ÚVOD

Spracovanie chýb

Skôr než sa začne so skúmaním vymožeností systému Windows, je dobré byť oboznámený s tým, ako Windows funkcie spracovávajú chyby.

Pri volaní Windows funkcie sú vyhodnocované parametre funkcie a následne sa funkcia vykonáva. Pokiaľ je nesprávny parameter, alebo funkcia zlyhá počas spracovania, neúspech je indikovaný návratovou hodnotou funkcie. Nasledujúca tabuľka zobrazuje prehľad najčastejšie používaných dátových typov návratovej hodnoty funkcie:

Dátový typ	Hodnota indikujúca zlyhanie
`VOID`	Táto funkcia prakticky nemôže zlyhať. Len veľmi málo Windows funkcií má návratovú hodnotu typu `VOID`.
`BOOL`	Ak funkcia neuspeje, návratová hodnota je 0, inak je hodnota nenulová. Pri testovaní tejto návratovej hodnoty vždy porovnávajte s hodnotou 0, vyhnite sa porovnávaniu s hodnotou `TRUE`.
`HANDLE`	Ak funkcia neuspeje, návratová hodnota je zvyčajne `NULL`, prípadne `INVALID_HANDLE_VALUE` – je vhodné pozrieť si dokumentáciu funkcie. Inak `HANDLE` identifikuje objekt, s ktorým je možné manipulovať.
`PVOID`	Ak funkcia neuspeje, návratová hodnota je `NULL`, inak `PVOID` identifikuje pamäťovú adresu bloku dát.
`LONG/DWORD`	Ak voláte funkciu s návratovou hodnotou `LONG/DWORD`, pozorne si preštudujte dokumentáciu k tejto funkcii, aby ste sa uistili, že správne pátrate po prípadných chybách.

Keď nejaká Windows funkcia vráti chybový kód, je dobré vedieť, prečo funkcia neuspela. Microsoft spravil zoznam všetkých možných chýb a každej priradil 32-bitové číslo ako chybový kód. Pre získanie tohto kódu použite funkciu *`GetLastError`*:

```
DWORD GetLastError();
```

Po obdržaní 32-bitového chybového kódu je potrebné previesť toto číslo na niečo užitočnejšie. Zoznam chybových kódov definovaných Microsoftom je obsiahnutý v hlavičkovom súbore WinError.h. Každý kód je zapísaný nasledovným spôsobom:

```
//
// MessageId: ERROR_FILE_NOT_FOUND
//
// MessageText:
//
//  The system cannot find the file specified.
//
#define ERROR_FILE_NOT_FOUND             2L
```

Ako je možné vidieť, každá chyba je reprezentovaná tromi spôsobmi: ID správy (makro, ktoré môže byť použité v zdrojovom kóde, napr. pri porovnávaní s návratovou hodnotu funkcie *GetLastError*), textová správa (anglický popis chyby) a číslo (nemalo by sa používať, vhodnejšie je použiť ID chyby).

Ak nejaká Windows funkcia neuspeje, mali by ste hneď volať funkciu *GetLastError*, pretože ak zavoláte ďalšiu Windows funkciu, je veľmi pravdepodobné, že hodnota chybového kódu bude prepísaná.

Po zavolaní funkcie *GetLastError* získate len nič nehovoriace číslo. Pri zlyhaní však väčšinou chcete oboznámiť užívateľa s chybou, ktorá nastala. Pre tento prípad Windows ponúka funkciu *FormatMessage*, ktorá skonvertuje chybový kód na jeho textový popis:

```
DWORD FormatMessage(
       DWORD flags,
       LPCVOID source,
       DWORD messageId,
       DWORD languageId,
       PTSTR buffer,
       DWORD bufferSize,
       va_list *Arguments
       );
```

