Vývoj ovladačů jádra

Cílem této knihy není pouze poskytnout čtenářům teoretické informace o vnitřním uspořádání a fungování jádra operačních systémů rodiny NT, ale také ukázat, jak těchto znalostí prakticky využít. Z tohoto důvodu v sobě některé kapitoly zahrnují i ukázkové zdrojové kódy ovladačů jádra.

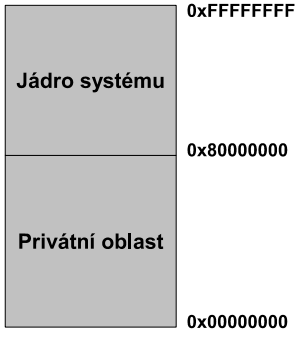
Tato kapitola si klade za cíl poskytnout takové informační pozadí, abyste ukázkovým zdrojovým kódům rozuměli, uměli si je přeložit do binární podoby a vyzkoušet. Dozvíte se podrobnější informace o tom, jak se ovladače liší od běžných aplikací, jaké prostředí pro jejich programování Microsoft nabízí a jak probíhá komunikace mezi ovladačem a kódem běžícím v uživatelském režimu.

Tato kapitola podává informace neformálním způsobem. Širší souvislosti se dozvíte v dalších částech knihy.

V závěru naleznete ukázkový zdrojový kód jednoduchého ovladače jádra, který demonstruje praktickou aplikaci znalostí, jež kapitola obsahuje.

Co je to ovladač

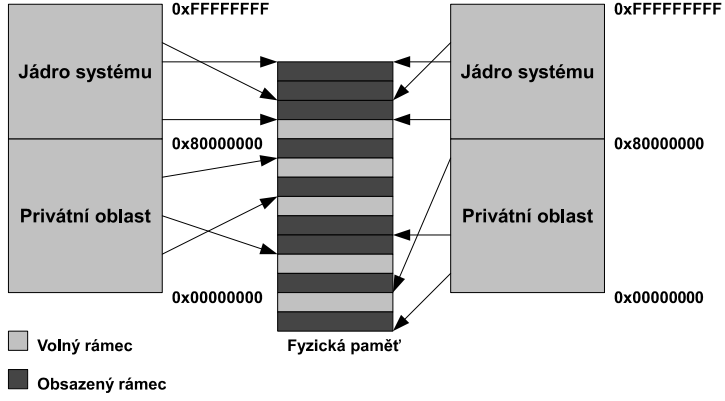
Již víte, že ovladače se na disku nacházejí ve spustitelných souborech formátu PE, většinou s příponou `.sys`. Na rozdíl od běžných aplikací, kód ovladačů není vykonáván v separátních virtuálních adresových prostorech, ale sdílí prostor společně s jádrem operačního systému. Přesněji, tento adresový prostor je namapován do prostoru každého běžícího procesu. Situaci vidíte na obrázku 3.1.



Obrázek 3.1: Zjednodušená struktura virtuálního adresového prostoru procesu

Privátní část adresového prostoru procesu (tedy oblast, kam ostatní procesy nemají přímý přístup) se nachází mezi adresami 0x00000000 a 0x7FFFFFFF. Na adresy 2 GB a vyšší je do všech

adresových prostorů namapována oblast jádra, která však z uživatelského režimu není přístupná. Naopak privátní části adresových prostorů z režimu jádra přístupné jsou. Obrázek 3.2 ukazuje mapování několika virtuálních adresových prostorů do fyzické paměti. Vidíte na něm, že horní část adresových prostorů je pokaždé namapována na stejné fyzické adresy.



Obrázek 3.2: Mapování několika virtuálních adresových prostorů do fyzické paměti

Upozornění: Zde načrtnutá struktura adresového prostoru procesu platí pro většinu 32bitových konfigurací a má pouze ilustrační charakter. Na některých konfiguracích je počátek prostoru jádra posunut až na adresu 3 GB (0xC0000000) a u 64bitových verzí operačního systému vypadá struktura celého prostoru úplně jinak. Bližší informace naleznete v kapitole 9, která se věnuje správě paměti.

Ovladače tedy nemají k dispozici žádný speciální kontext procesu, kterému by systém předával řízení, když by potřeboval vykonat jejich kód. Ten vykonává vlákno, které je zrovna naplánováno na procesoru. Jejich kód a data (proměnné) jsou namapovány ve virtuálním adresovém prostoru každého procesu na stejném místě. Z tohoto důvodu na konkrétním kontextu procesu (a vlákna) nezáleží.

Existuje však několik pravidel, pomocí kterých dokážete určit, v kterém kontextu kód ovladače poběží:

- Přechod z uživatelského režimu do režimu jádra v rámci systémového volání nemění kontext vlákna ani procesu.
- Kód vykonávaný některým z pracovních vláken (jedná se většinou o asynchronní zpracování odpovědi hardware na dřívější požadavek) běží vždy v kontextu procesu System.
- Obsluha výjimek a přerušení probíhá v kontextu vlákna, které se nacházelo na procesoru v okamžiku, kdy k příslušné události došlo.

Ovladačům většinou nezáleží na tom, v kontextu jakého procesu se jejich kód vykonává. Ve výjimečných případech se však mohou přepnout do libovolného běžícího procesu. Slouží k tomu funkce `KeAttachProcess`, `KeStackAttachProcess`, `KeUnstackDetachProcess` a `KeDetachProcess` exportované hlavním modulem jádra (`ntoskrnl.exe`). Tyto rutiny mění pouze kontext procesu; aktuální vlákno si při volání `KeAttachProcess` či `KeStackAttachProcess` „přivlastní“ zadaný proces

a ke svému pravému vlastníkovi se vlákno vrátí přes `KeDetachProcess` či `KeUnstackDetach Process`.

`KeAttachProcess` se od své zásobníkové varianty (`KeStackAttachProcess`) liší v tom, že pokud vlákno „navštívilo" cizí proces, musí se před návštěvou dalšího cizího procesu vrátit do kontextu svého vlastníka. Jinými slovy, mezi dvěma voláními `KeAttachProcess` se musí nacházet volání `KeDetachProcess`. Pro druhou dvojici funkcí toto neplatí. Volání však musí být párová – na každé volání `KeAttachProcess` (resp. `KeStackAttachProcess`) musí existovat volání `KeDetachProcess` (resp. `KeUnstackDetachProcess`).

Poznámka: Jak uvádí druhá kapitola, pro ochranu prostředí ovladačů slouží proces System. Jak ale vyplývá z odrážek výše, ovladače nemohou běžet pouze v kontextu tohoto procesu.

Další rozdíl mezi ovladači a běžnými aplikacemi skrývá práce s pamětí. Aplikace vidí všechnu paměť virtuálně – přistupují na různé virtuální adresy a je na operačním systému, aby zajistil existenci potřebných mapování do fyzické paměti. Pokud například program přistupuje do stránky, která se nachází ve stránkovacím souboru na disku, systém tuto stránku nahraje do fyzické paměti a vytvoří potřebné mapování. Tyto operace „v zákulisí" jsou pro aplikaci (až na případné malé časové zpoždění) neviditelné.

Ovladače také pracují téměř výhradně s virtuální pamětí. Jejich programátoři si ale musí uvědomovat, co se za virtuálními adresami skrývá. Za určitých okolností si totiž výpadek stránky (nastává, pokud požadovaná stránka virtuální paměti není mapována do fyzické paměti) nelze dovolit.

K takové situaci dochází například při obsluze přerušení. Představte si, že při obsluze přerušení disku dojde k výpadku stránky. Procesor vyvolá nové přerušení s číslem 0xE, které informuje systém, že nastal výpadek stránky. Operační systém se pokusí chybějící stránku nahrát ze stránkovacího souboru. Protože se stránkovací soubor nachází na pevném disku, jádro toto zařízení požádá o načtení příslušných dat. Jakmile disk splní požadavek, vyvolá další přerušení. Nezapomeňte, že v našem příkladu došlo k výpadku stránky právě při obsluze přerušení disku.

Takovýmto způsobem by mohlo snadno dojít k zatuhnutí celého systému. Proto Windows obsahují bezpečnostní opatření, která jejich běh preventivně ukončí, pokud nějaký ovladač udělá akci, jejíž provedení by za daných podmínek mohlo ohrozit chod celého systému a způsobit i poškození hardware. Takovou akcí je třeba výpadek stránky při obsluze drtivé většiny přerušení.

Z předchozích odstavců vyplývá, že za určitých okolností si programátor ovladače musí být jist, že k výpadu stránky `nemůže` dojít. Proto ovladače mohou alokovat a používat paměť dvou druhů:

- *Paměť stránkovaného fondu (PagedPool)* sestává ze stránek virtuální paměti, které mohou být uloženy na disk do stránkovacího souboru. Tento druh paměti se tedy chová úplně stejně jako privátní část adresového prostoru procesu uživatelského režimu.
- *Paměť nestránkovaného fondu (NonPagedPool)* sestává ze stránek virtuální paměti, které operační systém za žádných okolností neodloží do stránkovacího souboru na disk. Při přístupu na tyto adresy k výpadku stránky *nemůže* dojít. Paměť z nestránkovaného fondu byste měli používat s rozvahou. Využití velkého množství fyzické paměti pro účely tohoto fondu může negativně ovlivnit výkon celého systému, protože narušuje mechanismy virtuální paměti – ubírá fyzickou paměť, kterou je možné využít k načtení bloků ze stránkovacího souboru.

Upozornění: Výpadek stránky nastává, kdykoliv procesor není schopen určit, na jakou fyzickou adresu má požadovanou virtuální adresu přeložit. Tedy i v případě přístupu na adresu, na které není alokována žádná paměť. Před takovými výpadky stránky vás neuchrání ani nestránkovaný fond. Přístup na neplatnou virtuální adresu svědčí o tom, že někde v programu je chyba. U odladěného kódu by k takovému chování docházet nemělo. Pokud se váš ovladač chová správně, k výpadku stránky při přístupu do oblasti nestránkovaného fondu dojít opravdu nemůže.

Tip: Windows umožňují alokovat paměť i z dalších oblastí. Dokumentovány jsou fondy `PagedPoolMustSucceed` a `NonPagedPoolMustSucceed`, pro které systém vyhrazuje velmi malou oblast virtuální paměti (například 64 kilobajtů). Charakteristickým znakem těchto fondů je, že každá alokace musí proběhnout úspěšně. Pokud se tak nestane – například proto, že ve fondu není dostatek volné paměti – běh systému skončí modrou obrazovkou smrti. Ovladače a jádro by měly používat paměť z těchto zdrojů, pouze pokud je neúspěch alokace neslučitelný s dalším během systému.

Posledním velkým rozdílem mezi ovladači a aplikacemi běžícími v uživatelském režimu je moc, kterou nad systémem zástupci těchto kategorií mají. Na rozdíl od normálního programu ovladač může:

- Přímo komunikovat s periferními zařízeními.
- Ovlivňovat chování systému jako celku a dočasně narušovat základní mechanismy, mezi které patří plánování vláken na procesoru a fungování bezpečnostního modelu.

Na druhou stranu chyba v kódu ovladače způsobí pád celého operačního systému, kdežto chyba, kterou uděláte při programování běžné aplikace, většinou skončí násilným ukončením příslušného procesu ze strany Windows. Programování ovladačů je v tomto ohledu mnohem náročnější a dostupných nástrojů pro jejich pohodlné ladění také není mnoho.

Prostředí pro programování

Microsoft pro vývoj ovladačů poskytuje zdarma balík Windows Driver Kit (WDK), ve kterém najdete překladač, potřebnou dokumentaci a řadu užitečných nástrojů. WDK neobsahuje žádné sofistikované vývojové prostředí, ale pro kompilaci ovladačů z příkazové řádky, kterýžto způsob bude používán i v této knize, plně postačuje.

Ovladače můžete vytvářet i v příjemném prostředí Microsoft Visual Studia. I pak budete ale potřebovat WDK kvůli knihovnám a hlavičkovým souborům, kde jsou deklarovány konstanty a rutiny, jež ovladače jádra mohou používat.

Jak přeložit ovladač

Krom zdrojového kódu je k překladu potřeba dalších dvou souborů. Soubor MAKEFILE obsahuje obecné informace pro překladač a jeho obsah odkazuje na výchozí soubor s těmito údaji. Tento odkaz je zapsán následovně:

```
!INCLUDE $(NTMAKEENV)\makefile.def
```

Všechny informace týkající se přímo vašeho ovladače, jako například seznam souborů se zdrojovým kódem či seznam knihoven, se nachází v souboru SOURCES, jehož formát vidíte na výpisu 3.1.

Výpis 3.1: Formát souboru SOURCES

```
TARGETNAME=<jméno>
TARGETTYPE=<typ_výsledného_souboru>
SOURCES= <seznam_souborů_se_zdrojovým_kódem>
INCLUDES= <adresáře_s_hlavičkovými_soubory>
```

Jedná se v podstatě jen o přiřazení hodnot do proměnných prostředí, které se pak používají při překladu. Výpis 3.1 obsahuje pouze ty nejdůležitější z nich. Jejich význam a formát hodnot shrnuje tabulka 3.1.

Tabulka 3.1: Formát a význam několika základních proměnných v souboru SOURCES

Proměnná	Formát hodnoty	Význam pro překladač
TARGETNAME	Název výsledného souboru (bez přípony)	Udává název souboru (bez přípony), který linker vytvoří z přeloženého zdrojového kódu
TARGETTYPE	DRIVER, LIBRARY nebo PROGRAM	Určuje typ výsledného spustitelného souboru. Pro ovladače platí hodnota DRIVER a pro aplikace běžící v uživatelském režimu PROGRAM. Pokud chcete vytvořit knihovnu DLL, použijte hodnotu LIBRARY.
SOURCES	Seznam souborů. Oddělovačem je mezera. Pokud seznam zasahuje do více řádků, musí na konci každého z nich být zpětné lomítko „\“	Seznam všech souborů, které obsahují zdrojový kód ovladače.
INCLUDES	Seznam adresářů. Formát je stejný jako u proměnné SOURCES.	Seznam adresářů, ve kterých bude překladač vyhledávat hlavičkové soubory. Adresáře s hlavičkovými soubory WDK se prohledávají automaticky.

Spuštění prostředí překladače a následný překlad zdrojových kódů ovladače můžete provést následovně:

- V nabídce **Start** v sekci **Programy** (ve Windows Vista a Windows 7 se tato položka jmenuje **Všechny programy**) vyberte položku **Windows Driver Kits**.
- Zobrazí se seznam všech verzí WDK, které máte nainstalovány. Vyberte tu nejaktuálnější – v době psaní této kapitoly se jednalo o **WDK 7600.16385**.
- Prostředí překladače se skrývá pod volbou **Build Environments**.
- Nyní vyberte operační systém, pro který chcete ovladač zkompilovat. Novější verze OS umožňují ovladačům využívat nové funkce. Na druhou stranu i ovladač přeložený pro Windows 7 můžete bez problémů rozchodit na Windows XP, pokud nevyužívá specifik novější verze.

  Pokud chcete přeložit ovladač například pro Windows 7, vyberte položku **Windows 7**.
- Nyní je třeba vybrat architekturu procesoru. Překladač WDK dokáže kompilovat do instrukční sady procesorů x86, AMD64 (x64) a Ithanium (ia64). Architektura x86 implikuje 32bitový operační systém. Pokud chcete ovladač používat na 64bitových Windows, zvolte x64.

Dále se musíte rozhodnout, zda chcete svůj ovladač přeložit v prostředí tzv. `Free build`, nebo `Checked build`. Prostředí `Free build` je ekvivalentní modu `Release` z aplikace Microsoft Visual Studio; překladač provádí všechny dostupné optimalizace kódu a ignoruje makra jako `ASSERT`, `KdPrint` či `KdPrintEx`. Prostředí `Checked build` slouží pro testování. Překladač neprovádí optimalizace, které činí kód binárky méně čitelným pro disassemblery, a makra `ASSERT`, `KdPrint`, `KdPrintEx` a další provádějí svoji normální činnost: `ASSERT` v případě vyhodnocení zadaného výrazu na FALSE pošle zprávu debuggeru s informacemi o tom, na jakém řádku jakého souboru zdrojového kódu k selhání došlo, `KdPrint` a `KdPrintEx` se překládají na volání `DbgPrint` a `DbgPrintEx`. Prostředí dále definuje symbol DBG na hodnotu 1, čehož lze využít při podmíněném překladu.

> **Tip:** možnostem ladění ovladačů včetně použití maker `ASSERT`, `KdPrint`, `KdPrintEx`, `DbgPrint` a `DbgPrintEx` se věnuje část „Několik poznámek k ladění ovladačů" níže v této kapitole.

Pokud ovladač chcete testovat, použijte nastavení `Checked build`. Pokud si myslíte, že již může být nasazen do „ostrého provozu", přeložte jej pomocí `Free build`.

Rozhodnete-li se například pro architekturu x64 a chcete-li do ovladače přilinkovat i dodatečné ladicí informace, zvolte položku **x64 Checked Build Environment**.

