Vlákna

Je nesmierne dôležité porozumieť vláknam, pretože každý proces má aspoň jedno, primárne vlákno. Tento text sa bude zaoberať tým, ako systém pomocou objektov jadra spravuje vlákna, vlastnosťami vlákien a funkciami, pomocou ktorých je možné s týmito vlastnosťami pracovať.

V predchádzajúcom texte o procesoch bolo uvedené, že proces pozostáva z dvoch komponent: objektu jadra procesu a adresového priestoru. Podobne je to aj pri vláknach, ktoré tiež pozostávajú z dvoch komponent:

- Objektu jadra, ktorý operačný systém používa na riadenie vlákna. V tomto objekte jadra sú taktiež uchovávané štatistické informácie o vlákne.
- Zásobníku vlákna, ktorý uchováva všetky parametre funkcie a lokálne premenné potrebné pri vykonávaní kódu vlákna.

Ako už bolo povedané, procesy sú nečinné. Proces nikdy nič nevykonáva, je len akýmsi zásobníkom pre vlákna. Vlákna sú vždy vytvárané v kontexte nejakého procesu a žijú v ňom počas celého svojho života. V praxi to znamená, že vlákno vykonáva kód a manipuluje s dátami v rámci adresového priestoru procesu. Takže ak bežia dve alebo viac vlákien v kontexte toho istého procesu, zdieľajú ten istý adresový priestor. Vlákna môžu vykonávať ten istý kód a manipulovať s tými istými dátami. Vlákna môžu tiež zdieľať *handles* objektov jadra, pretože tabuľka deskriptorov existuje pre jednotlivé procesy a nie pre jednotlivé vlákna.

Nakoľko vlákna vyžadujú omnoho menej systémových zdrojov, mali by ste sa vždy snažiť vyriešiť prípadný programový problém vytvorením dodatočného vlákna a nie vytvorením nového procesu.

Funkcia vlákna

Každé vlákno musí mať svoju funkciu vstupného bodu. Funkciami vstupného bodu pre primárne vlákno je niektorá z funkcií *(w)main* alebo *(w)WinMain*. Funkcie vstupných bodov prípadných ďalších vlákien by mali vyzerať nasledovne:

```
DWORD WINAPI ThreadFunc(PVOID param)
{
    DWORD result = 0;
    .
    .
    .
    return result;
}
```

Táto funkcia môže splniť akúkoľvek úlohu. Keď funkcia príde na koniec, vráti sa. V tomto okamihu vlákno skončilo, pamäť pre jeho zásobník je uvoľnená a *usage count* objektu jadra vlákna je dekrementovaný. Ak je *usage count* 0, objekt jadra vlákna je zničený. Tak ako pri objektoch jadra procesu, aj pri objektoch jadra vlákna platí, že objekt jadra vlákna žije prinajmenšom tak dlho, ako vlákno, s ktorým je asociovaný, ale môže žiť aj dlhšie.

Dôležité je uvedomiť si niekoľko zásadných vecí o funkciách vstupných bodov vlákien:

- Na rozdiel od funkcie vstupného bodu primárneho vlákna, ktorá musí byť pomenovaná *(w)main* alebo *(w)WinMain*, pomenovanie vstupných bodov ďalších vlákien môže byť ľubovoľné. Ak je v aplikácií viac vlákien, musí mať každá z ich funkcií iné meno, inak by si prekladač mohol myslieť, že sú vytvorené rôzne implementácie tej istej funkcie.

- Význam parametru funkcie vlákna je definovaný vývojárom a nie systémom.
- Funkcia vlákna musí vrátiť hodnotu, ktorá bude exit kódom vlákna.
- Funkcia vlákna by mala čo najviac používať parametre a lokálne premenné. Pri použití statických a globálnych premenných sa totižto môže stať, že k danej premennej získa prístup viac vlákien naraz a poškodí jej obsah. Na druhej strane, parametre a lokálne premenné sú vytvorené v zásobníku vlkána a preto je nepravdepodobné, žeby ich mohlo poškodiť iné vlákno.