Pri vytváraní vlastných funkcií je dobrým zvykom používať nasledujúcu štruktúru funkcií:

```
BOOL ErrorSample()
{
       HANDLE fileHandle = NULL;
       BOOL result = TRUE;
       DWORD lastError = ERROR_SUCCESS;

       // volanie nejakej Windows funkcie, v tomto prípade CreateFile
       fileHandle = CreateFile(
              L”C:\\SampleApplication\\ErrorSample.txt”,
              GENERIC_WRITE,
              FILE_SHARE_READ,
              NULL,
              OPEN_ALWAYS,
              0,
              NULL
              );
       if((fileHandle == INVALID_HANDLE_VALUE) || (fileHandle == NULL))
              goto error;

       // zvyšný kód funkcie ErrorSample
       .
       .
       .
       goto final;

error:
       lastError = GetLastError();
       _RPTFW1(_CRT_WARN, L"An error occured, error code : %d.\n", lastError);
       result = FALSE;
final:
       // uvoľnia sa zdroje, ktoré už nebudú používané
```

```
	if((fileHandle != INVALID_HANDLE_VALUE) && (fileHandle != NULL))
		CloseHandle(fileHandle);
	return result;

}
```

Ako je naznačené v zobrazenej časti kódu, po volaní nejakej Windows funkcie sa testuje jej návratová hodnota. Pokiaľ Windows funkcia neuspela, preskočí sa zvyšok tela funkcie, spracuje sa chyba a funkcia je ukončená. Makro `_RPTFWn(reportType, format, [args])`, kde n je počet argumentov v *`args`* pomáha v sledovaní priebehu aplikácie generovaním debugovacích informácií. Aby však bolo možné používať toto makro, je nutné špecifikovať typ upozornení a súbor, do ktorého sa budú zapisovať. To by sa malo špecifikovať hneď po štarte aplikácie, aby bolo možné sledovať jej priebeh od začiatku:

```
// globálna premenná deklarovaná v hlavičkovom súbore Sample.h
HANDLE logFile;
-------------------------------------------------------------------------------
// časť kódu v Sample.c
logFile = NULL;
logFile = CreateFile(
	L"C:\\Sample\\SampleDebug.log",
	GENERIC_WRITE,
	FILE_SHARE_WRITE,
	NULL,
	CREATE_ALWAYS,
	FILE_ATTRIBUTE_NORMAL,
	NULL
	);
if ((logFile == INVALID_HANDLE_VALUE) || (logFile == NULL))
	goto error;

_CrtSetReportMode(_CRT_WARN, _CRTDBG_MODE_FILE | _CRTDBG_MODE_DEBUG);
_CrtSetReportFile(_CRT_WARN, logFile);
_CrtSetReportMode(_CRT_ERROR, _CRTDBG_MODE_FILE | _CRTDBG_MODE_DEBUG);
_CrtSetReportFile(_CRT_ERROR, logFile);

_RPTFW0(_CRT_WARN, L"Starting debug logs.\n");

// kód samotnej aplikácie
...
// pri ukončení aplikácie je uvoľnená logFile handle
if ((logFile != INVALID_HANDLE_VALUE) && (logFile != NULL))
{
	_RPTFW0(_CRT_WARN, L"Attempt to free logFile handle.\n");
	CloseHandle(logFile);
}
```

Jednotlivé `_RPTFWn` makrá môžete použiť na tých miestach v kóde, ktoré sú pre vás významné, a tým môžete sledovať priebeh aplikácie od začiatku až po jej ukončenie. V súbore, do ktorého sa upozornenia zaznamenávajú, sa okrem vami zadaného textu zobrazí aj plná cesta k súboru, v ktorom sa upozornenie vyskytlo, vrátane čísla riadku, na ktorom sa vyskytlo.

UNICODE

Čím bol operačný systém Windows vo svete populárnejší, tým výraznejšia začala byť otázka jeho možností lokalizácie. Až po OS Windows 98 sa pracovalo s ANSI sadou znakov. Prvým systémom od základov postaveným na sade znakov Unicode bol Windows 2000. Pri tejto príležitosti Microsoft vytvoril Windows API pre Unicode. V súčasnosti by ste mali vždy vyvíjať software v Unicode, aj v prípade, že práve neuvažujete o lokalizácii.