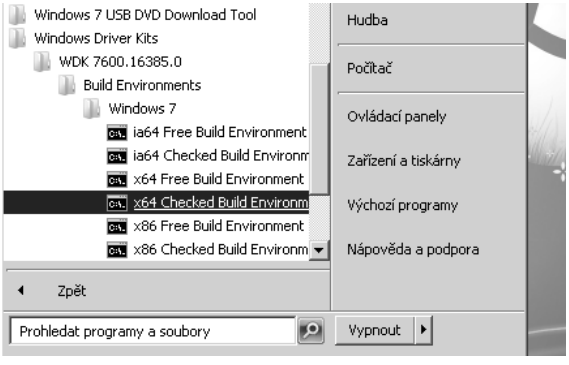
Dosavadní postup vidíte na obrázku 3.3.

- Spustí se konzole Příkazového řádku nastavená pro účely překladače WDK. Nyní se pomocí standardních příkazů přesuňte do adresáře, kde se nachází soubory MAKEFILE a SOURCES vašeho ovladače.
- Příkazem **build** zahájíte překlad a následné linkování. Překladač a linker vypisují průběžný stav do konzole. Chování příkazu lze ovlivnit mnoha parametry příkazové řádky. Některé možnosti ukazuje tabulka 3.2.

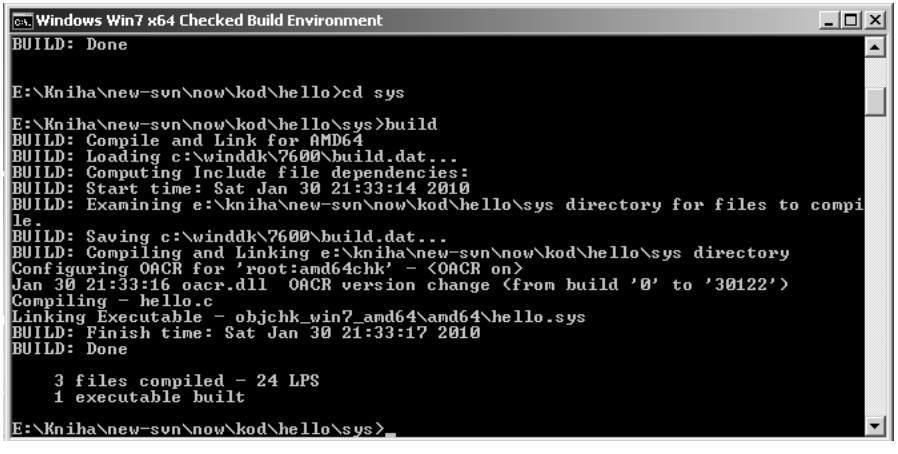
Tabulka 3.2: Ovlivnění chování příkazu build pomocí příkazové řádky

Příkaz	Popis
/c	Před zahájením překladu odstraní všechny existující objektové soubory.
/C	Před zahájením překladu odstraní soubory statických knihoven .lib.
/g	Při výpisu varování, chyb a výsledků překladu jednotlivé aspekty odlišuje barvami. Chyby jsou tisknuty červeně, varování žlutě a počet vytvořených spustitelných souborů zeleně.
/h	Nevypisuje podrobné informace o překladu do konzole.
/w	Do konzole jsou zobrazována i varování.
/T	Zobrazí úplný strom závislostí.
/$	Zobrazí úplný strom závislostí hierarchicky.

- Pokud překlad a linkování proběhly úspěšně, vytvoří se ve složce se souborem MAKEFILE nový adresář, v němž najdete i spustitelný soubor vašeho ovladače. Úspěšný překlad a linkování vidíte na obrázku 3.4.
- V případě, že se při překladu či linkování vyskytly problémy (viz obrázek 3.5), překladač (resp. linker) vypíše hlášení o neúspěchu a vytvoří dva nové textové soubory. Soubor s příponou `.wrn` obsahuje varování. Ta nebrání v úspěšném dokončení překladu a linkování, ale zvlášť při programování ovladačů je žádoucí, aby se žádná neobjevovala. Do souboru s příponou `.err` překladač (resp. linker) vypíše informace o nalezených chybách. Jedná se například o špatnou syntaxi jazykových konstrukcí či hlášení o použití neexistující proměnné či konstanty.

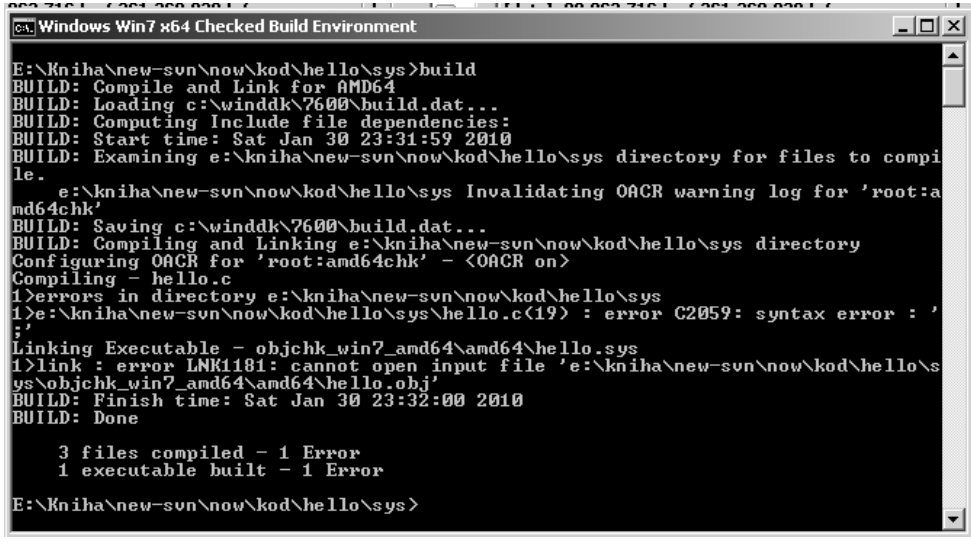


Obrázek 3.3: Výběr nastavení prostředí překladače WDK



Obrázek 3.4: Výstup při úspěšné kompilaci

Poznámka: Balík Windows Driver Kit se za dob, kdy nejnovější verze operačního systému nesla název Windows Server 2003, nesl jméno Driver Development Kit (DDK).



Obrázek 3.5: Během kompilace (linkování) byly nalezeny chyby

Tip: U ovladačů, které tvoří součást malware, se občas stává, že uniknou jejich verze určené pro ladění – tedy obsahují odkaz na soubor `.pdb` s ladicími informacemi. Tento odkaz je uložen ve formě absolutní cesty a může poskytnout cenné vodítko nejen k identifikaci typu malware (podle jména souboru ovladače to často není možné, protože může být generováno náhodně), ale i ke zjištění základních informací o autorovi. Vše závisí na tom, jaká jména autor volil pro složky, které se v odkazu nachází.

Načtení ovladače do jádra

Spustitelný soubor ovladače není možné (za běžného nastavení) načíst do jádra systému klasickým poklepáním, jak tomu je v případě běžné aplikace. Na rozdíl od spuštění programu se může jednat o relativně složitou operaci a existuje více cest, jak ji provést.

Čistý a oficiální způsob

Nejjednodušší způsob načtení ovladače do jádra spočívá ve vytvoření služby, která daný ovladač popisuje (pamatujte, Windows interně reprezentují ovladače jako služby) a jejím následném spuštění. Ukázkový kód vidíte na výpisech 3.2 a 3.3. Jedná se o techniku plně dokumentovanou a „zdvořilou" k operačnímu systému. Pravděpodobně nenajdete mnoho legitimních důvodů, proč ji nepoužít.

Pro vytvoření nové služby je nejprve nutné získat potřebná oprávnění k databázi služeb, kterou spravuje správce služeb (SCM). K tomuto účelu slouží funkce `OpenSCManagerW`, která jako své argumenty bere název počítače, k jehož databázi služeb chcete získat přístup, název konkrétní databáze a přístupová práva, která chcete získat. Pokud jako první dva parametry dostane hodnotu NULL, rutina se pokusí zajistit požadovaná oprávnění k právě používané databázi na lokálním počítači. Pro instalaci služby je nutné oprávnění SC_MANAGER_CREATE_SERVICE. Pokud se podaří

přístup získat, funkce vrátí handle databáze, kterým se bude proces prokazovat při volání dalších rutin pro komunikaci se SCM.

Po úspěšném získání oprávnění následuje vlastní vytvoření služby – volání funkce `CreateServiceW`. Tato rutina má mnoho parametrů, protože pokrývá veškeré možnosti nastavení všech typů služeb. Pro ovladače má význam druhý parametr (interní jméno), pátý parametr, který určuje typ služby (pro ovladač je rezervována hodnota SERVICE_KERNEL_DRIVER) a šestý a osmý parametr. Šestým parametrem ovlivňujete, kdy může být služba spuštěna. V ukázkovém kódu se předává hodnota SERVICE_DEMAND_START, která určuje, že ovladač popsaný touto službou bude načten do jádra „na požádání" libovolného programu s dostatečným oprávněním. Osmý parametr specifikuje jméno souboru ovladače.

Pokud volání `CreateServiceW` uspěje, službu ovladače se podařilo úspěšně nainstalovat. Nyní je třeba uklidit prostředky alokované jak při vlastní instalaci, tak při získávání přístupu k databázi služeb. Tento úkol patří rutině `CloseServiceHandle`, která je zodpovědná za uvolnění prostředků spojených s libovolným handle od služby (která vrací například `CreateServiceW`) či databáze služeb (které vzniká při úspěšném volání `OpenSCManagerW`).

Výpis 3.2: Vytvoření služby ovladače

```
BOOL scmInstallDriver (PWCHAR DriverName, PWCHAR FileName)
{
  BOOL ret = FALSE;
  SC_HANDLE hservice = NULL;

  hservice = CreateServiceW(hmanager_inst, DriverName, NULL,
SERVICE_ALL_ACCESS, SERVICE_KERNEL_DRIVER, SERVICE_DEMAND_START,
SERVICE_ERROR_NORMAL, FileName, NULL, NULL, NULL, NULL, NULL);
  ret = hservice!= NULL;
  if (ret)
    CloseServiceHandle(hservice);

  return ret;
}
```

Výpis 3.3 obsahuje zdrojový kód rutiny `scmLoadDriver`, která jako parametr vezme interní jméno ovladače a pokusí se jej načíst do jádra systému.

Pro načtení ovladače do jádra je třeba získat oprávnění pro spuštění služby, která jej reprezentuje. Aby mohla získat dané oprávnění, musí nejprve funkce `scmLoadDriver` získat přístup do databáze služeb. Tentokrát postačí oprávnění SC_MANAGER_CONNECT, které umožňuje pracovat s jednotlivými službami.

Po úspěšném připojení k databázi služeb se `scmLoadDriver` pokusí získat ke službě ovladače právo SERVICE_START pomocí volání `OpenServiceW`. Bez tohoto oprávnění není možné službu spustit.

Spuštění služby ovladače se projeví načtením příslušného souboru s příponou `.sys` do jádra. Samotnou akci provádí rutina `StartServiceW`. Tato funkce krom handle cílové služby bere i další dva argumenty, jež umožňují spouštěné entitě předat parametry obdobně jako se předávají parametry příkazového řádku při vytváření nového procesu. Ovladače tento mechanismus nepodporují, a proto na hodnotách těchto argumentů nezáleží.

Výpis 3.3: Načtení ovladače do jádra

```
BOOL scmLoadDriver (PWCHAR DriverName)
{
  SC_HANDLE hservice = NULL;
  BOOL ret = FALSE;

  hservice = OpenServiceW(hmanager_connect, DriverName, SERVICE_START);
  ret = hservice!= NULL;
  if (ret) {
    ret = StartServiceW(hservice, 0, NULL);
    CloseServiceHandle(hservice);
  }

  return ret;
}
```

Pokud již ovladač není v jádře potřeba, může být z paměti odstraněn. Postup, který demonstruje rutina `scmUnloadDriver` na výpisu 3.4 se velmi podobá předchozím; `scmUnloadDriver` nejprve získá přístup k databázi služeb, následně k službě se zadaným interním jménem a pošle jí řídící příkaz SERVICE_CONTROL_STOP, kterým požaduje okamžité zastavení činnosti. K vyslání tohoto příkazu je nutné získat k cílové entitě oprávnění SERVICE_STOP.

Výpis 3.4: Uvolnění ovladače z jádra

```
BOOL scmUnloadDriver (PWCHAR DriverName)
{
  SERVICE_STATUS ss;
  SC_HANDLE hservice = NULL;
  BOOL ret = FALSE;

  hservice = OpenServiceW(hmanager_connect, DriverName, SERVICE_STOP);
  ret = hservice!= NULL;
  if (ret) {
    ret = ControlService(hservice, SERVICE_CONTROL_STOP, &ss);
    CloseServiceHandle(hservice);
  }

  return ret;
}
```

Protože služby reprezentují vysoce privilegované aplikace a ovladače, není pro ně příkaz k zastavení činnosti závazný. Za určitých okolností by zastavení nějaké služby mohlo znamenat ohrožení pro celý systém. Jak se toto „odmítnutí“ implementuje u ovladačů, najdete na příkladu Hello World dále v této kapitole.

Odinstalování služby (funkce `scmUninstallDriver` na výpisu 3.5) spočívá pouze v získání dostatečných oprávnění k dané službě a k volání rutiny `DeleteService`.

Výpis 3.5: Smazání služby ovladače

```
BOOL scmUninstallDriver (PWCHAR DriverName)
{
  SC_HANDLE hservice = NULL;
  BOOL ret = FALSE;

  hservice = OpenServiceW(hmanager_connect, DriverName, DELETE);
  ret = hservice!= NULL;
  if (ret) {
    ret = DeleteService(hservice);
    CloseServiceHandle(hservice);
  }
  return ret;
}
```

Upozornění: Správce služeb umožňuje odstranit ze systému i službu, která je právě aktivní. Funkce `DeleteService` pouze smaže klíče registru, které vytvořilo volání `CreateService` při instalaci. Služba nedostane žádnou informaci o tom, že již v systému není nainstalována a může běžet až do restartu počítače.

Zdrojové kódy rutin uvedených na výpisech 3.2 až 3.5 naleznete na internetových stránkách knihy v projektu `drv` v souboru `scmDrivers.c`.

Web: www.jadro-windows.cz/projekty/drv

Méně známý způsob (nativní funkce NtLoadDriver)

Mezi méně známé způsoby načítání ovladače do jádra systému patří použití nativní funkce `NtLoadDriver`. Před seznámením s podrobnostmi této metody je třeba uvést několik základních fakt o nativních funkcích Windows API.

Již víte, že nativní funkce exportuje knihovna `ntdll.dll`, která má na starost komunikaci s jádrem. Na rozdíl od dokumentovaných rutin Windows API na vyšších vrstvách, které vrací nenulovou hodnotu v případě úspěchu a nulu, když daná operace selže, nativní funkce většinou pracují s návratovou hodnotou typu NTSTATUS. Jedná se o celé číslo bez znaménka, jehož význam vysvětluje tabulka 3.3.

Upozornění: Mnoho funkcí dostupných ovladačům jádra též vrací hodnotu NTSTATUS. Při zkoumání jejich významu si pečlivě přečtěte, jaké návratové kódy mohou vrátit. Rozdělení návratových hodnot může být ošidné. I když funkce vrátí hodnotu spadající do kategorie „úspěch", nemusí to znamenat, že daná operace byla úspěšně provedena. Úspěch operace v obecném případě garantuje pouze návratová hodnota STATUS_SUCCESS.

Tabulka 3.3: Význam hodnot NTSTATUS

Hodnota či rozsah hodnot	Význam
0x00000000 – 0x3FFFFFFF (úspěch)	Operace byla úspěšně provedena. Ideální případ nastává při návratové hodnotě 0, které odpovídá konstanta STATUS_ SUCCESS.
0x40000000 – 0x7FFFFFFF (informace)	Během provádění operace došlo k události, která její úspěch významně neovlivnila, ale přesto stojí za povšimnutí.
0x80000000 – 0xBFFFFFFF (varování)	Během provádění operace se vyskytl problém, který nepřímo brání jejímu dokončení. To nastává například v případě, že výsledky operace nelze zapsat do bufferu předávaného v parametru volání, protože je příliš malý.
0xC0000001 – 0xFFFFFFFF (chyba)	Při pokusu o provedení operace došlo k chybě.

Poznámka: Pro pohodlnější testování, do které kategorie určitá hodnota NTSTATUS patří, můžete využít některá z následujících maker:

- NT_SUCCESS – vyhodnotí se jako TRUE, pokud zadaná hodnota patří do kategorie „úspěch" či „informace". V ostatních případech vrátí FALSE.
- NT_INFORMATION – nabývá hodnoty TRUE právě tehdy, když zadaná hodnota patří do kategorie „informace".
- NT_WARNING – pokud hodnota spadá do kategorie „varování", vrátí TRUE. V jiném případě vrací FALSE.
- NT_ERROR – vyhodnotí se na TRUE pouze v případě hodnoty z oblasti „chyba".

Další rozdíl mezi standardními rutinami Windows API a nativními funkcemi spočívá v práci s řetězci. Standardní Windows API pracuje s tzv. *nulou ukončenými řetězci*. Takové řetězce jsou tvořeny posloupností znaků ukončenou znakem s hodnotou nula a jejich výhoda spočívá v tom, že není nutné si explicitně pamatovat jejich délku (v případě potřeby se vypočítá jako rozdíl adresy počátku řetězce a adresy koncového nulového znaku). Nativní funkce pracují s řetězci reprezentovanými strukturami UNICODE_STRING a ANSI_STRING. Tyto struktury v sobě uchovávají nejenom obsah celého řetězce, ale i jeho délku v bajtech. Protože jádro Windows pracuje téměř výhradně s řetězci ve formátu Unicode, používají se v drtivé většině případů struktury UNICODE_STRING.