Keď je už jasné, ako implementovať funkciu vlákna, pozrime sa bližšie na to, ako systém vytvorí vlákno, ktoré túto funkciu vykoná.

Vytvorenie vlákna

Vytvorenie nového vlákna má na starosti funkcia *Create-Thread*, ktorá je volaná z primárneho, prípadne iného vlákna:

```
HANDLE CreateThread(
    PSECURITY_ATTRIBUTES threadAttributes,
    DWORD stackSize,
    PTHREAD_START_ROUTINE startAddress,
    PVOID param,
    DWORD creationFlags,
    PDWORD threadID
    );
```

Po zavolaní funkcie *CreateThread* systém vytvorí objekt jadra vlákna. Tak ako pri objektoch jadra procesu, objekt jadra vlákna nie je samotné vlákno, ale malá dátová štruktúra, ktorú operačný systém používa na správu vlákna.

Systém alokuje pamäť z adresového priestoru procesu. Táto pamäť je použitá ako zásobník vlákna. Nové vlákno beží v tom istom kontexte procesu ako vlákno, ktoré ho vytvorilo. Nové vlákno má preto prístup ku všetkým objektom jadra procesu, k pamäti procesu a k zásobníkom všetkých vlákien v rámci tohto procesu.To uľahčuje vzájomnú komunikáciu medzi jednotlivými vláknami v tom istom procese.

Jednotlivé parametre funkcie *CreateThread* a ich význam:

- *threadAttributes* –ukazateľ na štruktúru SECURITY_ATTRIBUTES. Pre implicitné nastavenie bezpečnosti zvoľte tento parameter NULL.
- *stackSize* – iniciálna veľkosť zásobníka v bajtoch. Ak je tento parameter 0, bude použitá implicitná hodnota.
- *startAddress* – indikuje adresu funkcie vlákna, ktorú chcete aby vykonávalo nové vlákno.
- *param* – tento parameter je odovzdaný ako *param* argument funkcii vlákna.
- *creationFlags* – tento parameter môže nadobúdať tri rôzne hodnoty. Ak je parameter 0, vlákno je spustené hneď po vytvorení. Ak je parameter CREATE_SUSPENDED, systém vlákno vytvorí a inicializuje, ale následne je uvedené do suspendovaného stavu. Z tohto stavu sa vráti až po volaní funkcie *ResumeThread*. Hodnota STACK_SIZE_PARAM_IS_A_RESERVATION sa týka parametru *stackSize*.
- *threadID* – ukazateľ na premennú, ktorá obdrží hodnotu identifikátora vlákna. Ak je tento parameter NULL, žiadna hodnota nebude navrátená.

Funkcia *CreateThread* je síce Windows funkcia, ktorá vytvára nové vlákno, avšak pri písaní C/C++ kódu by ste ju nikdy nemali volať priamo. Namiesto toho by ste mali volať funkciu CRT knižnice *_beginthreadex*. Táto funkcia existuje iba v multithreadovej verzii CRT knižnice a okrem volania funkcie *CreateThread* zaisťuje bezpečné použitie premenných a funkcií CRT knižnice v multithreadovej aplikácii.

Ukončenie vlákna

Vlákno je možné ukončiť nasledujúcimi štyrmi spôsobmi:

- Funkcia vlákna sa vráti. (Odporúčaný spôsob ukončenia.)
- Vlákno sa ukončí samé volaním funkcie *ExitThread*. (Vyhnite sa tejto metóde.)
- Vlákno v tom istom, alebo inom procese zavolá funkciu *TerminateThread*. (Vyhnite sa tejto metóde.)
- Proces, ktorý vlákno obsahuje, skončí. (Vyhnite sa tejto metóde.)

Poznámka:
Pri funkcii ExitThread platí, podobne ako pri funkcii CreateThread, že by v C/C++ kóde nikdy nemala byť volaná priamo. Miesto nej by mala byť volaná funkcia CRT knižnice _endthreadex.

Jednotlivé tieto spôsoby ukončení vlákna sú veľmi podobné spôsobom ukončenia procesu, preto sa v tomto texte nebudú bližšie rozvádzať.