Podpora Unicode v C run-time knižnici

Aby bolo možné reťazce Unicode znakov plne využiť, boli definované nové dátové typy. Štandardný hlavičkový súbor String.h, bol zmenený tak, aby v ňom bol definovaný dátový typ `wchar_t`, čo je dátový typ pre Unicode znak:

```
typedef unsigned short wchar_t;
```

Štandardné funkcie pre prácu s reťazcom, ako napríklad `strcpy`, `strchr` a `strcat`, pracujú, samozrejme, len s ANSI reťazcami. Z tohto dôvodu má ANSI C doplnkovú sadu funkcií. Niektoré zo štandarných ANSI C funkcií pre prácu s reťazcom doplnené o ich ekvivalentné Unicode funkcie sú nasledovné:

```
char * strcat(char *, const char *);
wchar_t * wcscat(wchar_t *, const wchar_t *);

char * strchr(const char *, int);
wchar_t * wcschr(const wchar_t *, wchar_t);

int strcmp(const char *, const char *);
int wcscmp(const wchar_t *, const wchar_t *);

char * strcpy(char *, const char *);
wchar_t * wcscpy(wchar_t *, const wchar_t *);

size_t strlen(const char *);
size_t wcslen(const wchar_t *);
```

Všimnite si, že všetky Unicode funkcie začínajú na `wcs`, čo značí *wide character string*. Pri volaní Unicode funkcie, jednoducho nahraďte `str` prefix ANSI funkcie `wcs` prefixom.

Kód obsahujúci explicitné volania `str` alebo `wcs` funkcií nie je ľahko kompilovateľný pre obe sady znakov – ANSI aj Unicode. Je však možné vytvoriť jeden súbor so zdrojovým kódom, ktorý je možné skompilovať pre obe sady znakov. Pre využitie tejto možnosti použite namiesto hlavičkového súboru String.h súbor Tchar.h.

Hlavičkový súbor Tchar.h existuje výhradne pre pomoc s vytváraním ANSI/Unicode generického zdrojového kódu. Pozostáva zo súboru makier, ktoré by ste mali používať vo svojom zdrojovom kóde miesto priameho volania `str` alebo `wcs` funkcií. Ak pri kompilácii definujete `_UNICODE` makro, makrá budú odkazovať na sadu `wcs` funkcií. Ak nebude definované `_UNICODE`, makrá budú odkazovať na sadu `str` funkcií..

Napríklad, v Tchar.h existuje makro `_tcscpy`. Ak `_UNICODE` nie je definované pri pridaní tohto hlavičkového súboru, `_tcscpy` sa rozvinie do ANSI funkcie `strcpy`. Naopak, ak `_UNICODE` je definované, _tcscpy sa rozvinie do Unicode funkcie `wcscpy`. Všetky C run-time funkcie, ktoré majú reťazce ako svoj argument, majú svoje generické makro definované v Tchar.h. Ak namiesto ANSI/Unicode špecifických

názvov funkcií použijete generické makrá, budete na najlepšej ceste vytvoriť kód, ktorý bude môcť byť kompilovaný čisto pre ANSI alebo Unicode.

Hlavičkový súbor Tchar.h však obsahuje aj ďalšie potrebné makrá. Jedno z nich definuje dátový typ TCHAR. Ten je definovaný nasledovne:

```
#ifdef _UNICODE
typedef wchar_t TCHAR;
#else
typedef char TCHAR;
#endif
```

S použitím tohto dátového typu je možné vytvoriť aj ukazateľ na reťazec:

```
TCHAR *errorString = “error”;
```

Na tomto riadku však vzniká problém. Štandardne Microsoft C++ prekladače kompilujú všetky reťazce akoby to boli ANSI reťazce a nie Unicode reťazce. Vo výsledku by to znamenalo, že prekladač skompiluje tento riadok v poriadku v prípade, že `_UNICODE` nebude definované, ale vygeneruje chybu v prípade, že `_UNICODE` definované bude. Aby bol riadok preložený ako Unicode reťazec, bude musieť byť nasledovne prepísaný:

```
TCHAR *errorString = L”error”;
```