Načtení ovladače pomocí volání `NtLoadDriver` lze rozdělit do několika kroků. Nejprve je třeba v registru manuálně vytvořit klíč, který bude reprezentovat službu našeho ovladače a vyplnit hodnoty `Start`, `Type` a `ImagePath`. Tyto operace dělá funkce `ntldInstallDriver` z výpisu 3.6. Odinstalování ovladače spočívá ve smazání klíče služby z registru (rutina `ntldUninstallDriver`).

Výpis 3.6: Načtení a uvolnění ovladače z jádra

```
BOOL ntldInstallDriver (PWCHAR DriverName, PWCHAR FileName)
{
  DWORD start = SERVICE_DEMAND_START;
```

```
  DWORD type = SERVICE_KERNEL_DRIVER;
  BOOL ret = FALSE;
  HKEY driverkey = NULL;
  LONG res = 0;
  UNICODE_STRING uFullFileName;
  DWORD FileNameSize = (DWORD) (wcslen(FileName) + 1) * sizeof(WCHAR);

  res = RegCreateKeyExW(serviceskey, DriverName, 0, NULL, 0, KEY_ALL_ACCESS,
    NULL, &driverkey, NULL);
  ret = res == ERROR_SUCCESS;
  if (ret) {
    _PrepareFullName(FileName, L"\\??\\", &uFullFileName);
    if (ret) {
      res = RegSetValueExW(driverkey, L"ImagePath", 0, REG_SZ,
        (PVOID)uFullFileName.Buffer,
        uFullFileName.Length + sizeof(WCHAR));
      ret = res == ERROR_SUCCESS;
      if (ret) {
        res = RegSetValueExW(driverkey, L"Start", 0, REG_DWORD, (PVOID)&start,
          sizeof(start));
        ret = res == ERROR_SUCCESS;
        if (ret) {
          res = RegSetValueExW(driverkey, L"Type", 0, REG_DWORD, (PVOID)&type,
            sizeof(type));
          ret = res == ERROR_SUCCESS;
          if (!ret)
            RegDeleteKeyW(serviceskey, DriverName);
        } else RegDeleteKeyW(serviceskey, DriverName);
      } else RegDeleteKeyW(serviceskey, DriverName);

      _FreeFullName(&uFullFileName);
    }

    RegCloseKey(driverkey);
  }

  return ret;
}
BOOL ntldUninstallDriver (PWCHAR DriverName)
{
  BOOL ret = FALSE;
  LONG res = 0;

  ret = DeleteRegistryKey(serviceskey, DriverName);
  return ret;
}
```

Po jeho úspěšném vytvoření stačí název klíče předat jako parametr nativní funkci `NtLoadDriver`. Protože tato rutina akceptuje řetězec ve formátu UNICODE_STRING, nejprve je nutné nulou ukončený řetězec Unicode (typ `PWCHAR`) do této podoby převést. Konverze se provádí pomocí procedury `RtlInitUnicodeString`, která pro zadaný nulou ukončený řetězec vytvoří strukturu UNICODE_STRING, jež jej popisuje. Potřebné definice vidíte ve výpisu 3.7. Výpis 3.8 ukazuje přímé použití `NtLoadDriver` k načtení ovladače do jádra a `NtUnloadDriver` pro jeho uvolnění.

Výpis 3.7: Definice datových typů a konstant používaných nativními funkcemi Windows API

```
#define STATUS_SUCCESS          0x00000000L
#define STATUS_UNSUCCESSFUL     0xC0000001L
typedef struct _UNICODE_STRING {
  USHORT Length;
  USHORT MaximumLength;
  PWSTR  Buffer;
} UNICODE_STRING, *PUNICODE_STRING;

typedef VOID (NTAPI *RTLINITUNICODESTRING)(PUNICODE_STRING UnicodeString,
  PWCHAR WideString);
typedef NTSTATUS (NTAPI *NTLOADDRIVER)(PUNICODE_STRING DriverName);
typedef NTSTATUS (NTAPI *NTUNLOADDRIVER)(PUNICODE_STRING DriverName);
```

Rutina `NtUnloadDriver` je párová k `NtLoadDriver`. Provádí přesně opačnou operaci. Za parametr též bere název klíče služby ovladače. Obě nativní funkce indikují úspěch či neúspěch celé operace hodnotou NTSTATUS.

Výpis 3.8: Načtení a uvolnění ovladače

```
BOOL ntldLoadDriver (PWCHAR DriverName)
{
  BOOL ret = FALSE;
  UNICODE_STRING uFullName;
  NTSTATUS status = STATUS_UNSUCCESSFUL;

  ret = _PrepareFullName(DriverName, NTLD_NAME_PREFIX, &uFullName);
  if (ret) {
    status = _NtLoadDriver(&uFullName);
    ret = status == STATUS_SUCCESS;

    _FreeFullName(&uFullName);
  }

  return ret;
}

BOOL ntldUnloadDriver (PWCHAR DriverName)
{
  BOOL ret = FALSE;
```

```
  UNICODE_STRING uFullName;
  NTSTATUS status = STATUS_UNSUCCESSFUL;

  ret = _PrepareFullName(DriverName, NTLD_NAME_PREFIX, &uFullName);
  if (ret) {
    status = _NtUnloadDriver(&uFullName);
    ret = status == STATUS_SUCCESS;
    _FreeFullName(&uFullName);
  }

  return ret;
}
```

Funkce pro práci s ovladači pomocí nativních funkcí `NtLoadDriver` a `NtUnloadDriver` naleznete na internetových stránkách knihy v projektu `drv` v souboru `ntregDrivers.c`.

Web: http://www.jadro-windows.cz/projekty/drv

Poznámka: Zde ukázaný postup je téměř ekvivalentní k práci s ovladačem pomocí SCM. Správce služeb při instalaci služby ovladače vytvoří potřebné klíče a její spuštění provádí též přes volání `NtLoadDriver`. Jediný rozdíl mezi oběma postupy tkví v tom, že při manuálním vytvoření potřebných klíčů registru se SCM o nové službě nedozví. Seznam služeb se totiž nachází v paměti procesu `services.exe` a registr slouží pouze jako jeho trvalé úložiště. SCM nekontroluje, zda nějaká aplikace manuálně přidala klíče a hodnoty odpovídající instalaci nové služby. Z tohoto důvodu je načtení ovladače jádra přímo pomocí volání `NtLoadDriver` méně viditelné a často tento postup najdete ve škodlivých programech.

Tip: Pro instalaci ovladače musí být v klíči služby přítomné buď hodnoty `Type`, `Start` a `ImagePath`, nebo `Name`, `Type` a `Start`. Pokud není položka `ImagePath` přítomna, systém předpokládá, že se soubor ovladače nachází v systémovém adresáři pod jménem `<Obsah_položky_Name>.sys`.

Méně známý způsob (nativní funkce NtSetSystemInformation)

Nativní funkce `NtSetSystemInformation` slouží k úpravě různých aspektů systému. Mezi takto konfigurovatelná „nastavení“ patří i načtení nového ovladače do jádra. Odstranění ovladače z paměti tato rutina neumožňuje. Definici funkce vidíte na výpisu 3.9, ukázku volání pak na výpisu 3.10.

Výpis 3.9: Definice nativní funkce Windows API NtSetSystemInformation

```
typedef enum _SYSTEM_INFORMATION_CLASS {
  SystemBasicInformation,
  SystemProcessorInformation,
  SystemPerformanceInformation,
```

```
  SystemTimeOfDayInformation,
  ...
  SystemExtendServiceTableInformation,  // 38
  ...
  SystemMemoryListInformation,
  SystemFileCacheInformationEx,
  MaxSystemInfoClass
} SYSTEM_INFORMATION_CLASS, *PSYSTEM_INFORMATION_CLASS;

typedef NTSTATUS (NTAPI *NTSETSYSTEMINFORMATION)
  (SYSTEM_INFORMATION_CLASS SystemInformationClass, PVOID Buffer,
    ULONG Length);
```

`NtSetSystemInformation` akceptuje tři parametry. První parametr určuje, jaké nastavení systému chce volající změnit. Druhý parametr obsahuje adresu bloku paměti s novými hodnotami a třetí parametr udává délku tohoto bloku.

Pro načtení ovladače do jádra slouží hodnota prvního parametru 36 známá též jako `SystemLoadAndCallImage`. Systém předpokládá jako druhý parametr strukturu SYSTEM_LOAD_AND_CALL_IMAGE, která se skládá pouze z řetězce Unicode reprezentovaného pomocí struktury UNICODE_STRING udávajícího název a umístění souboru ovladače. Úspěch operace se dozvíte z návratové hodnoty NTSTATUS.

Výpis 3.10: Načtení ovladače pomocí volání NtSetSystemInformation

```
BOOL ntssiLoadDriver(PWCHAR DriverName)
{
  BOOL ret = FALSE;
  UNICODE_STRING uFullName;
  NTSTATUS status = STATUS_UNSUCCESSFUL;

  ret = _PrepareFullName(DriverName, NTSSI_NAME_PREFIX, &uFullName);
  if (ret) {
    status = _NtSetSystemInformation(SystemExtendServiceTableInformation,
      &uFullName,
      sizeof(UNICODE_STRING));
    ret = status == STATUS_SUCCESS;
    _FreeFullName(&uFullName);
  }

  return ret;
}
```

Upozornění: Ovladač načtený tímto způsobem se celý nachází ve stránkované paměti a nemůže standardními cestami provádět některé běžné úkony jako například vytváření virtuálních zařízení za účelem komunikace s aplikacemi běžícími v uživatelském režimu.

Práce s ovladači pomocí utility drv

Výše popsané metody práce s ovladači jádra naleznete na internetových stránkách knihy implementované v konzolové aplikaci `drv`. Pomocí tohoto programu a příkazového řádku Windows můžete snadno instalovat vlastní ovladače a načítat je do jádra systému. Program se ovládá následujícími příkazy:

```
drv -scm install <jmeno_sluzby> <jmeno_souboru>
drv -scm load <jmeno_sluzby>
drv -scm unload <jmeno_sluzby>
drv -scm uninstall <jmeno_sluzby>
drv -ntld install <jmeno_sluzby> <jmeno_souboru>
drv -ntld load <jmeno_sluzby>
drv -ntld unload <jmeno_sluzby>
drv -ntld uninstall <jmeno_sluzby>
drv -ntssi load <jmeno_souboru>
```

První parametr udává, kterou metodu má program použít (`scm` – správce služeb, `ntld` – nativní funkci `NtLoadDriver`, `ntssi` – nativní funkci `NtSetSystemInformation`). Podle hodnoty druhého se utilita rozhoduje, jakou akci provést. Nativní API funkce `NtSetSystemInformation` podporuje pouze načtení ovladače do jádra, ostatní způsoby vyžadují před vlastním načtením vytvoření služby, která ovladač popíše.

Web: http://www.jadro-windows.cz/projekty/drv

Jednoduchý příklad: Klasické „Hello World!“

V mnoha publikacích narazíte na ukázkové příklady, pro které se vžilo označení „Hello World!“ Cílem ukázek tohoto typu je většinou nějakým zajímavým způsobem vypsat ono anglické sousloví na obrazovku. Například zobrazením textu v dialogovém okně. O tento typ ukázky nebudete ochuzeni ani v této knize, protože dobře poslouží k ilustraci základní architektury ovladače jádra.

Web: http://www.jadro-windows.cz/projekty/hello

Ovladač `hello.sys` se skládá ze tří částí – souboru MAKEFILE se standardním obsahem, souboru SOURCES s instrukcemi pro překladač balíku WDK (výpis 3.11) a souboru `hello.c`, který obsahuje vlastní zdrojový kód (viz výpis 3.12).

Překladač se ze souboru SOURCES dozví, že má přeložit zdrojový kód v souboru `hello.c` (řádek 3) jako ovladač (řádek 2) se jménem `hello` (řádek 1). Linker ovladačům automaticky do jména souboru doplní příponu `.sys`.

Výpis 3.11: Soubor SOURCES

```
TARGETNAME=hello
TARGETTYPE=DRIVER
SOURCES= hello.c
```

Soubor `hello.c` obsahuje dvě rutiny – `DriverEntry` a `DriverUnload`. `DriverEntry` je analogií k funkcím `main` a `WinMain` známých z prostředí programování běžných aplikací. Systém této funkci předá řízení během načítání ovladače do jádra. Úkolem podprogramu je provést nezbytnou inicializaci, aby po svém načtení do jádra mohl ovladač okamžitě fungovat.

Výpis 3.12: Soubor hello.c

```
#include <ntddk.h>

VOID DriverUnload(PDRIVER_OBJECT DriverObject)
{
  DbgPrint ("Good Bye\n");
  return;
}

NTSTATUS DriverEntry (PDRIVER_OBJECT DriverObject,
  PUNICODE_STRING RegistryPath)
{
  NTSTATUS status = STATUS_UNSUCCESSFUL;

  DbgPrint("HELLO WORLD!\n");
  DriverObject->DriverUnload = DriverUnload;

  status = STATUS_SUCCESS;
  return status;
}
```

Inicializace nemusí vždy proběhnout úspěšně. Proto pokud `DriverEntry` vrátí hodnotu NTSTATUS odpovídající chybě, načtení ovladače do jádra skončí nezdarem.

`DriverEntry` akceptuje dva parametry – adresu struktury DRIVER_OBJECT, která reprezentuje ovladač jako entitu v jádře a obsahuje veškeré informace a nastavení, a úplný název klíče služby, jež ovladač reprezentuje.

> **Poznámka:** Pokud ovladač dostanete do jádra pomocí výše popsané nativní funkce `NtSetSystemInformation`, rutina `DriverEntry` obdrží oba parametry nastavené na hodnotu NULL. Takto zavedený ovladač nemá žádnou strukturu DRIVER_OBJECT, která by jej v jádře reprezentovala, a ani službu, jež by jej zastupovala ve správci služeb. Navíc je celý soubor ovladače tvořen stránkovanou pamětí, jak uvádí poznámka výše.

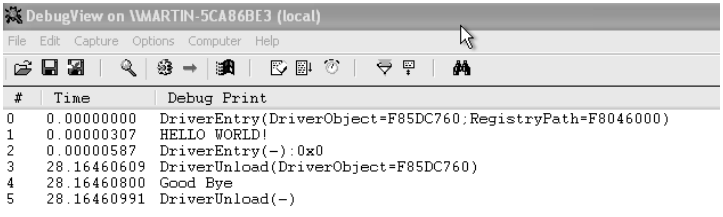
Krom ladicího výpisu „HELLO WORLD!“ provádí rutina `DriverEntry` velmi důležitou věc – umožňuje ovladač uvolnit z paměti za běhu operačního systému. Při povídání o službách jste se dozvěděli, že okolí jim může zasílat různé požadavky, kterým služby ale nemusí vyhovět. Něco podobného platí i pro ovladače. Nemusí dovolit své odstranění z paměti jádra.

Ovladač lze dynamicky uvolnit z jádra právě tehdy, když má nastavenou proceduru, které systém předá řízení během odstraňování z paměti. Tato rutina se často nazývá `DriverUnload` a její úkol je opačný k úkolu funkce `DriverEntry` – uvolnit všechny prostředky, jenž ovladač používal během své přítomnosti v jádře. Adresa této rutiny se musí nastavit do položky `DriverUnload` struktury DRIVER_OBJECT. Na rozdíl od `DriverEntry`, která má přímý vliv na úspěch zavedení ovladače do jádra, tato „uklízecí" procedura nemůže nijak ovlivnit úspěch či neúspěch jeho odstranění.

Protože inicializace ovladače `hello.sys` spočívá pouze v nastavení „uklízecí" procedury a ve vypsání známého anglického sousloví, `DriverUnload` nemá co uklízet a pouze vypíše řetězec „Good Bye".

Ovladače nemohou snadno zobrazovat textové řetězce na obrazovku například pomocí dialogových oken. Nejjednodušším způsobem je použití ladicích funkcí, mezi které patří `DbgPrint`, jež pošle zadaný řetězec debuggeru jádra, je-li v systému přítomen.

Volání `DbgPrint` lze monitorovat i s pomocí jednoduché aplikace `DebugView`, kterou naleznete na serveru `www.sysinternals.com`, nebo internetových stránkách této knihy. Pro zachytávání volání `DbgPrint` je nutné program spustit s administrátorskými právy a v menu **View** zaškrtnout položky **Capture Kernel** (zachytávat události jádra) a **Enable Verbose Kernel Output** (povolit čitelné zobrazování ladicích výpisů jádra). Po načtení ovladače `hello.sys` do jádra a jeho následném odstranění uvidíte v bílém poli programu něco podobného obrázku 3.6. K práci s ovladačem můžete využít program `drv.exe` popsaném na konci předchozího oddílu. Ukázku zavedení ovladače do jádra a jeho následného odstranění pomocí této utility vidíte na obrázku 3.7.