Pri samotnom ukončení vlákna dochádza k nasledujúcim akciám:

- Všetky *handles* User objektov vlastnené vláknom sú uvoľnené.
- Exit kód vlákna sa zmení zo STILL_ACTIVE na kód odovzdaný funkcii *ExitThread* alebo *TerminateThread*.
- Stav objektu jadra vlákna je signalizovaný.
- Ak je vlákno posledným aktívnym vláknom procesu, systém ukončí aj proces.
- *Usage count* objektu jadra vlákna je dekrementovaný.

Získanie vlastnej identity

Často sa stáva, že vlákna pri vykonávaní volajú Windows funkcie, ktoré menia prostredie ich výkonu. Napríklad vlákno môže chcieť zmeniť svoju prioritu, prípadne prioritu jeho procesu. Nakoľko je pre vlákna bežné meniť svoje prostredie, Windows ponúka funkcie, ktoré uľahčujú vláknu odkázať sa na objekt jadra jeho procesu, prípadne na jeho vlastný objekt jadra vlákna:

```
HANDLE GetCurrentProcess();
HANDLE GetCurrentThread();
```

Obe tieto funkcie vrátia *pseudo-handle* objektu jadra procesu (vlákna) volajúceho vlákna. Ani jedna z týchto funkcií nevytvára novú *handle* v tabuľke deskriptorov volajúceho procesu. Volanie týchto funkcií taktiež nemá žiadny účinok na *usage count* objektu jadra procesu alebo vlákna. Ak zavoláte funkciu *CloseHandle* s touto *pseudo-handle* ako parametrom, funkcia *CloseHandle* volanie jednoducho ignoruje a vráti FALSE.

Pri volaní Windows funkcie, ktorá vyžaduje *handle* procesu alebo vlákna, môžete použiť *pseudo-handle* a funkcia vykoná svoju akciu na volajúcom procese alebo vlákne. Vlákno môže napríklad zistiť odpracovaný čas jeho procesu volaním funkcie *GetProcessTimes*:

```
FILETIME creationTime, exitTime, kernelTime, userTime;
GetProcessTimes(GetCurrentProcess(),
    &creationTime, &exitTime, &kernelTime, &userTime);
```

Pozastavenie a obnovenie vlákna

Objekt jadra vlákna obsahuje hodnotu, ktorá uchováva tzv. *suspend count* vlákna. Po zavolaní funkcie *CreateProcess* alebo *CreateThread* je vytvorený objekt jadra vlákna a jeho *suspend count* je inicializovaný na hodnotu 1. Vlákno tým predíde tomu, aby mu bola priradená CPU jednotka. To je, samozrejme, požadované chovanie, nakoľko vlákno je nutné najkôr inicializovať.

Po tom, čo je vlákno plne inicializované, funkcia *CreateProcess* resp. *CreateThread* skontroluje, či jej bol odovzdaný príznak CREATE_SUSPENDED. Ak áno, funkcia sa vráti a vlákno ostane v pozastavenom stave. Ak nie, funkcia dekrementuje *suspend count* vlákna na 0. Keď je *suspend count* vlákna 0 a vlákno nečaká na žiadnu udalosť, je možné mu priradiť CPU jednotku.

Vytvorenie vlákna v pozastavenom stave umožňuje zmeniť prostredie vlákna (napr. prioritu) skôr, než má vlákno príležitosť vykonať akýkoľvek kód. Pokiaľ je prostredie vlákna zmenené, musí byť toto vlákno hneď obnovené. Vlákno obnovíte volaním funkcie *ResumeThread*:

```
DWORD ResumeThread(HANDLE threadHandle);
```

Ak je funkcia *ResumeThread* úspešná, vráti predchádzajúci *suspend count* vlákna, inak vráti 0xFFFFFFFF.

