOS ablakokban legyenek láthatóak a hosszú fájlnevek, míg más ablakokat a régi módon kezeljen a rendszer. Ehhez csak annyit kell tenni, hogy néhány ablakban el kell indítani az LFN.COM-ot, míg más ablakokban nem.

5. fejezet

Összefoglaló

A kitűzött célt tehát sikerült elérni, a hosszú névvel rendelkező fájlok is elérhetővé váltak a DOS ablakokban. A VLFN megoldásának előnye, hogy végső soron egy általános kommunikációs utat nyit a VDM-ek és egy OS/2-es alkalmazás közt, így sok minden másra is használható. Csak hogy a hosszú fájlneveknél maradjunk, könnyen kibővíthető úgy, hogy a Microsoft Windows DOS-os programok számára használható hosszú fájlneves kiterjesztéseit is emulálja, azaz például a DOS alatt futó újabb tömörítő programok is tudnának olyan tömörített állományokat készíteni, melyekben a fájlok eredeti, hosszú nevükkel szerepelnének.

Vannak azonban még hiányosságai és hibái is. Habár az egyszerűbb DOS-os alkalmazásokkal együttműködik, a VDM alatt futtatott Windows 3.11 (melynek az OS/2-be integrált, illetve ahhoz módosított változatának neve *WinOS/2*) már nem mindenhol tűri meg a VLFN-t. Egyes gépeken fut, másokon azonban nem indul el a WinOS/2, ha az LFN.COM aktív. Ennek valószínűleg az az oka, hogy a nagyobb memóriaigényű DOS-os alkalmazások arra kényszerítik a DOS kernelt, hogy a kevésbé fontos részeit feláldozza, és majd az alkalmazás befejeztével azt a merevlemezről visszatöltse. Ez természetesen azt vonja maga után, hogy az LFN.COM indításakor elvégzett átalakításaink elvesznek.

Másik probléma, amit az EA pufferelése hoz magával, hogy ha egy EA már bekerült a pufferbe, mindig a pufferbeli érték lesz felhasználva, akkor is, ha közben egy másik OS/2-es program ,,titokban" megváltoztatja azt. Ez valószínűleg megoldható lesz az OS/2 egy speciális részének, a *SES*-nek (Security Enabling Services) telepítésével, melynek használatával az egyes fájlrendszer-beli események bekövetkeztéről is tudomást szerezhet programunk.

Van tehát még mit javítani rajta, habár a kód alapvetően már ellátja feladatát, és használhatónak nevezhető.

5.1. Felhasznált programok, felhasznált irodalom

A program fejlesztése során a következő alkalmazások és fejlesztői könyvtárak kerültek felhasználásra:

- Virtual Pascal v2.1 build 243 (*www.vpascal.com*)
- Microsoft Macro Assembler
- Borland Turbo Assembler
- IBM OS/2 Developer's Toolkit version 4.5
- IBM Developer Connection Device Driver Kit
- Interactive Disassembler v4.04 (IDA) (*www.datarescue.com*)
- IBM Interactive Code Analysis Tool (ICAT) for OS/2
- IBM OS/2 Kernel Debugger
- x86/MS-DOS Interrupt List, Release 60

A következő könyvek szintén sokat segítettek a fejlesztésben:

- Hargittai Péter, Kaszanyiczki László: *A DOS titkai*, LSI Oktatóközpont, 1993 (ISBN 963 577 058 8)
- Martin C. Sullivan: *Secrets of the OS/2 Warp Masters*, John Wiley & Sons, Inc., 1996 (ISBN 0-471-13171-7)

A diplomamunka megszerkesztéséhez a *MicroPress VTeX v7.53* TEX-fordító OS/2-es változatát, és Bujdosó Gyöngyi, Fazekas Attila: *TEX Kezdőlépések* (Tertia kiadó, ISBN 963 85129 5 4) könyvét használtam.

5.2. Köszönetnyilvánítás

Szeretnék köszönetet mondani **Dave Evans**-nek az ICAT-tel és a Kernel Debuggerrel kapcsolatban nyújtott segítségéért, **Henk Kelder**-nek amiért elküldte a kibővített attribútumokat kezelő programjának, az *EA-Browser*-nek a forráskódját, **Daniela Engert**-nek és **Timur Tabi**-nak a VDD-beli ugró-tábla kezdeti problémáinak megoldásában nyújtott segítségéért, **Joe Nord**-nak a VDM címeinek FLAT címekké alakításában nyújtott segítségéért és végül, de nem utolsó sorban témavezetőmnek, **Dr. Bölcskei András**-nak amiért elvállalta a témát, és felügyelete alatt ennek fejlesztésével foglalkozhattam.