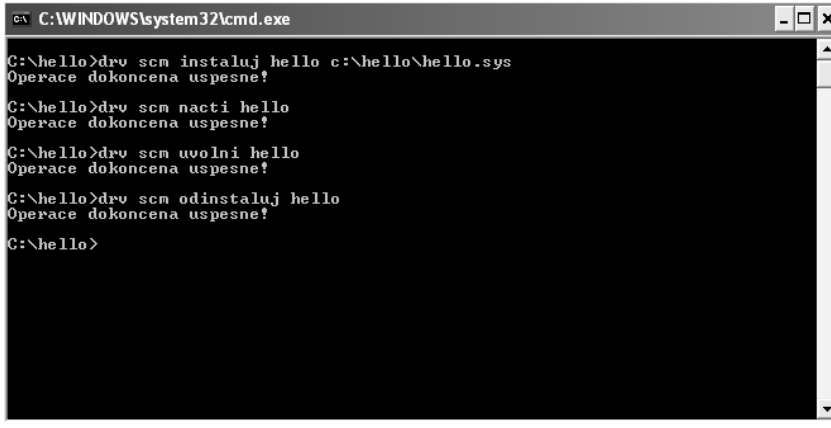
Obrázek 3.6: Projevy ovladače hello.sys v programu DebugView

Několik poznámek k ladění ovladačů

K ladění běžných aplikací většinou stačí program zvaný `debugger`, který umožňuje na specifických místech běh aplikace zastavit, krokovat či prohlížet aktuální hodnoty proměnných, což velmi pomáhá při odhalování nejrůznějších chyb. Debugger je aplikace jako každá jiná – běží v uživatelském režimu a pouze využívá podpory pro ladění, kterou systém Windows disponuje.

Pro ladění ovladačů obyčejný debugger nestačí, protože ovladače běží v režimu jádra a některé akce, které provádějí, snadno kontrolovat nelze. Metody na odhalování chyb ale existují.

Mezi nejúčinnější metody patří prevence. Než vámi vytvořený ovladač pustíte do jádra operačního systému, pečlivě si pročtěte celý zdrojový kód a přemýšlejte, za jakých okolností bude systém jeho jednotlivé části vykonávat a jestli nemůže dojít k uváznutí (deadlock) či špatné manipulaci se sdílenými daty. Nedělejte unáhlené předpoklady, že některé kusy kódu fungují správně, protože jsou krátké a provádějí pouze jednoduché operace. Chyba se často vyskytuje právě v nich. Ačkoliv se tato technika může jevit jako velmi zdlouhavá a otravná, překvapivě ušetří spoustu času a vede k odhalení řady malých chyb.



Obrázek 3.7: Použití drv.exe k manipulaci s ovladačem hello.sys

Další způsob, jak se vyhnout dlouhému ladění, spočívá v přesunutí složitosti řešeného problému z režimu jádra do uživatelské aplikace. Struktura ovladače se tak velmi zjednoduší a ladění aplikace lze provést standardními postupy. Ovladač pak funguje pouze jako prodloužená ruka aplikace – umožňuje jí provádět jinak zakázané operace.

Prevence není nikdy stoprocentně účinná. Nemusí odhalit problémy složitějšího charakteru, které se projeví jen za velmi specifických podmínek. Jedná se například o chyby, jež se objeví pouze při specifickém pořadí plánování vláken na procesoru. Pro jejich odhalení je potřeba použít prostředky umožňující chování ovladače sledovat v reálném čase.

DbgPrint

Na rozdíl od obyčejných programů ovladače nemohou jednoduše zobrazovat dialogová okna se zprávami, aby oznamovaly svůj aktuální stav. Mohou ale využívat rutiny `DbgPrint`, která (podobně jako funkce `OutputDebugString` z rozhraní Windows API) pošle zadaný text debuggeru.

Debuggery jádra umožňují při ladění ovladačů používat podobné postupy jako při ladění aplikací. Dokáží umisťovat breakpointy a prohlížet obsah paměti. Jejich ovládání je méně přívětivé, protože nemohou využívat většiny služeb jádra. Musí totiž běžet na nižší úrovni než většina ovladačů, aby mohly kontrolovat jejich běh. Proto některé mechanismy jádra obcházejí a činí operační systém méně stabilním.

Pro monitorování volání funkce `DbgPrint` nepotřebujete plnohodnotný debugger jádra. Postačující práci odvede i utilita `DebugView`, která pouze zobrazuje výstupy z funkcí, jako je právě `DbgPrint` či `OutputDebugString` a jejíž grafické uživatelské rozhraní jste viděli na obrázku 3.6.

DbgPrintEx

Tato rutina plní stejnou úlohu jako DbgPrint, navíc dovoluje volajícímu určit, jaký typ danou zprávu posílá a jak je tato zpráva závažná; zda jde o pouhou informaci, nebo došlo k závažnému problému. Deklaraci rutiny vidíte na výpisu 3.13.

Výpis 3.13: Deklarace funkce DbgPrintEx

```
NTSTATUS __cdecl DbgPrintEx(
  ULONG ComponentId,
  ULONG Level,
  PCSTR Format,
  ... arguments);
```

Hodnota parametru `ComponentId` určuje typ ovladače, který zprávu zasílá. Možné hodnoty vidíte v tabulce 3.4. Parametr `Level` určuje závažnost zprávy. Povoleny jsou libovolné hodnoty, doporučuje se ale používat pouze následující:

- **DPFLTR_ERROR_LEVEL (0)** – závažná chyba,
- **DPFLTR_WARNING_LEVEL (1)** – varování,
- **DPFLTR_TRACE_LEVEL (2)** – oznámení o vykonávání určité části kódu (například určité funkce),
- **DPFLTR_INFO_LEVEL (3)** – nezávažné oznámení jiného druhu.

Ostatní parametry mají stejný význam jako v případě funkce `DbgPrint` či jiných rutin určených pro formátování řetězců (například `printf`). Funkce indikuje úspěch vrácením hodnoty typu NTSTATUS.

Tabulka 3.4: Hodnoty parametru ComponentId funkce DbgPrintEx a jejich význam

Konstanta	Název komponenty	Popis
DPFLTR_IHVVIDEO_ID	IHVVIDEO	Ovladač videa.
DPFLTR_IHVAUDIO_ID	IHVAUDIO	Ovladač zvuku.
DPFLTR_IHVNETWORK_ID	IHVNETWORK	Síťový ovladač.
DPFLTR_IHVSTREAMING_ID	IHVSTREAMING	Ovladač pracující s proudem dat (například dekódující proud zvukových dat).
DPFLTR_IHVBUS_ID	IHVBUS	Ovladač sběrnice.
DPFLTR_IHVDRIVER_ID	IHVDRIVER	Jiný typ ovladače.

Windows dovolují jednotlivé zprávy na základě hodnot parametrů `ComponentId` a `Level` filtrovat; debuggeru doručují jen zprávy s určitými kombinacemi těchto hodnot. Konkrétní nastavení filtrování lze provést buď přímo v debuggerem modifikací daného nastavení přímo v paměti jádra, nebo změnou hodnot klíče HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Debug Print Filter. Tento klíč obsahuje hodnoty typu DWORD. Název hodnoty odpovídá názvu komponenty (viz druhý sloupeček tabulky 3.4), data jsou interpretována jako bitová maska určující, jak závažné zprávy od daného typu ovladače (závažnost zprávy se posuzuje podle hodnoty parametru `Level`) bude systém zasílat. Změna tohoto klíče se do nastavení filtrování promítne až po restartu počítače, protože si operační systém tento klíč čte pouze během svého startu. Změna přímo v paměti jádra se projeví okamžitě.

Rozhodování, zda bude zpráva zaslána debuggeru, probíhá následovně:

- Z hodnoty parametru `Level` systém vypočítá bitovou masku. Hodnota mezi 0 a 31 včetně odpovídá masce s pouze jedním bitem nastaveným na jedničku. Jedná se o danou mocninu dvojky. Například hodnota 5 znamená masku 0x00000020 = 32 = 2^5. Ostatní hodnoty parametru `Level` nejsou nijak transformovány a systém je interpretuje přímo jako bitovou masku.
- Následně jádro provede logický součin (operace AND) masky z předchozího kroku s maskou nastavení filtrování zpráv příslušné komponenty logickým součtem (operace OR) zkombinovanou s maskou WIN2000. Maska příslušné komponenty je při startu systému načtena z registru z výše jmenovaného klíče a může být později přímo upravena debuggerem jádra. Pokud je výsledkem nenulové číslo, zpráva je debuggeru poslána. V opačném případě zaniká. Pokud bitová maska pro danou komponentu neexistuje, jádra její hodnotu uvažuje jako nulovou a po operaci logického součtu nabude hodnoty masky WIN2000.

Poznámka: Maska WIN2000 má ve výchozím natavení hodnotu 0x1. Protože je vždy zkombinována s maskou natavení filtrování určité komponenty, zajišťuje, že nejnižší bit odpovídající úrovni DPLFLTR_ERROR_LEVEL bude vždy nastaven na jedničku, a tak hlášení o závažných chybách budou odeslány debuggeru i v případě, že nastavení filtrování pro příslušnou komponentu by takové jejich odesílání nepovolovalo.

Od Windows Vista se volání `DbgPrint` chová stejně jako volání `DbgPrintEx` s parametrem `ComponentId` DFLTR_DEFAULT_ID a hodnotou `Level` DPLFLTR_INFO_LEVEL. Na starších verzích operačního systému `DbgPrint` vždy posílá zprávy debuggeru.

Tip: Program `DebugView`, který naleznete na stránce www.sysinternals.com, zobrazuje zprávy nehledě na nastavení filtrování v operačním systému. Při zaškrtnuté volbě **Pass Through** v menu **Options** všechny obdržené zprávy přeposílá dále debuggeru jádra a k filtrování vůbec nedochází.

ASSERT

Makro ASSERT slouží testování invariantů a různých podmínek během vykonávání kódu ovladače. Jeho syntaxe je následující:

```
ASSERT(Vyraz)
```

Vyhodnotí-li se obsah výrazu `Vyraz` na TRUE, makro nedělá nic. V opačném případě pošle zprávu debuggeru jádra. V této zprávě najdete název souboru zdrojového kódu a číslo řádky, kde se dané makro nachází, a také text výrazu, který se vyhodnotil na FALSE. Makro slouží k ověření, že při vykonávání daného kódu platí ještě další podmínky, které ale nejsou testovány pomocí konstrukcí if, protože platnost těchto podmínek je dána implicitně – jinak řečeno, ve finální verzi ovladače takové podmínky platí vždy.

Ve finální verzi ovladače se tedy všechna použití makra ASSERT vyhodnotí na TRUE, takže nedochází k odesílání zpráv debuggeru. Makro má výše popsanou sémantiku pouze v případě překladu ovladače v prostředí `Checked build`, kdy má symbol DBG hodnotu 1. V prostředí `Free build`, kde symbol DBG má hodnotu 0, nebo není vůbec definován, se makra ASSERT nezahr-

nují do vygenerovaného binárního souboru `.sys`. Překladač se chová tak, jako by ve zdrojovém kódu vůbec nebyla napsána.

KdPrint a KdPrintEx

Tato makra se v případě překladu zdrojového kódu v prostředí `Checked build` chovají jako rutiny `DbgPrint` a `DbgPrintEx`. Pokud ovladač překládáte v prostředí `Free build`, překladač tato makra ignoruje. Chová se tedy podobně jako v případě makra `ASSERT`. Na rozdíl od volání ostatních funkcí a maker, seznam parametrů pro `KdPrint` a `KdPrintEx` musíte uvádět ve dvojitých kulatých závorkách, například takto:

```
KdPrint(("Testovaci ladici vypis\n"));
```

Důvodem této nutnosti je implementace těchto maker v hlavičkových souborech balíku Windows Driver Kit.

WinDbg

Pokud výpisy z `DbgPrint` k identifikaci problému nestačí, můžete zkusit plnohodnotný debugger jádra. Jedním z nich je program WinDbg, který naleznete v balíku Debugging Tools For Windows, jenž se standardně instaluje společně s WDK. Tento debugger umožňuje za běhu prohlížet a upravovat obsah paměti jádra. Krokování a breakpointy v režimu jádra však nepodporuje lokálně. Abyste mohli ovladač krokovat, musíte jej pustit na jiném (třeba i virtuálním) počítači, na který se potom například pomocí pojmenované roury či sériového portu připojíte.

Protože plně nepodporuje lokální ladění jádra, nemusí tato aplikace ani obcházet mechanismy, čímž nesnižuje stabilitu celého systému. Grafické uživatelské rozhraní debuggeru vidíte na obrázku 3.8.

Pokud hledaná chyba ovladače způsobuje modrou obrazovku smrti, systém většinou na disk ukládá obsah paměti jádra v okamžiku zjištění problému – tzv. crash dump. WinDbg dokáže tyto výpisy paměti při selhání analyzovat, a tak můžete vědět přesně, kde chyba nastala, aniž byste potřebovali druhý stroj, i když třeba jen virtuální.

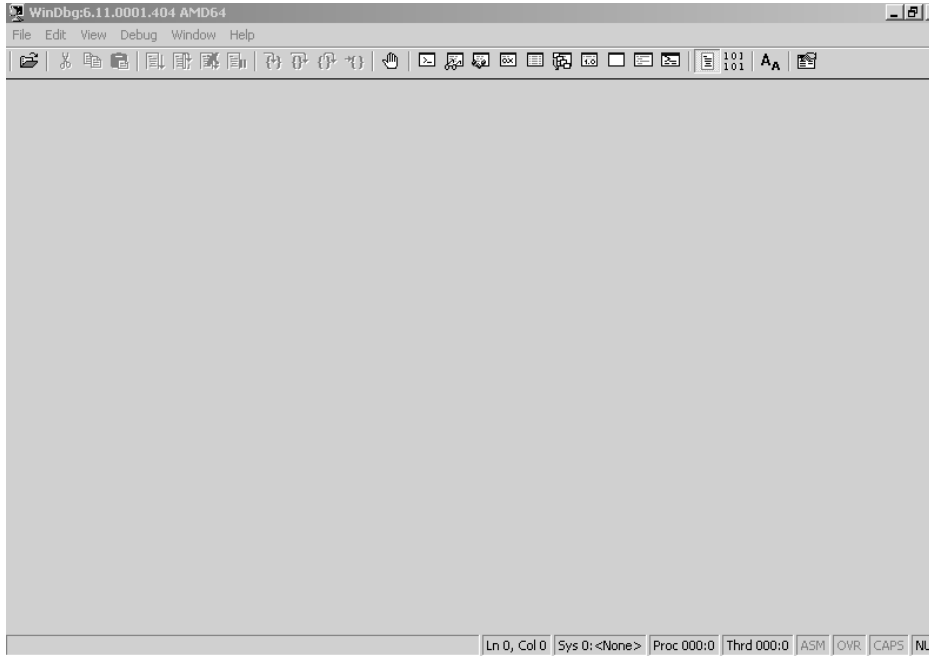
Obsluha WinDbg není nejpříjemnější – program se ovládá podobně jako příkazová řádka. Pro diagnostiku většiny problémů s ovladači ale vystačíte jen s několika málo příkazy, které popisují následující odstavce.

WinDbg umí pracovat s údaji v souborech `.pdb`, které obsahují například názvy a umístění proměnných, datových typů a podprogramů. Tyto informace jsou dostupné i pro hlavní moduly jádra (`ntoskrnl.exe`, `hal.dll`, `win32k.sys`) a nachází se na stránkách Microsoftu.

Pro korektní nastavení ladicích symbolů stačí v menu **File** zvolit položku **Symbol Path...** a do textového pole vyplnit adresu serveru, která zní

```
srv*DownstreamStore*http://msdl.microsoft.com/download/symbols
```

WinDbg z tohoto serveru stahuje ladicí informace k těm souborům, jež zrovna potřebuje. Po správné konfiguraci symbolů můžete začít s vlastní analýzou souboru crash dump.



Obrázek 3.8: Uživatelské rozhraní programu WinDbg

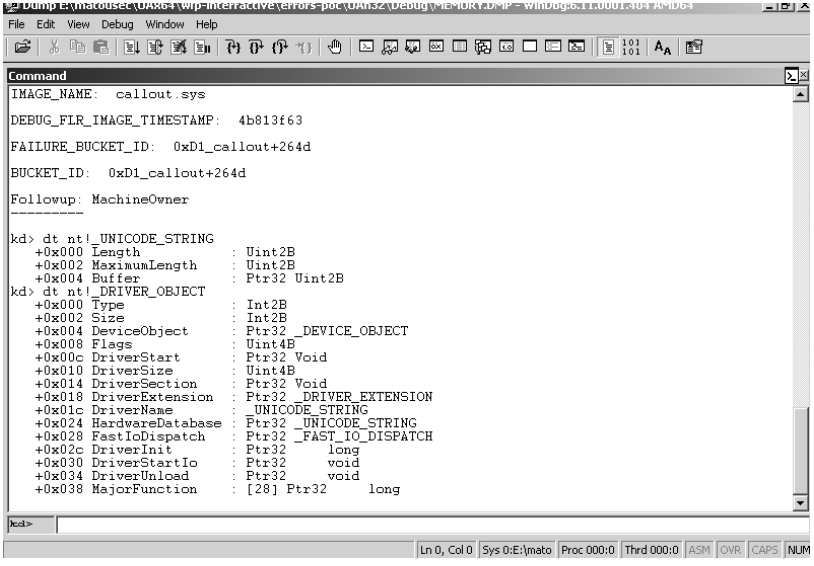
Poznámka: Pokud váš počítač není během práce s debuggerem připojen k Internetu, můžete si ladicí informace k většině ovladačů od Microsoftu stáhnout na disk a do textového pole ve WinDbg vyplnit jejich umístění.