Samotné vlákno môže byť pozastavené aj niekoľkokrát. Ak je, napríklad, vlákno pozastavené 3 krát, skôr než bude možné mu priradiť CPU jednotku, bude musieť byť 3 krát obnovené. Okrem použitia príznaku CREATE_SUSPENDED je možné vlákno pozastaviť aj volaním funkcie *SuspendThread*:

```
DWORD SuspendThread(HANDLE threadHandle);
```

Ktorékoľvek vlákno môže zavolať túto funkciu na pozastavenie iného vlákna, pokiaľ má *handle* tohto vlákna. Vlákno môže pozastaviť samo seba, ale je jasné, že sa nemôže samo obnoviť. Tak ako funkcia *ResumeThread*, aj funkcia *SuspendThread* vracia predchádzajúci *suspend count* vlákna. Vlákno môže byť pozastavené najviac MAXIMUM_SUSPEND_COUNT krát (vo WinNT.h definované ako 127).

Pri volaní funkcie *SuspendThread* však musíte byť mimoriadne opatrní, keďže neviete, čo práve robí vlákno, ktoré chcete pozastaviť. Môže tak ľahko dôjsť k uviaznutiu.

V prípade potreby je možné vlákno aj uspať, a to v okamihu, keď vlákno určitú presnú dobu nechce, aby mu bola pridelená CPU jednotka. Vlákno sa uspáva volaním funkcie *Sleep*:

```
VOID Sleep(DWORD milliseconds);
```

Synchronizácia vlákien v užívateľskom režime

Všetky vlákna v systéme musia mať prístup k systémovým zdrojom. Ak si niektoré vlákno vyžiada výlučný prístup k nejakému zdroju, ostatné vlákna nemôžu pokračovať vo svojej práci. Na druhej strane však nie je možné povoliť, aby ktorékoľvek vlákno malo kedykoľvek prístup ku všetkým zdrojom.

Vlákna potrebujú spolu komunikovať v dvoch základných situáciách:

- Keď niekoľko vlákien pristupuje ku zdieľanému zdroju.
- Keď vlákno potrebuje upozorniť iné vlákno, prípadne viac vlákien, že bola vykonaná nejaká špecifická úloha.

V užívateľskom režime je na synchronizáciu možné využiť buď rodinu *interlocked* funkcií, alebo kritické sekcie. *Interlocked* funkcie pracujú iba s jednou hodnotou a nikdy neuvedú vlákno do čakajúceho stavu. Typickým príkladom je funkcia *InterlockedExchangeAdd*:

```
LONG InterlockedExchangeAdd(
    LONG volatile *Addend,
    LONG value
    );
```

Táto funkcia inkrementuje premennú *Addend* o hodnotu value. Pri *interlocked* funkciách je dôležité vedieť, že funkcia garantuje, že hodnota, ktorú chcete meniť bude zmenená atomicky, s čím súvisí aj rýchlosť ich spracovania. Nakoľko však tieto funkcie pracujú len s jednoduchými hodnotami, omnoho užitočnejšie je použitie kritických sekcií, ktoré umožňujú „atomický“ prístup aj k omnoho sofistikovanejším dátovým štruktúram.

Kritická sekcia je malá časť kódu, ktorá vyžaduje výlučný prístup ku zdieľanému zdroju skôr, než bude kód pokračovať. Toto je spôsob, ako zabezpečiť, aby niekoľko riadkov kódu „atomicky“ manipulovalo so zdrojom. Slovo „atomicky“ v tomto prípade znamená, že kód vie, že žiadne iné vlákno v tom čase nemá k danému zdroju prístup.

Pokiaľ máte zdroj, ku ktorému pristupuje viac vlákien , mali by ste vytvoriť štruktúru `CRITICAL_SECTION`. Štruktúry `CRITICAL_SECTION` sú väčšinou alokované ako globálne premenné, aby sa na ne mohli ľahko odkazovať všetky vlákna v procese. Inicializácia prebieha volaním funkcie *InitializeCriticalSection*:

```
VOID InitializeCriticalSection(PCRITICAL_SECTION cs);
```

Táto funkcia jednoducho nastaví niektoré členy štruktúry. Funkcia musí byť volaná skôr, než ktorékoľvek vlákno zavolá funkciu *EnterCriticalSection*. Pokiaľ viete, že vlákna v procese sa už nepokúsia získať prístup ku zdieľaným zdrojom, mali by ste upratať štruktúru `CRITICAL_SECTION` volaním tejto funkcie:

```
VOID DeleteCriticalSection(PCRITICAL_SECTION cs);
```

Funkcia *DeleteCriticalSection* vymaže členy štruktúry. Prirodzene, nemali by ste zmazať štruktúru, ktorá je ešte používaná.