Veľké písmeno „L“ pred samotným reťazcom informuje prekladač, že reťazec má byť preložený ako Unicode reťazec. V tomto prípade však opäť dochádza k problému – program bude úspešne preložený len pokiaľ je `_UNICODE` definované. Riešenie ponúka makro `_TEXT`, ktoré na základe toho, či je `_UNICODE` definované, pridá písmeno „L“ pred samotný reťazec. Makro `_TEXT` je definované v súbore Tchar.h nasledovne:

```
#ifdef _UNICODE
#define _TEXT(x) L ## x
#else
#define _TEXT(x) x
#endif
```

S použitím tohto makra je možné prepísať problémový riadok tak, aby bol správne preložený či už `_UNICODE` je definované, alebo nie:

```
TCHAR *errorString = _TEXT(“error”);
```

Unicode dátové typy definované Windowsom

Hlavičkové súbory Windowsu definujú dátové typy v nasledujúcej tabuľke:

Dátový typ	Popis
`WCHAR`	Unicode znak
`PWSTR`	Ukazateľ na Unicode reťazec
`PCWSTR`	Ukazateľ na konštantný Unicode reťazec

Tieto dátové typy vždy odkazujú na Unicode znaky alebo reťazce. Hlavičkové súbory Windowsu taktiež definujú ANSI/Unicode generické dátové typy `PTSTR` a `PCTSTR`. Tieto dátové typy ukazujú buď na ANSI reťazec, alebo na Unicode reťazec v závislosti na tom, či je pri kompilácií definované makro `UNICODE`.

Všimnite si, že v tomto prípade je `UNICODE` makro bez podtržítka. Makro `_UNICODE` je totižto používané pre C run-time hlavičkové súbory a makro `UNICODE` pre Windows hlavičkové súbory. Obvykle je nutné pri kompilovaní definovať obe makrá.

Objekty jadra

Objekty jadra sú používané systémom a aplikáciami na riadenie početných zdrojov ako napríklad procesov, vlákien, súborov, udalostí, mutexov, semafórov,... Tieto objekty sú vytvárané rôznymi funkciami. Funkcia `CreateFile` napríklad vytvorí objekt súboru. Každý objekt jadra je jednoducho pamäťový blok alokovaný jadrom a je prístupný iba jadru. Tento pamäťový blok je dátová štruktúra, ktorej členy uchovávajú informácie o objekte. Niektoré členy (ako napríklad *security descriptor, usage count*, ...) sú rovnaké pre všetky typy objektov, väčšina je však špecifická pre jednotlive typy objektov.

Nakoľko dátové štruktúry objektov jadra sú prístupné iba jadru, pre aplikácie je nemožné lokalizovať ich v pamäti a priamo zmeniť ich obsah. Microsoft zámerne vynucuje tieto obmedzenia, aby bolo isté, že si štruktúry objektov jadra zachovajú konzistentný stav. Toto opatrenie zároveň umožňuje Microsoftu pridať, odstrániť alebo zmeniť členy v týchto štruktúrach bez narušenia akejkoľvek aplikácie.

Ak však nevieme priamo zmeniť tieto štruktúry, ako potom aplikácie pracujú s týmito objektmi jadra? Odpoveďou je, že Windows ponúka sadu funkcií, ktoré pracujú s týmito štruktúrami. Objekty jadra sú vždy prístupné cez tieto funkcie. Keď zavoláte funkciu, ktorá vytvára objekt jadra, funkcia vráti *handle*, ktorá objekt identifikuje. Túto handle potom odovzdávate rôznym Windows funkciám, takže systém vie, s ktorým objektom jadra chcete pracovať.

Usage counting

Objekty jadra vlastní jadro a nie proces. To znamená, že ak váš proces volá funkciu, ktorá objekt jadra vytvorí a potom váš proces skončí, nemusí to nutne znamenať, že objekt jadra bol zničený. Vo väčšine prípadov objekt jadra zničený je, môže však nastať situácia, keď to neplatí. Táto situácia nastáva, keď s objektom jadra pracuje ešte iný proces, než ten, ktorý ho vytvoril. V tomto prípade jadro vie, že nesmie zničiť objekt, kým sa neukončí aj ten druhý proces. Dôležité je zapamätať si, že objekt jadra vie prežiť proces, ktorý ho vytvoril.