Po otevření souboru crash dump (**File – Open Crash dump...**) WinDbg automaticky vykoná příkaz `!analyze`, který provede zběžný průzkum výpisu paměti. Dozvíte se kód chyby, která modrou obrazovku způsobila, a název ovladače, v němž vznikla.

Podrobnější informace vypíše příkaz `!analyze` s parametrem `-v`. Debugger zobrazí krátký text, který popisuje nejčastější příčiny vzniku dané chyby a význam parametrů. Program také vypíše hodnoty registrů procesoru a stav zásobníku volání v době pádu. Tak se dozvíte, v jaké rutině k chybě došlo. WinDbg se snaží všechny adresy v paměti překládat pomocí ladicích symbolů na jména proměnných a funkcí. Pokud najde i zdrojový kód ovladače, zobrazí číslo řádky, na kterém došlo k volání určité rutiny. Pro některé chyby zobrazuje přímo výpis zdrojového kódu.

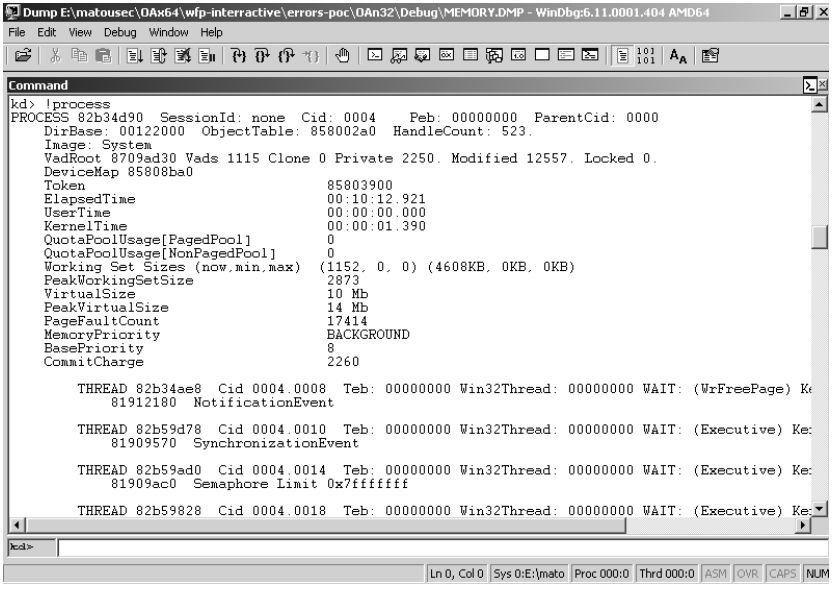
Pokud informace poskytnuté příkazem `!analyze -v` nejsou dostatečné, můžete použít některé z následujících příkazů:

- `dt <typ> [adresa]` – zobrazí strukturu zadaného datového typu. Pokud uvedete i nepovinný parametr `adresa`, pokusí se obsah paměti na této adrese interpretovat jako obsah položek zadaného datového typu. Tento příkaz se hodí nejenom na zjišťování obsahu paměti, ale i na zkoumání datových struktur samotného operačního systému. Podmínkou je mít správně nastavenou cestu k souborům se symboly. Na obrázku 3.9 vidíte několik ukázek použití.



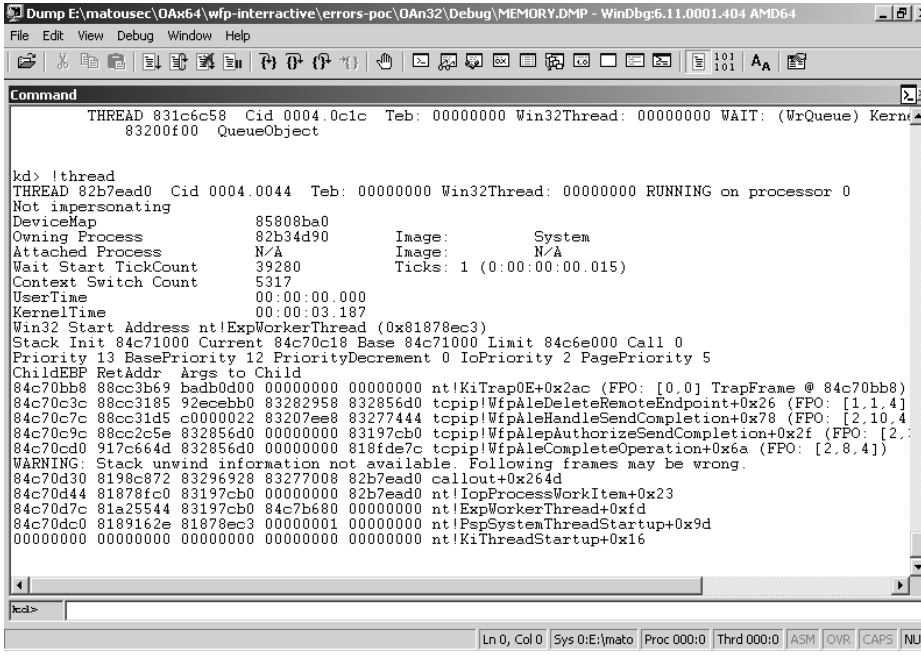
Obrázek 3.9: Příklad použití příkazu dt

- `!process` – zobrazí informace o běžících procesech v době selhání (viz obrázek 3.10).

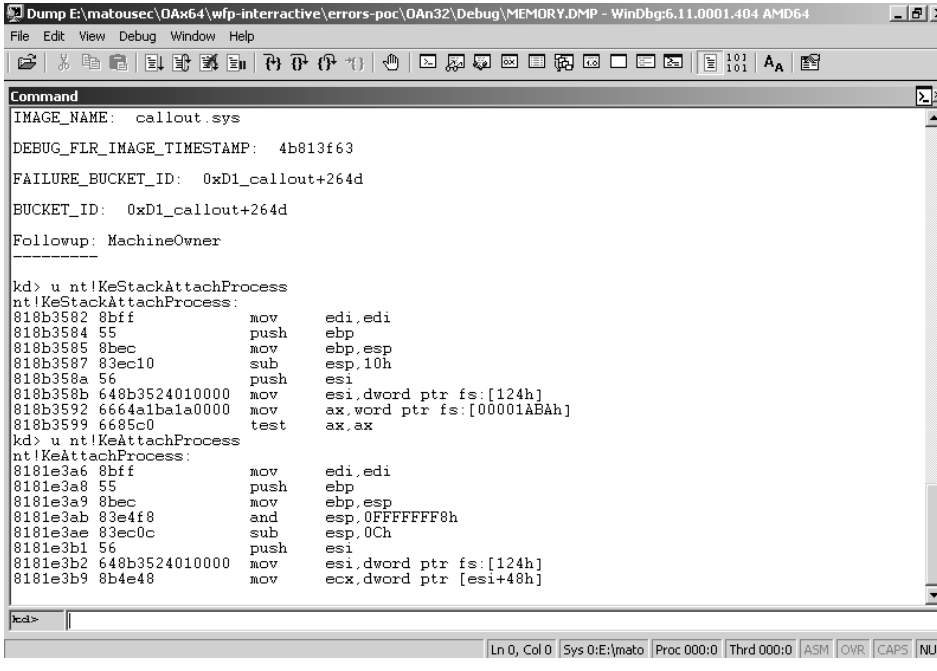


Obrázek 3.10: Příkaz !process

- `!thread` – zobrazí informace o vláknu, které selhání způsobilo. Dozvíte se i údaje o procesu, jemuž patří. Ukázkový výpis vidíte na obrázku 3.11.

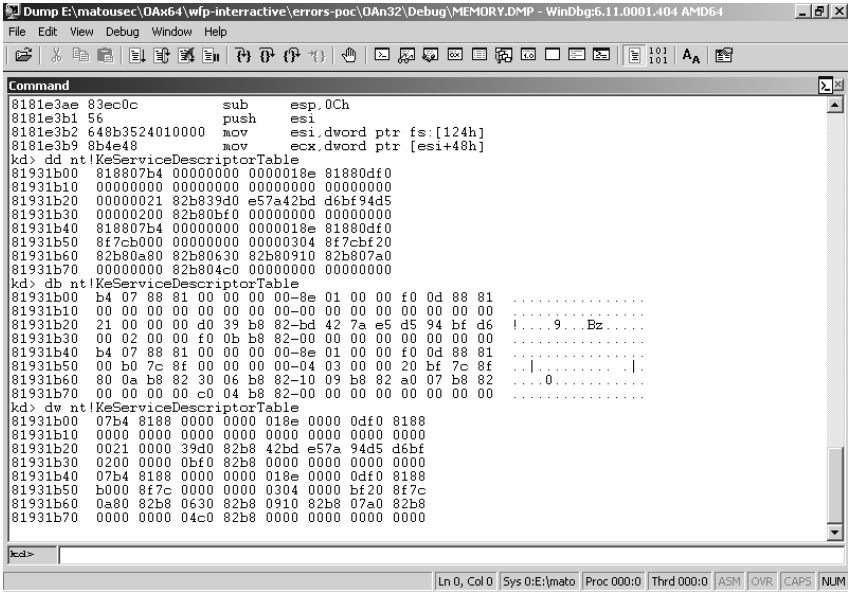


Obrázek 3.11: Příkaz !thread



Obrázek 3.12: Příkaz u

- `u <adresa>` – interpretuje obsah paměti na zadané adrese jako instrukce procesoru. Jako adresu můžete zadat i jméno funkce. Pokud je cesta k souborům PDB dobře nastavena, program si jméno na adresu sám převede. Příklad najdete na obrázku 3.12.
- `db`, `dw`, `dd` – vypíše obsah paměti na zadané adrese jako posloupnost bajtů, slov či dvojslov (viz obrázek 3.13).



Obrázek 3.13: Příkazy db, dw a dd

Tip: Seznam všech příkazů a jejich podrobný popis naleznete v nápovědě k programu. Zde jsou uvedeny pouze ty nejčastěji používané.

Modrá obrazovka smrti

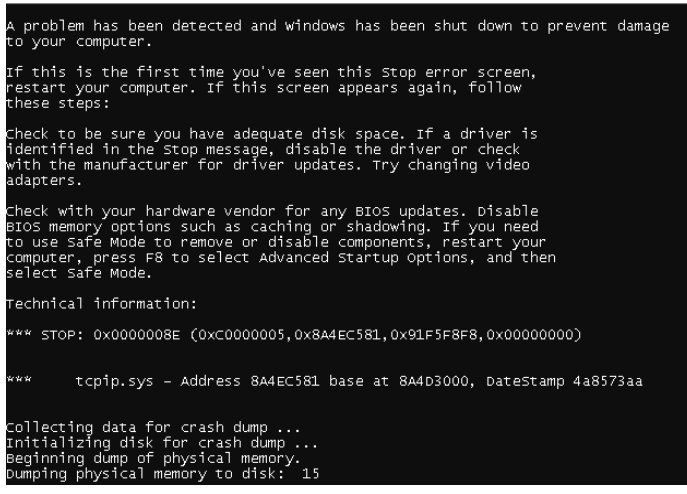
Modrou obrazovku smrti (Blue Screen Of Death – BSOD) již někteří uživatelé pamatují z dob Windows 9x/Me. Systém tuto obrazovku zobrazí, pokud dojde k události neslučitelné s pokračováním v jeho činnosti. Tento oddíl se vám pokusí přiblížit okolnosti jejího vzniku a procesy, které probíhají během vypisování bílého textu na modré pozadí a po něm. Konkrétní ukázku takové modré obrazovky vidíte na obrázku 3.14.

Text modré obrazovky obsahuje informace o tom, co mohlo selhání způsobit a jak by se mu dalo do budoucna předejít. Protože příčin pádu může být velmi mnoho, tyto rady mají velmi obecný charakter, a tudíž nejsou příliš užitečné.

Naopak velmi užitečné informace se nachází v dolní části obrazovky, kde mimo chybového kódu za textem STOP:někdy naleznete i název ovladače, jenž problém pravděpodobně způsobil.

Do horní části obrazovky systém někdy velkými písmeny oddělenými znakem podtržítka vypíše slovní označení pro daný kód chyby.

O probíhajícím výpisu obsahu paměti do stránkovacího souboru systém informuje na posledních řádcích. Pokud se tyto informace neobjeví, ukládání do stránkovacího souboru neproběhlo. Výpis stavu paměti při selhání lze následně použít k odhalení příčiny.



Obrázek 3.14: Modrá obrazovka smrti

Okolnosti vzniku

Když systém zjistí, že není něco v pořádku, vyvolá modrou obrazovku a ukončí svoji činnost. Důvodem může být jak selhání pevného disku či poškození konzistence samotného jádra, ale i drobná chyba v cizím ovladači, ze které by se systém teoreticky mohl zotavit. Včasným zastavením činnosti ale chce předejít možnému poškození hardware.

Pro detekci takových nebezpečných situací má systém zabudováno několik mechanismů. Například správce paměti dokáže označováním bloků paměti, se kterými pracuje, zjistit, že se nějaký ovladač pokusí uvolnit oblast, která mu nepatří. Dále jádro kontroluje, jak rychle probíhají určité kritické operace. Pokud zjistí, že některá z nich trvá moc dlouho, usoudí, že došlo k uváznutí v nekonečné smyčce, a ukončí svoji činnost.

Průběh

Systém vyvolá modrou obrazovku pomocí interních funkcí, mezi které patří `KeBugCheck` a `KeBugCheckEx`. Těchto rutin mohou využít i programátoři ovladačů, pokud usoudí, že za určitých okolností systém prostě nemůže pokračovat v činnosti.

Poznámka: Někdy je vhodné modrou obrazovku smrti vyvolat, i když to okolnosti nevyžadují. Některé viry totiž při vypínání počítače normálním způsobem kontrolují, zda jsou stále zapsány v registrech a na pevném disku tak, aby se znovu spustily při příštím startu. Pokud však běh počítače ukončíte modrou obrazovkou, malware si nemůže ověřit prakticky nic.

Hned po zavolání některé z výše zmíněných rutin systém provede několik kroků k tomu, aby bylo možné zaznamenat informace o selhání na disk. Nejprve pošle všem procesorům krom aktuálního požadavek na zablokování. Jestli se je zablokovat skutečně podařilo, už ale nekontroluje. Jádro se totiž nachází v nedefinovaném stavu, kdy již nelze použít žádných synchronizačních primitiv, protože by mohlo dojít k dalšímu selhání, jež by zamezilo výpisu dat o příčině toho původního na bezpečné místo na pevném disku. Systém si zároveň zajistí naprostou exkluzivitu aktuálního procesoru – aktuálně vykonávaný kód nemůže být ničím přerušen.

Jakmile má aktuální vlákno zaručen exkluzivní přístup k procesoru a k veškerému hardware, přepne monitor do textového režimu a vypíše známý text o tom, že došlo k selhání. Systém následně zapíše informace o chybě na disk, a pokud uživatel má zakázán automatický restart při selhání, procesor přejde do vykonávání nekonečné smyčky.

Během vypisování obsahu modré obrazovky se dostanou ke slovu i jiné ovladače než hlavní modul jádra, který obsahuje rutiny `KeBugCheck` a `KeBugCheckEx`. Každý ovladač si může zaregistrovat rutinu, jíž jádro předá řízení těsně po výpisu většiny textu na obrazovku. Rutina má ale vzhledem k nekonzistentnímu stavu systému velmi omezený repertoár možností. Základní omezení jsou tato:

- Nelze alokovat paměť – ani stránkovanou ani nestránkovanou.
- Nelze přistupovat ani k již alokované stránkované paměti. Nekonzistentní stav systému nedovoluje obsluhovat výpadky stránek.
- Není možné použít žádných synchronizačních mechanismů.
- Nelze volat drtivou většinu funkcí exportovaných jádrem.

Co tedy taková rutina dělat může? Jediná užitečná možnost spočívá v předání důležitých informací perifernímu zařízení, například pevnému disku, což Windows opravdu udělají, jsou-li tak nakonfigurovány.

Jakmile systém zjistí všechny dostupné informace o selhání, nashromážděná data zapíše na disk, aby se na ně uživatel mohl později podívat. Údaje zapisuje do stránkovacího souboru, který se následně označí speciálním příznakem, aby při příštím startu systém věděl, že obsah souboru není bezcenný.

Během bootovací sekvence systém zkontroluje, zda se ve stránkovacím souboru nenachází informace o předchozím selhání. Pokud ano, uloží je do předem nastaveného adresáře.