Skôr než vaša aplikácia začne vykonávať kód, ktorý pracuje so zdieľaným zdrojom, musí zavolať funkciu *EnterCriticalSection*:

```
VOID EnterCriticalSection(PCRITICAL_SECTION cs);
```

Funkcia skontroluje členy vo vnútri štruktúry, ktoré by mali indikovať, či nejaké vlákno má v tom okamihu prístup ku zdieľanému zdroju. Funkcia robí nasledovné testy:

- Ak ku zdieľanému zdroju nepristupuje žiadne vlákno, funkcia aktualizuje členy štruktúry tak, aby indikovali, že volajúcemu vláknu bol zaručený prístup a hneď sa vráti, aby vlákno mohlo okamžite pokračovať vo vykonávaní kódu.
- Ak členy štruktúry indikujú, že prístup ku zdieľanému zdroju bol zaručený volajúcemu vláknu, funkcia aktualizuje premenné tak, aby indikovali koľkokrát má volajúce vlákno zaručený prístup ku zdieľanému zdroju. Funkcia potom okamžite skončí, aby vlákno mohlo pokračovať. Táto situácia je veľmi zriedkavá a nastane iba ak vlákno volá funkciu *EnterCriticalSection* viackrát po sebe bez toho, aby medzitým bola volaná funkcia *LeaveCriticalSection*.
- Ak členy štruktúry indikujú, že niektoré iné vlákno získalo prístup ku zdieľanému zdroju, funkcia umiestni volajúce vlákno do čakajúceho stavu. To je skvelé vzhľadom na to, že nedochádza k mrhaniu času procesoru. Systém si pamätá , že vlákno chce prístup k danému zdroju a automaticky aktualizuje členy štruktúry `CRITICAL_SECTION` a prebudí vlákno v okamihu, keď vlákno, ktoré dovtedy pristupovalo ku zdieľanému zdroju, zavolá funkciu *LeaveCriticalSection*.

Funkcia *EnterCriticalSection* vlastne spraví len niekoľko testov. Hodnotnou ju však robí to, že tieto testy robí atomicky. Aj keď dve rôzne vlákna zavolajú funkciu *EnterCriticalSection* v presne tom istom čase na multiprocesorovom počítači, funkcia sa bude chovať korektne: jedno vlákno získa prístup ku zdieľanému zdroju, druhé bude umiestnené do čakajúceho stavu.

Po skončení práce so zdieľaným zdrojom je nutné okamžite zavolať túto funkciu:

```
VOID LeaveCriticalSection(PCRITICAL_SECTION cs);
```

Funkcia preskúma členy štruktúry `CRITICAL_SECTION` a dekrementuje o 1 premennú, ktorá indikuje, koľkokrát bol volajúcemu vláknu udelený prístup ku zdieľanému zdroju. Ak je potom táto hodnota väčšia ako 0, funkcia sa vráti. Ak je táto hodnota 0, funkcia skontroluje, či nejaké iné vlákna čakajú na prístup ku zdieľanému zdroju. Ak áno, aktualizuje členy štruktúry a prebudí jedno z čakajúcich vlákien. Ak nie, členy štruktúry sú aktualizované tak, aby indikovali, že žiadne vlákno práve nepristupuje ku zdieľanému zdroju.

Synchronizácia vlákien pomocou objektov jadra

Kým synchronizácia vlákien v užívateľskom režime ponúka vysoký výkon, majú svoje obmedzenia a pre mnoho aplikácií jednoducho nie sú vhodné. V tomto prípade prichádza do úvahy synchronizácia vlákien pomocou objektov jadra. Jediná nevýhoda tohto spôsobu synchronizácie je v nižšom výkone, čo súvisí s tým, že volajúce vlákno sa musí prepnúť z užívateľského režimu do režimu jadra.