Jadro vie, koľko procesov práve používa daný objekt jadra, pretože každý objekt má svoj tzv. *usage count*. *Usage count* je jedným z dátových členov spoločných pre všetky typy objektov jadra. Keď je objekt prvýkrát vytvorený, jeho *usage count* je nastavený na 1. Ak potom ďalší proces získa prístup k už existujúcemu objektu jadra, *usage count* sa inkrementuje. Keď sa proces ukončí, jadro automaticky dekrementuje *usage count* všetkých objektov jadra, ktoré mal daný proces otvorené. Ak sa *usage count* objektu dostane na 0, jadro objekt zničí. To zaisťuje, že žiadny objekt jadra nezostane v systéme potom, čo naň už neodkazuje žiadny proces.

Bezpečnosť

Objekty jadra sú chránené pomocou bezpečnostného deskriptoru (*security descriptor*). Bezpečnostný deskriptor popisuje, kto vytvoril objekt, kto má k nemu prístup, kto ho môže využívať a komu nie je prístup k objektu povolený. Bezpečnostné deskriptory sa zvyčajne používajú pri vytváraní serverových aplikácií; pri písaní klientských aplikácií ich môžete ignorovať.

Takmer všetky funkcie, ktoré vytvárajú objekty jadra majú ako svoj argument ukazateľ na štruktúru `SECURITY_ATTRIBUTES`, ako je možné vidieť na príklade funkcie *`CreateThread`*:

```
HANDLE WINAPI CreateThread(
        LPSECURITY_ATTRIBUTES sa,
        SIZE_T stackSize,
        LPTHREAD_START_ROUTINE startAddress,
        LPVOID parameter,
        DWORD creationFlags,
        LPDWORD threadId
        );
```

Väčšinou je hodnota tohto argumentu `NULL`, čo znamená, že bezpečnosť objektu jadra je nastavená implicitne. V tomto prípade ktorýkoľvek administrátor a tvorca objektu majú plný prístup k objektu. Nikto iný prístup nemá. Každopádne je však možné aj alokovať štruktúru `SECURITY_ATTRIBUTES`, inicializovať ju a odovzdať adresu štruktúry ako argument danej funkcii. Samotná štruktúra vyzerá nasledovne:

```
typedef struct _SECURITY_ATTRIBUTES
{
      DWORD length;
      LPVOID securityDescriptor;
      BOOL inheritHandle;
} SECURITY_ATTRIBUTES;
```

Napriek tomu, že štruktúra sa nazýva `SECURITY_ATTRIBUTES`, zahŕňa iba jeden člen, ktorý má niečo spoločné s bezpečnosťou, a to *`securityDescriptor`*.

Popri objektoch jadra môže aplikácia využívať aj iné objekty, ako napríklad menu, okná, kurzor myši,... Tieto objekty patria medzi objekty užívateľa (*User*) alebo objekty rozhrania grafického zariadenia (*Graphics Device Interface - GDI*), nie medzi objekty jadra. Pri programovaní pre Windows môžete byť zo začiatku zmätení, keď sa pokúsite rozoznať *User* alebo GDI objekty od objektov jadra. Najjednoduchší spôsob, ako rozoznať, či je objekt objektom jadra, je skontrolovať funkciu, ktorá objekt vytvára. Ako už bolo spomenuté, takmer každá funkcia vytvárajúca objekt jadra obsahuje parameter špecifikujúci atribúty bezpečnosti, ako to bolo aj pri funkcii *`CreateThread`* vyššie. Žiadna z funkcií vytvárajúcich *User* alebo GDI objekty parameter `PSECURITY_ATTRIBUTES` nemá.

Tabuľka deskriptorov pre objekty jadra procesu

Pri inicializácií procesu je preň systémom alokovaná tabuľka deskriptorov (*handle table*). Tabuľka deskriptorov je používaná iba pre objekty jadra. Detaily o štruktúre tabuľky deskriptorov a jej riadení nie sú zdokumentované, preto nasledujúci text nebude najpresnejší, ale bude z neho jasné ako systém pracuje.