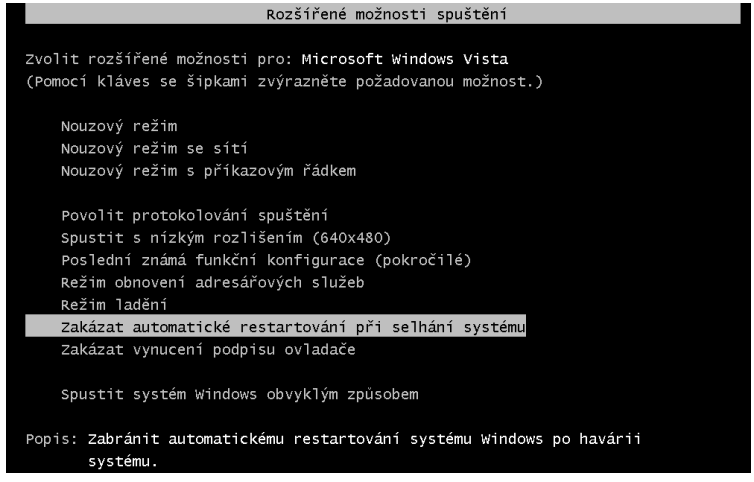
Z výše popsaného vyplývají následující důsledky:

- Pokud stránkovací soubor nemáte zapnutý, informace o selhání systému se na disk neuloží. Proto je rozumné u stránkovacího souboru nastavit alespoň minimální velikost, která se například u Windows XP rovná dvěma megabajtům.
- Nemělo by se stávat, aby se vypisované informace do stránkovacího souboru nevešly. Tudíž mějte tento soubor dost velký, aby k tomu nedocházelo. Pokud chcete vypisovat jen nejnutnější informace o selhání, bohatě stačí nastavit velikost na 2 MB. Pokud při selhání chcete vypisovat obsah celé paměti jádra, resp. veškeré fyzické paměti, stránkovací soubor by měl být tak velký, aby tyto informace pojal – výpis celé paměti jádra zabere často i pár set megabajtů.
- Pro vyjmutí informací o selhání ze stránkovacího souboru je nutné systém úspěšně nastartovat do použitelného stavu, přinejhorším do nouzového režimu. Informace o selhá-

ní se ukládají do odděleného souboru až po zavedení celého jádra a spuštění procesu `smss.exe`.

- Pokud k selhání dojde v rané fázi zavádění jádra, kdy stránkovací soubor ještě neexistuje, nebo jej systém ještě neotevřel, informace o selhání se nikam neuloží.

Tip: Pokud k selhání dojde v rané fázi spouštění systému, můžete se po restartu těsně předtím, než se objeví logo Windows, stisknutím klávesy F8 dostat do hlavní nabídky zavaděče (viz obrázek 3.15). Z tohoto menu můžete krom jiného instruovat systém, aby se pokusil nastartovat do nouzového režimu, nebo restartování při selhání zakázat. Pokud restartování při selhání zakážete, systém zůstane „viset" na modré obrazovce a vy máte čas poznamenat si údaje o chybě.



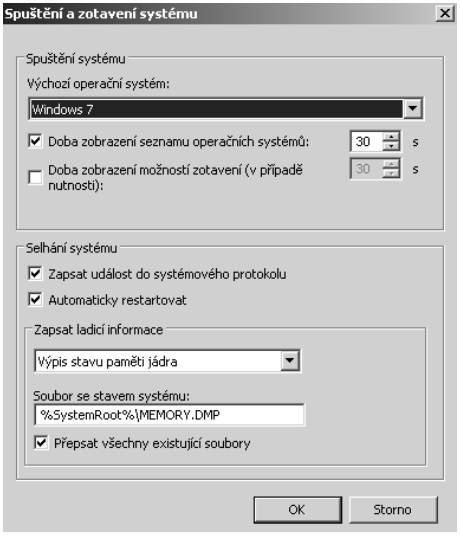
Obrázek 3.15: Hlavní menu zavaděče

V případě, že se během startu systému objeví modrá obrazovka, vyplatí se vyzkoušet, zda nenaběhne alespoň nouzový režim či nouzový režim s prací v síti. V těchto režimech Windows používají jen ty nejnutnější ovladače a rezidentní ochrany bezpečnostních aplikací obvykle nejsou aktivní.

Nastavení výpisu příčin selhání

Všechna nastavení týkající se selhání se nacházejí v jediném okně, které vidíte na obrázku 3.16. K tomuto dialogu se ve Windows XP dostanete spuštěním aplikace **Systém** v **Ovládacích panelech**. Následně vyberte záložku **Upřesnit** a v oblasti **Spuštění a zotavení systému** klepněte na tlačítko **Nastavení**. Na vyšších verzích Windows je třeba v **Ovládacích panelech** klepnout na skupinu **Systém a zabezpečení**, dále na **Systém** a poté otevřít okno **Upřesnit nastavení systému**. Tím spustíte aplikace, která se ve Windows XP skrývá pod ikonou **Systém** a další postup je shodný.

Pro chování při výskytu modré obrazovky smrti jsou důležité oblasti **Selhání systému** a **Zapsat ladicí informace**. V první z nich můžete nastavit, zda si přejete, aby se zpráva o selhání objevila i v systémovém protokolu (**Zapsat událost do systémového protokolu**) a zda se má systém po dokončení všech úkonů spojených se svým pádem automaticky restartovat (položka **Automaticky restartovat**).



Obrázek 3.16: Okno nastavení chování při selhání

Tip: Při povoleném automatickém restartování může výpis modré obrazovky proběhnout tak rychle, že si jej ani nevšimnete, a celá situace se bude jevit jako náhlý restart. Proto se doporučuje automatické restartování zakázat. Tato volba byla zavedena hlavně kvůli serverům, které musí být schopné plnit požadavky klientů pokud možno neustále.

Oblast **Zapsat ladicí informace** dovoluje nastavit, jak velký objem dat bude systém o selhání ukládat do stránkovacího souboru. Z těchto údajů se po příštím úspěšném startu vytvoří soubor s koncovkou `.dmp`, který systém uloží do umístění specifikovaného v poli **Soubor se stavem systému**. Množství vypisovaných informací lze ovlivnit zvolením jedné z následujících možností.

- **(Žádný)** – systém se nepokusí do stránkovacího souboru ukládat žádné informace o stavu.
- **Omezený výpis stavu paměti jádra (256 KB)** – do stránkovacího souboru se uloží pouze kód chyby, která selhání způsobila, a některé části paměti týkající se vlákna a procesu, v jejichž kontextu k ní došlo. Z těchto údajů lze krom aktuálního procesu a vlákna vyčíst, jaká rutina selhání způsobila (ukládá se i zásobník volání aktuálního vlákna) a které ovladače se v době incidentu v jádře nacházely. Soubory DMP, které obsahují tuto špetku informací, mají velikost od 64 do 256 KB, a proto vám při tomto nastavení stačí i nejmenší velikost stránkovacího souboru. Běžně se nazývají `minidumpy` a ve výchozím nastavení se ukládají do složky `Minidump` v kořenovém adresáři instalace (například `C:\Windows\Minidump`).
- **Výpis paměti jádra** – systém vypíše do stránkovacího souboru veškerý obsah paměti jádra. Velikost výsledného souboru DMP se pohybuje v řádu stovek megabajtů; vyžaduje tedy větší stránkovací soubor. Velikost však plně vyvažuje množství informací, které v mnoha případech již umožňuje jednoznačně určit příčinu selhání. Soubor totiž obsahuje i paměť, v níž se nacházely kód a data jednotlivých ovladačů, a tudíž je možné provést kontrolu, zda některý z nich nebyl modifikován, což může ukazovat na přítomnost malware.

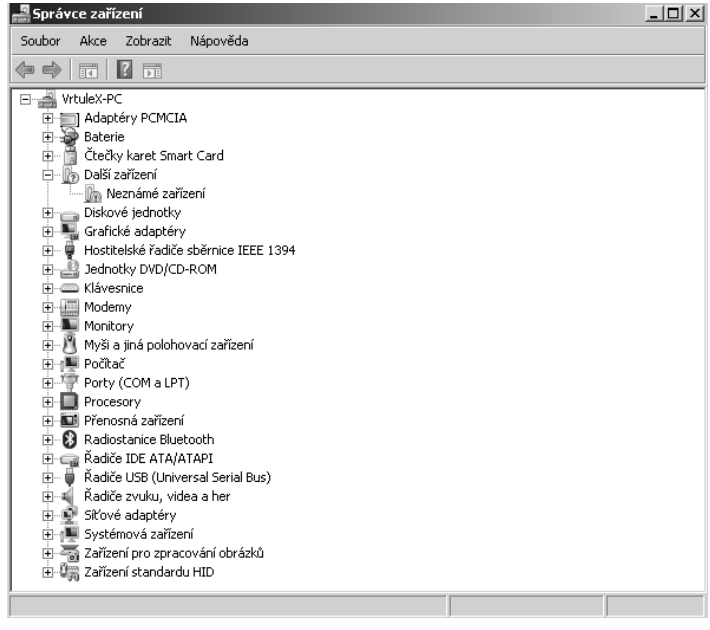
- **Úplný výpis stavu fyzické paměti** – do stránkovacího souboru se vypíše obsah celé paměti RAM (stránkovací soubor musí být alespoň tak velký, jako fyzická paměť). Ze zaznamenaných údajů se můžete dozvědět i informace o stavu privátní části adresového prostoru některých procesů, což ale ve většině případů při odhalování příčiny selhání nepomůže. Tato možnost se nevyskytuje na Windows 7.

Výpisy stavu paměti jádra a výpisy celé fyzické paměti se ve výchozím nastavení ukládají do souboru MEMORY.DMP v kořenovém adresáři instalace Windows (např. `C:\WINDOWS\MEMORY.DMP`).

Zjišťování příčin modrých obrazovek

Modrou obrazovku smrti může způsobit virus, špatně fungující hardware nebo nesprávně pracující ovladač. To jsou příčiny velmi obecné a zjištění konkrétnějších údajů může být velmi obtížný úkol. Ne však nemožný. Pokud se budete držet následujících kroků, určitou šanci na úspěch máte.

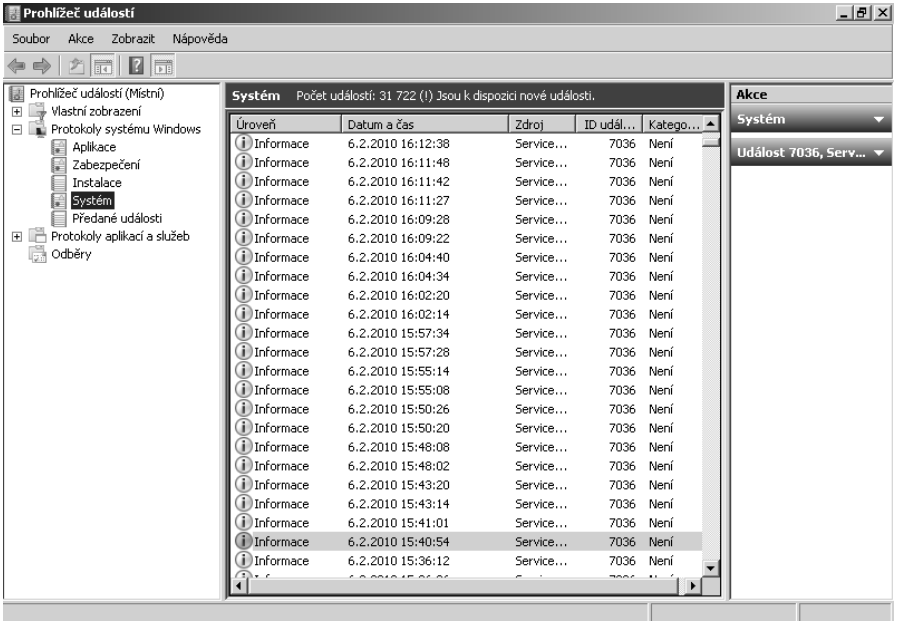
- Podívejte se do Správce zařízení. Zde se můžete dozvědět, zda nějaké ovladače nechybí, nebo nejsou špatně nainstalovány. Tento snap-in ve Windows XP zobrazíte vybráním záložky **Hardware** v aplikaci **Sytém** v **Ovládacích panelech**. V novějších verzích operačního systému musíte v **Ovládacích panelech** vybrat položku **Systém a zabezpečení**, dále klepnout na **Systém** a na **Upřesnit nastavení systému**. Správce zařízení opět najdete na kartě **Hardware**. Uživatelské rozhraní programu vidíte na obrázku 3.17.



Obrázek 3.17: Správce zařízení

- Podívejte se do Prohlížeče událostí (jehož okno je zobrazeno na obrázku 3.18), zejména do protokolu **Systém**. Do této sekce mohou aplikace, služby a ovladače zapisovat různá hlášení. Pokud se tam například často objevují hlášení od ovladače disku o tom, že se

nepodařilo načíst stránku ze stránkovacího souboru, pevný disk pravděpodobně utrpěl nevratná poškození. Prohlížeč událostí ve Windows XP najdete v **Ovládacích panelech** pod ikonou **Nástroje pro správu**. Ve Windows Vista a Windows 7 se tato aplikace skrývá také pod položkou **Systém a zabezpečení** – ve skupině **Nástroje pro správu** klepněte na odkaz **Zobrazit protokoly událostí**.



Obrázek 3.18: Prohlížeč událostí

- Relativně často jsou modré obrazovky způsobeny špatně fungující pamětí RAM. K jejich testování slouží například bezplatný nástroj `Memtest`.
- Lze použít i nástroj WinDbg, který ukáže, jaký ovladač pravděpodobně selhání způsobil a pokusí se automaticky nalézt příčinu pádu. Postupovat můžete stejným způsobem jako při ladění pádů způsobených svými ovladači.

Zjišťovat příčinu modré smrti lze i „on the fly" – tedy přímo při jejím vzniku. Pokud systém nastartujete v režimu ladění (musíte před bootováním Windows stisknout klávesu F8, nebo použít příkaz `bcdedit /set debug on`), nezobrazí se při selhání modrá obrazovka. Systém přejde do nekonečné smyčky a čeká, dokud se k němu nepřipojí další počítač s nainstalovaným debuggerem jádra.

Závěrečný příklad

Web: http://www.jadro-windows.cz/projekty/logptm

Poslední část neformálního povídání o ovladačích se pokusí vaše dosavadní znalosti zcelit a ještě více prohloubit. Následující řádky popisují, na jakých principech funguje ovladač `logptm.sys`,

jehož kompletní zdrojový kód najdete na internetových stránkách knihy. Tento ovladač zaznamenává následující systémové události a posílá je aplikaci, jež je zobrazuje:

- Vytvoření nového procesu.
- Ukončení procesu.
- Vytvoření nového vlákna.
- Ukončení běhu vlákna.
- Načtení nového souboru formátu PE (souboru EXE, knihovny DLL či ovladače) do paměťového prostoru procesu.

Tato část popisuje pouze techniky, principy a sémantiku rozhraní a funkcí, kterých ovladač využívá. Nenajdete v ní popis zdrojového kódu řádek po řádku; k tomuto účelu slouží komentáře v samotném kódu.

Způsob uchovávání událostí

`Logptm.sys` rozeznává události tří kategorií: události procesu, události vlákna a události modulu (souboru PE). Každá událost je reprezentována záznamem, který se skládá ze dvou částí. První obsahuje obecné informace jako typ události, celkovou velikost záznamu a čas, kdy nastala. Druhá část uchovává informace specifické pro danou kategorii událostí (například PID nově vytvořeného procesu či jméno souboru nově načteného ovladače). `Logptm.sys` si všechny události uchovává v obousměrném cyklickém spojovém seznamu s hlavou.

Jádro Windows pro práci se spojovými seznamy exportuje řadu funkcí. Nejdůležitější z nich najdete v tabulce 3.5.

Tabulka 3.5: Důležité rutiny pro práci s obousměrnými spojovými seznamy

Název rutiny	Parametry	Popis
`InitializeListHead`	PLIST_ENTRY ListHead	Inicializuje strukturu LIST_ENTRY na hlavu prázdného seznamu.
`InsertHeadList`	PLIST_ENTRY ListHead PLIST_ENTRY Entry	Vloží nový prvek (`Entry`) na první pozici za hlavu (`ListHead`).
`InsertTailList`	PLIST_ENTRY ListHead PLIST_ENTRY Entry	Vloží nový prvek (`Entry`) na konec spojového seznamu (`ListHead`).
`RemoveHeadList`	PLIST_ENTRY ListHead	Odstraní ze seznamu s hlavou `ListHead` první prvek a vrátí jeho adresu.
`RemoveTailList`	PLIST_ENTRY ListHead	Odstraní ze seznamu s hlavou `ListHead` poslední prvek a vrátí jeho adresu.
`RemoveEntryList`	PLIST_ENTRY Entry	Odstraní zadaný prvek (`Entry`) ze seznamu.
`IsListEmpty`	PLIST_ENTRY Entry	Vrátí TRUE, pokud je seznam `ListHead` prázdný (obsahuje pouze hlavu). V opačném případě je vrácena hodnota FALSE.

Obousměrný spojový seznam je reprezentován strukturou LIST_ENTRY, která obsahuje odkaz na následníka (`Flink`) a předchůdce (`Blink`). Pokud chcete spojit nějaké záznamy do takového spojového seznamu, stačí do jejich struktury vložit položku typu LIST_ENTRY a pomocí odkazů na předka a následníka a za využití funkcí z tabulky 3.3 je propojit.