Takmer všetky objekty jadra je možné použiť k synchronizácií vlákien. Každé z týchto objektov jadra sa nachádza buď v signalizovanom, alebo nesignalizovanom stave. Prepínanie tohto stavu je determinované pravidlami, ktoré sú vytvorené pre každý objekt. Napríklad objekt jadra procesu je vždy vytvorený v nesignalizovanom stave. Keď je proces ukončený, operačný systém ho automaticky signalizuje. V okamihu, keď je objekt jadra procesu signalizovaný, zostane tak už navždy. Signalizácia prebieha tak, že vo vnútri objektu jadra procesu je hodnota typu *Boolean*, ktorá je pri nesignalizovanom stave nastavená na hodnotu `FALSE` a pri signalizovaní je zmenená na hodnotu `TRUE`.

Ak chcete napísať kód, ktorý by kontroloval, či nejaký proces stále beží, všetko, čo musíte urobiť, je zavolať funkciu, ktorá požiada operačný systém o skontrolovanie *Boolean* hodnoty objektu jadra procesu. Tiež môžete chcieť, aby systém umiestnil vaše vlákno do čakajúceho stavu a automaticky ho zobudil, keď sa *Boolean* hodnota zmení z `FALSE` na `TRUE`. K tomuto účelu slúžia tzv. *Wait* funkcie.

Wait funkcie

Wait funkcie spôsobujú, že vlákno sa dobrovoľne umiestni do čakacieho stavu, až kým nie je signalizovaný špecifický objekt jadra. Najbežnejšou z týchto funkcií je funkcia *WaitForSingleObject*:

```
DWORD WaitForSingleObject(
    HANDLE objectHandle,
    DWORD milliseconds
    );
```

Prvý parameter tejto funkcie, *objectHandle*, identifikuje objekt jadra, druhý parameter, *milliseconds*, umožňuje vláknu indikovať, ako dlho je ochotné čakať na signalizovanie objektu. Napríklad ak chcete zistiť, či bol proces, identifikovaný parametrom *processHandle*, ukončený, zavolajte funkciu nasledovne:

```
WaitForSingleObject(processHandle, INFINITE);
```

Hodnota `INFINITE` udáva, že vlákno je ochotné čakať až kým daný objekt nebude signalizovaný – v tomto prípade, že daný proces bude ukončený. Pokiaľ nechcete čakať večne, je možné nastaviť aj konkrétnu hodnotu času, ktorý ste ochotní čakať. Použitie funkcie v tomto prípade je nasledovné:

```
DWORD result = WaitForSingleObject(processHandle, 5000);
switch(result){

    case WAIT_OBJECT_0:
       // proces bol ukončený
       break;

    case WAIT_TIMEOUT:
       // proces nebol ukončený včas
       break;

    case WAIT_FAILED:
       // neplatná handle??
       break;
}
```

Ďalšou zo skupiny *Wait* funkcií je funkcia *WaitForMultipleObject*, ktorá umožňuje súbežne sledovať hneď niekoľko objektov jadra:

```
DWORD WaitForMultipleObject(
    DWORD count,
    CONST HANDLE* objects,
    BOOL waitAll,
    DWORD milliseconds
    );
```

Parameter count udáva počet objektov jadra, ktoré má funkcia kontrolovať. Táto hodnota musí byť medzi 1 a MAXIMU_WAIT_OBJECTS (definované ako 64). Parameter *objects* je ukazateľ na pole objektov jadra.

Funkciu *WaitForMultipleObjects* je možné použiť dvomi spôsobmi – umožniť vláknu vstúpiť do čakajúceho stavu, až kým jeden zo špecifikovaných objektov jadra nie je signalizovaný, alebo až kým všetky z týchto objektov nie sú signalizované. K tomu slúži parameter *waitAll*.