Tabuľku deskriptorov je možné si predstaviť ako pole dátových štruktúr. Každá z týchto štruktúr obsahuje ukazateľ na objekt jadra, masku prístupu a indikátory. Tabuľku deskriptorov je možné si predstaviť nasledovne:

Index	Ukazateľ na blok pamäte objektu jadra	Maska prístupu	Indikátory
1	0x????????	0x????????	0x????????
2	0x????????	0x????????	0x????????
...	...	...	...

Vytvorenie objektu jadra

Pri inicializácií procesu je tabuľka deskriptorov prázdna. Keď potom vlákno v procese volá funkciu, ktorá vytvorí objekt jadra, jadro alokuje blok pamäte pre objekt a inicializuje ho. Jadro potom hľadá v tabuľke deskriptorov procesu voľný vstup. Nakoľko je tabuľka ešte prázdna, nájde vstup s indexom 1 a inicializuje ho. Štruktúra odpovedajúca tomuto indexu sa naplní – ukazateľu na pamäť je priradená vnútorná pamäťová adresa dátovej štruktúry objektu jadra, maska prístupu bude nastavená na plnú kontrolu a budú nastavené aj indikátory.

Všetky funkcie vytvárajúce objekty jadra vracajú *handle* spätú s procesom, ktorú môže úspešne využívať ktorékoľvek vlákno v procese. Hodnota tejto *handle* je vlastne indexom z tabuľky deskriptorov procesu, ktorý určuje, kde sú uložené informácie o objekte jadra.

Kedykoľvek voláte funkciu, ktorá akceptuje *handle* objektu jadra ako svoj argument, používate hodnotu vrátenú niektorou z funkcií `Create*`. Interne funkcia nazrie do tabuľky deskriptorov procesu, aby získala adresu objektu jadra, s ktorým chcete manipulovať a následne pracuje s dátovou štruktúrou objektu.

Pri volaní `Create*` funkcií je však nutné dať si pozor na návratovú hodnotu v prípade zlyhania funkcie. Väčšinou je pri zlyhaní vrátená hodnota 0 (`NULL`), môže však byť vrátená aj hodnota -1 (`INVALID_HANDLE_VALUE`).

Zatvorenie objektu jadra

Nezávisle na spôsobe vytvorenia objektu jadra, systém je oboznámený na ukončenie manipulovania s objektom jadra volaním funkcie `CloseHandle`:

```
BOOL CloseHandle(HANDLE objectHandle);
```

Táto funkcia najskôr skontroluje tabuľku deskriptorov volajúceho procesu, aby sa uistila, že index (*handle*) odovzdaný funkcii identifikuje objekt, ku ktorému má daný proces prístup. Ak je index platný, systém získa adresu dátovej štruktúry objektu jadra a dekrementuje jeho *usage count*; ak je *usage count* 0, jadro zničí objekt jadra. Ak je index neplatný, funkcia `CloseHandle` vráti hodnotu `FALSE` a funkcia `GetLastError` vráti hodnotu `ERROR_INVALID_HANDLE`.

Skôr, než sa funkcia `CloseHandle` vráti, odstráni vstup v tabuľke deskriptorov procesu – táto *handle* je od tejto chvíle pre proces neplatná a už by sa s ňou nemalo manipulovať. Odstránenie vstupu sa deje nezávisle na tom, či bude objekt jadra zrušený (jeho *usage count* ešte nemusí byť 0)!

Pokiaľ zabudnete zavolať funkciu `CloseHandle`, dôjde k úniku pamäte. K tomuto úniku však bude dochádzať len po dobu, kým bude bežať daný proces. Po ukončení procesu sa o všetky úniky pamäte a zdrojov postará operačný systém. Pre objekty jadra to znamená, že po ukončení procesu systém automaticky prejde tabuľku deskriptorov procesu a uzavrie všetky platné *handles* objektov. Takže vo vašej aplikácii môže dôjsť k únikom zdrojov, kým je spustená, po jej skončení je však garantované systémom, že všetky zdroje sú uvoľnené.

Použitá literatúra:

RICHTER, Jeffrey. CLARK, Jason D. *Programming Server-Side Applications for Microsoft Windows 2000*. 1. vyd. 2000. ISBN 0-7356-0753-2

Microsoft Developer Network, domovská www stránka, dostupná na URL http://msdn.microsoft.com