Upozornění: Programujete-li ovladač pro Windows XP, dejte si pozor na rutinu `RemoveEntryList`. Tato funkce podle názvu odstraní zadaný prvek z obousměrného spojového seznamu. Její sémantika pod Windows XP a staršími systémy provede pouze polovinu práce; upraví odkazy v prvku, který zamýšlíte odebrat, ale odkazů v předku a následníku se nedotkne. Předek a následník tedy zůstávají v iluzi, že odebraný prvek do seznamu stále patří. Na Windows Vista a novějších rutina `RemoveEntryList` naopak upravuje odkazy v předku a následníku, nikoliv odkazy v odebíraném prvku.

Tip: Jeden záznam (blok) může být zapojen do více spojových seznamů najednou. Každá položka LIST_ENTRY obsažená v definici záznamu může zajistit zapojení do jednoho obousměrného spojového seznamu. Tohoto triku Windows relativně často využívají; například struktury EPROCESS, které reprezentují procesy, jsou navzájem propojeny hned několika spojovými seznamy. Též struktury ETHREAD reprezentující vlákna jsou členy více seznamů najednou.

Paměť pro uchovávání jednotlivých událostí ovladač `logptm.sys` alokuje z nestránkovaného fondu. Pro alokaci a uvolňování bloků používá funkce `ExAllocatePoolWithTag` a `ExFreePoolWithTag`.

`ExAllocatePoolWithTag` bere tři argumenty. První z nich určuje, z jakého fondu chcete paměť alokovat. Nejčastěji se používá paměť ze stránkovaného (`PagedPool`) či nestránkovaného (`NonPagedPool`) fondu. V druhém parametru specifikujete, kolik bajtů potřebujete. Třetí parametr udává, jakou značkou má správce paměti tento blok označit. Stejnou značku budete muset specifikovat i při uvolnění bloku. Funkce v případě úspěchu vrátí adresu nově alokovaného bloku a NULL, pokud se alokace nepovede.

Rutina `ExFreePoolWithTag` slouží k uvolnění paměti alokované funkcí popsanou v předešlém odstavci. Pro úspěšné uvolnění bloku musíte specifikovat jeho adresu a značku, kterou jste uvedli při alokaci.

Ovladač `logptm.sys` používá vylepšené spojové seznamy, které jsou implementovány v souborech `lists.c` a `lists.h`.

Implementace je obecnější, než by bylo pro účely tohoto modulu zapotřebí. Umožňuje seznamy vytvářet jak ze stránkovaného, tak z nestránkovaného paměťového fondu a obsahuje podporu pro současný přístup k jednomu seznamu z více vláken najednou. Protože se k synchronizaci přístupu z více vláken používá spinlock, o němž se dočtete v následující kapitole, seznam musí být tvořen z nestránkovaného fondu, pokud chcete zabudované synchronizace využít. Spinlock je také jediným důvodem, proč ovladač `logptm.sys` položky seznamů z nestránkovaného fondu alokuje. Rutiny pro synchronizovaný přístup obsahují v názvu příponu `–Lock` (například `lListTailInsertLock` či `lListHeadRemoveLock`).

Aparát pro vytváření, rušení a provádění dalších operací se záznamy událostí najdete v souborech `events.c` a `events.h`. Tento aparát využívá vylepšené implementace spojových seznamů ze souborů `lists.c` a `lists.h`.

Použité rozhraní pro zachytávání událostí

Jádro systému obsahuje různá rozhraní, která ovladačům umožňují monitorovat a měnit chování celého systému. Každá nová verze Windows většinou přináší nová rozhraní, která jsou mnohem sofistikovanější než jejich předchůdci a snadněji se s nimi pracuje. Nové funkce se mohou objevit i v rámci významnější aktualizace typu Service Pack.

Většina těchto rozhraní slouží k monitorování a ovlivňování aktivity vláken, procesů, sítě či souborového systému a s nejvýznamnějšími z nich se setkáte v průběhu dalších kapitol.

Ovladač `logptm.sys` pro sběr událostí využívá velmi staré rozhraní, které dovoluje provádět pouze monitorování. Ovladač je například schopen zaznamenat vznik nového procesu, nedokáže však tomuto procesu zabránit ve spuštění kódu, což je schopnost užitečná zejména pro bezpečnostní aplikace, jako firewally a rezidentní ochrany antivirových programů.

Veškeré prostředky potřebné pro sběr událostí naleznete v souborech `notify.c`, `notify.h`, `notify_events.c` a `notify_events.h` v projektu `logptm` na internetových stránkách knihy.

`Logptm.sys` zaregistruje do systému několik vlastních podprogramů, kterým jádro předá řízení, kdykoliv dojde ke sledované události. Takovémuto druhu rutin, jenž nevolá modul, v němž jsou implementovány, ale jiná součást systému, se říká *zpětně volané funkce* (*callback*).

`Logptm.sys` zaregistruje rutinu pro sledování událostí týkajících se procesů pomocí volání funkce `PsSetCreateProcessNotifyRoutine`. Pro zahájení sledování aktivit vláken a souborů PE použije `PsSetCreateThreadNotifyRoutine` a `PsSetLoadImageNotifyRoutine`. Všechny tři rutiny jako parametr berou adresu zpětně volané funkce, jež chce volající zaregistrovat. `PsSetCreateProcessNotifyRoutine` navíc akceptuje druhý parametr typu `boolean`, který určuje, zda se má zadaná rutina zaregistrovat, nebo odregistrovat. Pro odregistrování podprogramů pro monitorování vláken a souborů PE slouží funkce `PsRemoveCreateThreadNotifyRoutine` a `PsRemoveLoadImageNotifyRoutine`, které jako svůj jediný parametr požadují adresu zpětně volané funkce.

Všechny zpětně volané funkce ovladače `logptm.sys` provádějí stejný úkol – získají o dané události co nejvíce informací, zapíšou je do záznamu a ten přidají na konec spojového seznamu všech událostí. Jejich kód vidíte na výpisu 3.14.

Procedura monitorující události týkající se procesů – `NotifyProcess` – se při vytváření nového procesu dozví jeho PID a také identifikátor rodiče. Protože je tato rutina volána v kontextu vlákna, které nový proces vytvořilo, pomocí funkce `PsGetCurrentProcessId`, jež vrací PID procesu, v jehož kontextu se aktuální vlákno nachází, lze získat i identifikaci tvůrce. Pozor, proces, který je rodičem nového procesu, nemusí být jeho tvůrcem. Z parametru `Create` se ovladač dozví, jestli systém zavolal proceduru `ProcesNotify` v důsledku vytvoření nového (hodnota TRUE) či ukončení existujícího (hodnota FALSE) procesu.

Podprogram `NotifyThread` je zavolán při vytvoření nebo ukončení vlákna. Z hodnot jeho parametrů lze vyčíst identifikátor TID vlákna a PID procesu, kterému vlákno patří. Parametr `Create` opět udává, zda systém proceduru vyvolal v důsledku vytvoření nového (hodnota TRUE) či ukončení běžícího (hodnota FALSE) vlákna. V případě vytvoření nového vlákna běží rutina v kontextu tvůrce nové entity.

Procedura `NotifyImage` je systémem volána, kdykoliv dojde k namapování spustitelného souboru do paměti. První parametr obsahuje jméno mapovaného souboru. Druhý identifikuje proces, do jehož adresového prostoru je soubor mapován. Pokud dochází k mapování do prostoru jádra, tento argument nese hodnotu NULL. Třetí parametr odkazuje na strukturu s dalšími informacemi o mapovaném souboru, například jeho velikostí či virtuální adresou jeho počátku.

Výpis 3.14: Kód zpětně volaných funkcí pro sběr událostí

```
VOID NotifyProcess(HANDLE ParentId, HANDLE ProcessId, BOOLEAN Create)
{
  PEVENT Event = NULL;
  LARGE_INTEGER SysTime;
  LARGE_INTEGER LocTime;

  // Zjistime aktualni cas udalosti a prevedeme do spravne casove zony
  KeQuerySystemTime(&SysTime);
  ExSystemTimeToLocalTime(&SysTime, &LocTime);

  // Vytvorime novou udalost popisujici vznik/zanik procesu
  Event = ProcessEventCreate(&LocTime, ParentId, PsGetCurrentProcessId(),
    PsGetCurrentThreadId(), ProcessId, Create);
  if (Event!= NULL)
    // Podari-li se, ulozime ji do seznamu udalosti
    EventAddToListLock(Event);
  return;
}

VOID NotifyThread(HANDLE ProcessId, HANDLE ThreadId, BOOLEAN Create)
{
  PEVENT Event = NULL;
  LARGE_INTEGER SysTime;
  LARGE_INTEGER LocTime;

  // Zjistime aktualni cas udalosti a prevedeme do spravne casove zony
  KeQuerySystemTime(&SysTime);
  ExSystemTimeToLocalTime(&SysTime, &LocTime);

  // Vytvorime novou udalost popisujici vznik/zanik vlakna
  Event = ThreadEventCreate(&LocTime, ThreadId, ProcessId,
PsGetCurrentThreadId(),
    PsGetCurrentProcessId(), Create);
  if (Event!= NULL)
    // Podari-li se, ulozime ji do seznamu udalosti
    EventAddToListLock(Event);

  return;
}
```

```
VOID NotifyImage(PUNICODE_STRING FullImageName, HANDLE ProcessId,
  PIMAGE_INFO ImageInfo)
{
  PEVENT Event = NULL;
  LARGE_INTEGER SysTime;
  LARGE_INTEGER LocTime;

  // Zjistime aktualni cas udalosti a prevedeme do spravne casove zony
  KeQuerySystemTime(&SysTime);
  ExSystemTimeToLocalTime(&SysTime, &LocTime);

  // Vytvorime novou udalost popisujici nacteni souboru PE do pameti.
  Event = ImageEventCreate(&LocTime, FullImageName, ProcessId,
    PsGetCurrentThreadId(),
    ImageInfo->ImageBase, (ULONG)ImageInfo->ImageSize);
  if (Event!= NULL)
    // Podari-li se, ulozime ji do seznamu udalosti
    EventAddToListLock(Event);

  return;
}
```

Inicializace a úklid

Inicializace probíhá v rutině `DriverEntry`. Ovladač nejprve inicializuje části obstarávající sběr událostí a jejich zaznamenávání (rutiny `EventsInit` a `NotifyInit`, výpis 3.15). Poslední část inicializace (rutina `DriverInit`) spočívá v přípravě na komunikaci s aplikací.

Rutina `NotifyInit` připojí zpětně volané funkce `NotifyProcess`, `NotifyThread` a `NotifyImage` do systému – pomocí `PsSetXxxNotifyRoutine` je zaregistruje pro odběr příslušných událostí. Funkce `EventsInit` inicializuje spojový seznam, do kterého bude ovladač ukládat příchozí události, které ještě nepředal k dalšímu zpracování do uživatelského režimu.

Výpis 3.15: Rutiny NotifyInit, EventsInit a DriverEntry

```
NTSTATUS NotifyInit(VOID)
{
  NTSTATUS status = STATUS_UNSUCCESSFUL;

  // Zaregistrujeme rutinu pro sbirani udalosti tykajicich se procesu
  status = PsSetCreateProcessNotifyRoutine(NotifyProcess, FALSE);
  if (NT_SUCCESS(status)) {
    // Zaregistrujeme rutinu pro sbirani udalosti tykajicich se vlaken
    status = PsSetCreateThreadNotifyRoutine(NotifyThread);
    if (NT_SUCCESS(status)) {
      // Registrace rutiny pro zachytavani udalosti tykajicich se souboru PE.
      status = PsSetLoadImageNotifyRoutine(NotifyImage);
      if (!NT_SUCCESS(status)) {
        // V pripade neuspechu je nutne odregistrovat, co bylo zaregistrovano
```

```
        PsRemoveCreateThreadNotifyRoutine(NotifyThread);
        PsSetCreateProcessNotifyRoutine(NotifyProcess, TRUE);
      }
    } else {
      // V pripade neuspechu je nutne odregistrovat, co bylo zaregistrovano
      PsSetCreateProcessNotifyRoutine(NotifyProcess, TRUE);
    }
  }

  return status;
}

NTSTATUS EventsInit(VOID)
{
  NTSTATUS status = STATUS_UNSUCCESSFUL;

  Events = lListCreate(NonPagedPool, _EventDestroy);
  if (Events!= NULL)
    status = STATUS_SUCCESS;
  else status = STATUS_INSUFFICIENT_RESOURCES;

  return status;
}

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
  NTSTATUS status=STATUS_UNSUCCESSFUL;

  // Inicializujeme seznam, do ktereho budeme zaznamenavat prichazeici udalosti
  // o procesech, vlaknech a modulech
  status = EventsInit();
  if (NT_SUCCESS(status)) {
    // Zaregistrujeme notifikacni rutiny, ktere system zavola
    // pri vytvareni/ukoncovani
    // procesu a vlakna a behem nacitani noveho modulu do pameti
    status = NotifyInit();

    if (NT_SUCCESS(status)) {
      status = DriverInit(DriverObject);
      if (!NT_SUCCESS(status)) {
        // Pokud se neco nezdarilo, musime vse uvest do puvodniho stavu
        NotifyFinit();
        EventsFinit();
      }
    } else EventsFinit();
  }
  return status;
}
```

Příprava na komunikaci probíhá ve dvou fázích. V první ovladač vytvoří objekt zařízení a pomocí symbolického odkazu jej zpřístupní procesům uživatelského režimu. Příslušný kód vidíte na výpisu 3.16.

Výpis 3.16: První fáze přípravy na komunikaci ovladače s aplikací

```
NTSTATUS DriverInit(PDRIVER_OBJECT DriverObject)
{
  UNICODE_STRING udevice;
  UNICODE_STRING usymlink;
  NTSTATUS status = STATUS_UNSUCCESSFUL;
  // Vytvorime zarizeni, pres ktere s budou s ovladacem komunikovat aplikace
  RtlInitUnicodeString(&udevice, LOGPTM_DEVICE);
  status = IoCreateDevice(DriverObject, 0, &udevice, FILE_DEVICE_UNKNOWN, 0,
    FALSE, &DriverObject->DeviceObject);
  if (NT_SUCCESS(status)) {
    // Komunikacni zarizeni bylo vytvoreno v oblasti,
    // do ktere normalni aplikace
    // nemaji za vyuziti dokumentovanych prostredku pristup.
    // Zarizeni zpristupnime vytvorenim symbolickeho odkazu
    RtlInitUnicodeString(&usymlink, LOGPTM_SYMLINK);
    status = IoCreateSymbolicLink(&usymlink, &udevice);
...
```

Okolní svět pracuje se zařízením podobně jako s obyčejným souborem. Kdykoliv nějaká aplikace nebo ovladač na zařízení pošle požadavek (otevření, čtení, zápis...), správce vstupně/výstupních operací se podívá do pole `MajorFunction` struktury DRIVER_OBJECT, která popisuje ovladač `logptm.sys`. Toto pole obsahuje adresy zpětně volaných funkcí, kterým má správce předat řízení při obsluze požadavků určitého typu. Nejčastěji využívané rutiny a jejich indexy v poli `MajorFunction` popisuje tabulka 3.6. V tabulce také vidíte, kterými standardními rutinami rozhraní Windows API mohou aplikace požadavky příslušného typu vytvářet a odesílat.

Tabulka 3.6: Typy požadavků a jim odpovídající části pole MajorFunction ve struktuře DRIVER_OBJECT

Index obslužné rutiny v poli MajorFunction	Typ požadavku	Rutina Windows API, která tento typ požadavku vytváří
IRP_MJ_CREATE	Získání přístupu k zařízení (otevření).	`CreateFile`
IRP_MJ_CLOSE	Ukončení práce se zařízením (zavření).	`CloseHandle`
IRP_MJ_READ	Čtení dat.	`ReadFile`
IRP_MJ_WRITE	Zápis dat.	`WriteFile`
IRP_MJ_DEVICE_CONTROL	Zaslání obecné zprávy.	`DeviceIoControl`
IRP_MJ_DIRECTORY_CONTROL	Zjištění obsahu adresáře (hlavně u zařízení ovladačů souborového systému).	`FindFirstFile`, `FindNextFile`

Těsně před nastavením pole `MajorFunction` ovladač vyplní položku `DriverUnload` své struktury DRIVER_OBJECT adresou procedury, kterou má systém vykonat při uvolnění modulu z jádra. Její kód provádí reverzní operace k funkci `DriverEntry`.

Druhou fázi přípravy na komunikaci, která je zároveň závěrečnou fází inicializace, vidíte na výpisu 3.17.

Výpis 3.17: Druhá fáze přípravy na komunikaci (pokračování rutiny DriverInit z výpisu 3.15)

```
...
    if (NT_SUCCESS(status)) {
      // Nastavime rutinu, kterou ma system zavolat pri odstranovani ovladace
      // z jadra. Neni-li tato rutina nastavena,
      // ovladac nelze z jadra odstranit
      DriverObject->DriverUnload = DriverUnload;

      // Nastavime funkce, ktere system preda rizeni, kdykoliv nejaka aplikace
      // zavola CreateFile ci CloseHandle na nase zarizeni
      DriverObject->MajorFunction[IRP_MJ_CREATE] = DriverCreateClose;
      DriverObject->MajorFunction[IRP_MJ_CLOSE] = DriverCreateClose;

      // Nase odpoved na volani DeviceIoControl
      DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DriverDeviceControl;
    } else IoDeleteDevice(DriverObject->DeviceObject);
  }

  return status;
}
```


Ovladač `logptm.sys` podporuje dynamické uvolnění z paměti jádra. Jinak řečeno, aplikace (či jiná entita) jej může z jádra uvolnit prakticky kdykoliv za běhu operačního systému. Při této operaci však `logptm.sys` musí po sobě uklidit – smazat symbolický odkaz a objekt zařízení, odregistrovat zpětně volané funkce pro sběr událostí a uvolnit paměť zabranou seznamem událostí nepředaných do uživatelského režimu.