Príčinu, ktorá spôsobila, že vlákno sa prebudí, je možné zistiť z návratovej hodnoty funkcie. Návratové hodnoty WAIT_FAILED a WAIT_TIMEOUT sú samovysvetľujúce. Ak je parameter *waitAll* TRUE a všetky objekty boli signalizované, návratová hodnota je WAIT_OBJECT_0. Ak je parameter *waitAll* FALSE, funkcia sa vráti hneď ako je ktorýkoľvek z daných objektov signalizovaný. V tomto prípade je možné zistiť, ktorý z objektov bol signalizovaný podľa návratovej hodnoty funkcie. Návratová hodnota je medzi hodnotami WAIT_OBJECT_0 a (WAIT_OBJECT_0 + *count* - 1) a označuje index v poli objektov jadra udaného parametrom *objects*. Tento index určuje, ktorý z objektov bol signalizovaný.

Názorný príklad použitia:

```
HANDLE objects[3];
objects[0] = processHandle1;
objects[1] = processHandle2;
objects[2] = processHandle3;

DWORD result = WaitForMultipleObjects(3, objects, FALSE, 5000);
switch(result){

    case WAIT_FAILED:
       // neplatná handle??
       break;

    case WAIT_TIMEOUT:
       // žiadny z objektov nebol signalizovaný za 5000 milisekúnd
       break;
```

```
    case WAIT_OBJECT_0 + 0:
       // process indentifikovaný s object[0] (procesHandle1) bol ukončený
       break;

    case WAIT_OBJECT_0 + 1:
       // process indentifikovaný s object[1] (procesHandle2) bol ukončený
       break;

    case WAIT_OBJECT_0 + 2:
       // process indentifikovaný s object[2] (procesHandle3) bol ukončený
       break;
}
```

Udalosti

Zo všetkých objektov jadra sú najprimitívnejšie udalosti. Obsahujú *usage count*, *Boolean* hodnotu určujúcu, či je udalosť automaticky nastaviteľná, alebo manuálne nastaviteľná a ďalšiu *Boolean* hodnotu, ktorá určuje, či je udalosť signalizovaná, alebo nesignalizovaná.

Udalosti dávajú znamenie, či je nejaká operácia hotová. Sú dva rôzne typy udalostí: automaticky nastaviteľné udalosti a manuálne nastaviteľné udalosti. Ak je signalizovaná manuálne nastaviteľná udalosť, prebudia sa všetky vlákna, ktoré na ňu čakajú. Ak je signalizovaná automaticky nastaviteľná udalosť, prebudí sa len jedno z čakajúcich vlákien.

Udalosti sa bežne používajú, ak jedno vlákno robí nejakú inicializáciu, a potom signalizuje inému vláknu, aby spravilo zostávajúcu prácu. Udalosť je inicializovaná ako nesignalizovaná a po tom, ako vlákno dokončí počiatočnú prácu, nastaví udalosť ako signalizovanú. V tomto okamihu sa vlákno, ktoré doposiaľ čakalo, prebudí a vykoná zvyšok práce.

Udalosť je možné vytvoriť pomocou funkcie *`CreateEvent`*:

```
HANDLE CreateEvent(
    PSECURITY_ATTRIBUTES sa,
    BOOL manualReset,
    BOOL initialState,
    PCTSTR name
    );
```

Po vytvorení udalosti systém vráti *handle* objektu udalosti spätú s procesom. Vlákna v iných procesoch k nej môžu získať prístup volaním funkcie *`CreateEvent`* s použitím toho istého *`name`* parametru, použitím dedičnosti, použitím funkcie *`DuplicateHandle`*, alebo volaním funkcie *`OpenEvent`* a špecifikovaním parametru name tak, aby súhlasilo s parametrom name funkcie *`CreateEvent`*:

```
HANDLE OpenEvent(
    DWORD access,
    BOOL inherit,
    PCTSTR name
    );
```

V okamihu, keď je udalosť vytvorená, priamo kontrolujete jej stav. Zavolaním funkcie *`SetEvent`*, zmeníte stav udalosti na signalizovaný:

```
BOOL SetEvent(HANDLE eventHandle);
```