Úklid zahajuje procedura `DriverUnload`. Nejprve voláním `DriverFinit` smaže symbolický odkaz a objekt zařízení, takže se k ovladači již nemůže nikdo připojit a přijímat události. Následně dojde k zavolání procedury `NotifyFinit`, která zajistí odregistrování všech zpětně volaných funkcí. Poslední krok spočívá ve volání rutiny `EventsFinit`, která odstraní všechny nezpracované události a uvolní prostředky potřebné na spravování jejich seznamu. Těla všech zmíněných rutin vidíte na výpisu 3.18.

Výpis 3.18: Rutiny DriverFinit, NotifyFinit, EventsFinit a DriverFinit

```
VOID NotifyFinit(VOID)
{
  PsRemoveLoadImageNotifyRoutine(NotifyImage);
  PsRemoveCreateThreadNotifyRoutine(NotifyThread);
  PsSetCreateProcessNotifyRoutine(NotifyProcess, TRUE);
```

```
  return;
}

VOID EventsFinit(VOID)
{
  lListDestroyLock(Events);
  Events = NULL;

  return;
}

VOID DriverFinit(PDRIVER_OBJECT DriverObject)
{
  UNICODE_STRING usymlink;

  // Smazeme symbolicky odkaz, takze dalsi aplikace se nemohou pomoci volani
  // CreateFile pripojit na nase zarizeni.
  RtlInitUnicodeString(&usymlink, LOGPTM_SYMLINK);
  IoDeleteSymbolicLink(&usymlink);

  // Odstranime nase komunikacni zarizeni ze systemu
  IoDeleteDevice(DriverObject->DeviceObject);

  return;
}

VOID DriverUnload(PDRIVER_OBJECT DriverObject)
{
  // Odstranime symbolicky link a objekt zarizeni
  DriverFinit(DriverObject);

  // Odregistrujeme notifikacni rutiny a uvolnime pamet, kterou zabira
  // seznam udalosti
  NotifyFinit();
  EventsFinit();
  return;
}
```

Poznámka: Rutiny `XxxInit` a `XxxFinit` tvoří páry. První rutina zajišťuje inicializaci určitého mechanismu, druhá potřebné úklidové operace.

Komunikace s aplikací

Ovladač `logptm.sys` obsluhuje na svém zařízení následující typy požadavků:

- **Otevření** – aplikace tento požadavek vytvoří a odešle zařízení pomocí funkce Windows API `CreateFile`. Tím se pokusí zařízení otevřít jako soubor.

- **Obecná zpráva** – aplikace může libovolnému zařízení odeslat zprávu v obecném tvaru pomocí funkce Windows API `DeviceIoControl`. Při posílání takové zprávy specifikuje její číselný kód a vyhradí paměť na vstupní a výstupní buffery. Obslužná rutina ovladače, která je zapsána na příslušné pozici v poli `MajorFunction` struktury DRIVER_OBJECT, obdrží číselný kód zprávy, na jehož základě se rozhodne, zda ji dokáže obsloužit. Pokud je odpověď na tuto otázku kladná, ovladač na základě obsahu vstupního bufferu a kódu zprávy provede požadovanou operaci, jejíž výsledky zapíše do výstupního bufferu. Výstupní buffer následně obdrží původní odesilatel zprávy, například aplikace.
- **Ukončení práce se zařízením** – aplikace tento požadavek zašlou pomocí funkce Windows API `CloseHandle`, které jako parametr předají identifikátor zařízení získaný při volání rutiny `CreateFile` (viz první odrážka).

Obsluha požadavků zahájení a ukončení práce se zařízením probíhá v rutině `DriverCreateClose`, kterou najdete na výpisu 3.19. Tato funkce voláním podprogramu `IoCompleteRequest` informuje systém, že obsluha požadavku byla úspěšně dokončena, a skončí.

Výpis 3.19: Rutina DriverCreateClose

```
NTSTATUS DriverCreateClose(PDEVICE_OBJECT DeviceObject, PIRP Irp)
{
  NTSTATUS status = STATUS_UNSUCCESSFUL;

  // Upozornime system, aby povazoval pozadavek (IRP)
  // za obslouzeny a vratime uspech
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
  status = STATUS_SUCCESS;

  return status;
}
```

Obsluhu obecných zpráv má na starosti funkce `DriverDeviceControl`. Na počátku získá z požadavku reprezentovaného strukturou IRP informace o číselném kódu zprávy a velikosti a umístění výstupního bufferu. Rutina se nezabývá zjišťováním informací o vstupním bufferu, protože přijímá pouze jediný druh požadavku, jehož obsluha tento buffer nevyžaduje. Průběh zjišťování těchto údajů vidíte na výpisu 3.20.

Výpis 3.20: DriverDeviceControl – zjišťování informací o požadavku

```
NTSTATUS DriverDeviceControl(PDEVICE_OBJECT DeviceObject, PIRP Irp)
{
  PVOID OutBuffer = NULL;
  ULONG ControlCode = 0;
  ULONG OutBufferLength = 0;
  PIO_STACK_LOCATION IrpStack = NULL;
  NTSTATUS status = STATUS_UNSUCCESSFUL;

  /* Zjistime adresu a velikost vystupniho bufferu a kod zpravy, kterou
     aplikace posila. Velikost vstupniho bufferu nema smysl zjistovat,
     protoze veskere potrebne informace dostaneme v kodu zpravy.
```

```
   */
  IrpStack = IoGetCurrentIrpStackLocation(Irp);
  OutBufferLength = IrpStack->Parameters.DeviceIoControl.OutputBufferLength;
  OutBuffer = Irp->AssociatedIrp.SystemBuffer;
  ControlCode = IrpStack->Parameters.DeviceIoControl.IoControlCode;
  Irp->IoStatus.Information = 0;

  // Zjistime, o ktery prikaz se jedna
  switch (ControlCode) {
    // Poslat aplikaci prvni zpravu ze seznamu udalosti
    case IOCTL_LOGPTM_GET_EVENT: {
...
```

Ovladač `logptm.sys` umí obsluhovat pouze jedinou zprávu, která má za následek odebrání první události ze seznamu a její nakopírování do výstupního bufferu. Pokud se po odebrání události zjistí, že velikost bufferu nedostačuje, je do seznamu opět vrácena a obslužná rutina skončí s chybovým kódem STATUS_BUFFER_TOO_SMALL, který říká odesilateli, k jakému problému došlo. Část zpracování zprávy ukazuje výpis 3.21.

Výpis 3.21: Zpracování zprávy (pokračování rutiny DriverDeviceControl z výpisu 3.18)

```
...
      KIRQL Irql;
      PEVENT e = NULL;

      EventListLock(&Irql);
      // Ziskame prvni udalost ze seznamu udalosti
      e = EventGetFirst();
      if (e!= NULL) {
        // Seznam je neprazdny. Zjistime, zda je vystupni buffer
        // dostatecne velky
        if (EVENT_DATA_SIZE(e) + sizeof(UM_EVENT) <= OutBufferLength) {
          /*
             Nastavime promenne na adresy prvni udalosti ze seznamu
             a vystupniho bufferu. Bude se snadneji kopirovat.
           */
          PUM_EVENT ue = (PUM_EVENT)OutBuffer;

          // Vyplnime obecne polozky udalosti a zkopirujeme jeji data
          ue->Type = e->Type;
          ue->Size = EVENT_DATA_SIZE(e) + sizeof(UM_EVENT);
          ue->Time.QuadPart = e->Time.QuadPart;

          // Prekopirujeme data specificka pro typ udalosti
          RtlCopyMemory(UM_EVENT_TO_DATA(ue), EVENT_TO_DATA(e),
            EVENT_DATA_SIZE(e));

          // Nastavime pocet bajtu,
```

```
            // ktere ma spravce vstupne/vystupnich zarizeni
            // zkopirovat do privatni casti pametoveho prostoru aplikace.
            Irp->IoStatus.Information = ue->Size;
            status = STATUS_SUCCESS;

            // Uvolnime pamet zabranou zaznamem o udalosti
            EventFree(e);
          } else {
            EventInsertFirst(e);
            status = STATUS_BUFFER_TOO_SMALL;
          }
        } else status = STATUS_NO_MORE_ENTRIES;

        EventListUnlock(Irql);
      }
        break;
...
```

Kód zprávy není pouze náhodně zvolené číslo, ale nese v sobě i řadu informací pro správce vstupně/výstupních operací, který podle nich upravuje podmínky při jejím přenosu. Obvykle se definuje pomocí makra CTL_CODE a skládá se z následujících částí:

- **Způsob přenosu vstupního a výstupního bufferu** – určuje, jak má správce nakládat s vstupním a výstupním bufferem během přenosu zprávy. Kód zprávy ovladače `logptm.sys` přikazuje, aby použil tzv. `bufferovanou` metodu (METHOD_BUFFERED). Tento způsob přenosu spočívá v tom, že správce nakopíruje obsah vstupního bufferu do nestránkované paměti a zavolá obslužnou rutinu definovanou v poli `MajorFunction` ve struktuře DRIVER_OBJECT. Obslužná rutina přepíše nestránkovanou paměť obsahující vstupní buffer obsahem výstupního bufferu (blok nestránkované paměti je alokován tak, aby se jeho velikost rovnala maximu z velikostí vstupního a výstupního bufferu). Správce následně překopíruje obsah nestránkované paměti do výstupního bufferu, který specifikoval odesilatel zprávy.

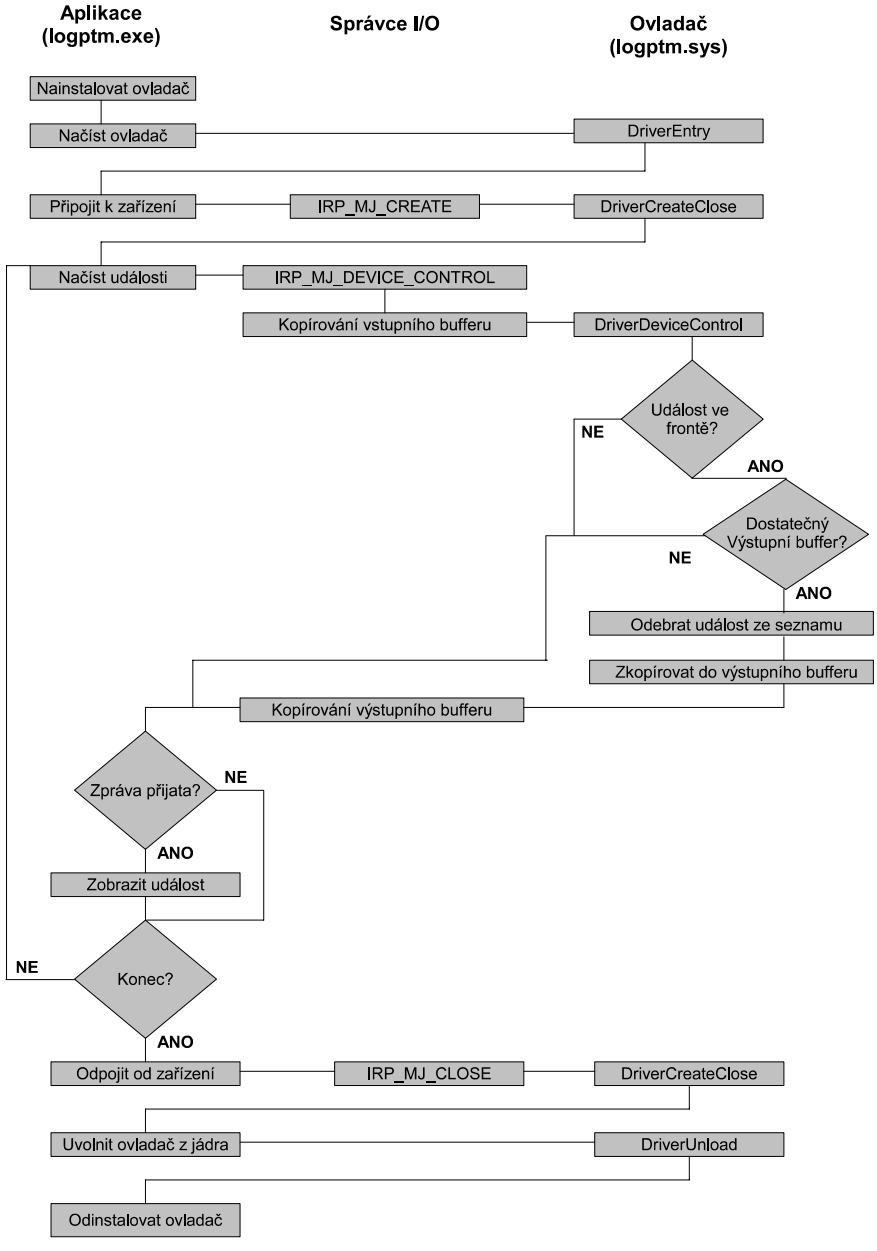
  Tento způsob přenosu přináší řadu výhod. Obslužná rutina nemusí testovat, zda je adresa vstupního a výstupního bufferu platná, protože tuto kontrolu zajistí správce vstupně/výstupních operací. Alokace potřebné paměti probíhají z nestránkovaného fondu, a proto lze čtení a zápis provádět za libovolných okolností.

  Bufferovaná metoda se uplatňuje zejména při méně častých přenosech malého objemu dat, kdy nespotřebovává příliš mnoho paměti z nestránkovaného fondu a režie na alokaci a uvolnění bloku nevytváří úzké hrdlo celé komunikace.

- **Typ zařízení** – musí se shodovat s typem zařízení, které zpráva s příslušným kódem obsluhuje. Zařízení ovladače `logptm.sys` má typ FILE_DEVICE_UNKNOWN.

- **Rozlišovací kód zprávy** – slouží k rozlišení více druhů zpráv pro stejný typ zařízení, které používají stejnou metodu přenosu vstupních a výstupních bufferů. Jedná se o jedinou část kódu zprávy, kterou si můžete zvolit naprosto libovolně.

- **Oprávnění** – určuje, jaká oprávnění k zařízení musí subjekt získat, aby mohl zprávu s daným kódem úspěšně poslat.



Obrázek 3.19: Průběh interakce aplikace s ovladačem logptm.sys

Kód zprávy, kterou obsluhuje ovladač `logptm.sys`, je definován na výpisu 3.22. Při přenosu bufferů se používá bufferovaná metoda a zpráva je určena pro zařízení typu FILE_DEVICE_UNKNOWN. Rozlišovací kód má hodnotu 1 a zprávu mohou posílat pouze subjekty, jež získaly

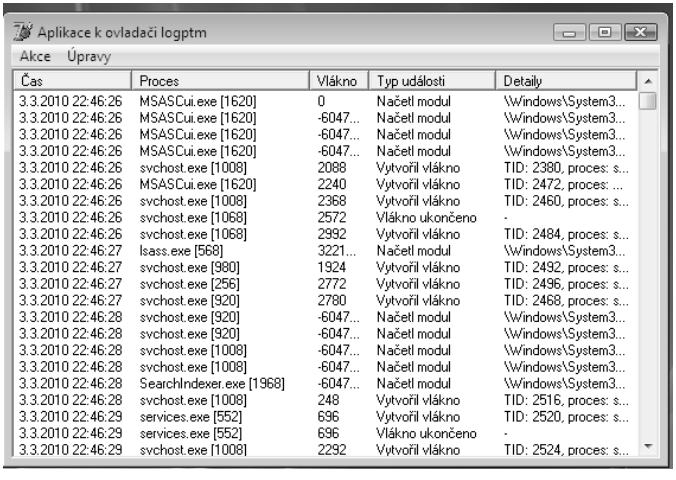
k zařízení všechna přístupová práva, například voláním funkce `CreateFile` s požadavkem na oprávnění GENERIC_ALL.

Výpis 3.22: Definice kódu zprávy

```
#define IOCTL_LOGPTM_GET_EVENT CTL_CODE (FILE_DEVICE_UNKNOWN, 0x01,
  METHOD_BUFFERED, FILE_ALL_ACCESS)
```

Celý průběh komunikace aplikace s ovladačem `logptm.sys` vidíte na obrázku 3.19. Aplikace nejprve nainstaluje službu ovladače pomocí rozhraní správce služeb (SCM) a načte ovladač do jádra. Následně se pomocí rutiny `CreateFile` připojí k zařízení a periodicky posílá zprávu pro získání události. Pokud ovladač zaznamenal nějakou událost, vrátí ji ve výstupním bufferu. Aplikace událost zpracuje a vypíše do okna svého uživatelského rozhraní.

Během ukončování programu dojde k odpojení od zařízení ovladače pomocí volání funkce `CloseHandle`. Následuje uvolnění ovladače z jádra a odinstalování jeho služby. Grafické uživatelské rozhraní aplikace vidíte na obrázku 3.20.



Obrázek 3.20: Grafické uživatelské rozhraní aplikace