Zmenu stavu udalosti na nesignalizovaný vykoná funkcia *`ResetEvent`*:

```
BOOL ResetEvent(HANDLE eventHandle);
```

Waitable timers

Waitable timers sú objekty jadra, ktoré sú signalizované v určitom čase, alebo v pravidelných intervaloch. Najčastejšie sú tieto objekty využívané, keď je nutné vykonať nejakú akciu v určitom čase. *Waitable timer* je vytvorený volaním funkcie `CreateWaitableTimer`:

```
HANDLE CreateWaitableTimer(
    PSECURITY_ATTRIBUTES sa,
    BOOL manualReset,
    PCTSTR name
    );
```

Parameter `manualReset`, podobne ako pri udalostiach, určuje, či sa jedná o manuálne nastaviteľný *waitable timer*, alebo o automaticky nastaviteľný *waitable timer*. Význam manuálne/automaticky nastaviteľného objektu je rovnaký ako pri udalostiach.

Waitable timer objekty sú vytvorené vždy v nesignalizovanom stave. Aby boli signalizované, musíte volať funkciu `SetWaitableTimer`:

```
BOOL SetWaitableTimer(
    HANDLE timerHandle,
    Const LARGE_INTEGER *dueTime,
    LONG period,
    PTIMERAPCROUTINE completionRoutine,
    PVOID argToCompletionRoutine,
    BOOL resume
    );
```

Parametre `dueTime` a `period` sa používajú spoločne. Parameter `dueTime` určuje, kedy má byť časovač prvýkrát signalizovaný a parameter `period` určuje, ako často sa má signalizovať potom.

Semafóry

Semafóry sú objekty jadra používané k počítaniu zdrojov. Tak ako všetky objekty jadra, obsahujú *usage count*. Okrem toho však obsahujú ešte dve ďalšie hodnoty: maximálny počet zdrojov a aktuálny počet zdrojov. Maximálny počet zdrojov určuje koľko zdrojov je schopný semafór kontrolovať a aktuálny počet zdrojov uvádza, koľko zdrojov je z nich aktuálne dostupných.

Pravidlá pre semafór sú nasledovné:

- Ak je aktuálny počet zdrojov väčší ako 0, semafór je signalizovaný.
- Ak je aktuálny počet zdrojov 0, semafór je nesignalizovaný.
- Aktuálny počet zdrojov nesmie byť nikdy záporný.
- Aktuálny počat zdrojov nesmie byť nikdy väčší ako maximálny počet zdrojov.

Objekta jadra semafóru vytvorí funkcia `CreateSemaphore`:

```
HANDLE CreateSemaphore(
    PSECURITY_ATTRIBUTE sa,
    LONG initialCount,
    LONG maximumCount,
    PCTSTR name
    );
```

Parameter `initialCount` označuje aktuálny počet zdrojov pri inicializácií semafóru. Spravidla býva rovnaký ako `maximumCount`, prípadne je rovný 0. Ak je `initialCount` 0, prebehne inicializácia semafóru. Naplno však bude môcť byť využitý až po zmene aktuálneho počtu zdrojov, a to pomocou volania funkcie `ReleaseSemaphore`:

```
BOOL ReleaseSemaphore(
    HANDLE semaphoreHandle,
    LONG releaseCount,
    PLONG previousCount
    );
```

Táto funkcia jednoducho nastaví aktuálny počet zdrojov na hodnotu *releaseCount*. Táto hodnota však nesmie byť väčšia ako maximálny počet zdrojov.

Pokiaľ je semafór signalizovaný (sú dostupné niektoré ním kontrolované zdroje), vlákno, ktoré naň čaká, získa prístup a zároveň dekrementuje aktuálny počet zdrojov semafóru. Po skončení práce s daným zdrojom, je zdroj opäť uvoľnený a aktuálny počet zdrojov semafóru je inkrementovaný.

Mutexy

Mutexy ako objekty jadra zabezpečujú vzájomné vylúčenie v prístupe ku zdieľanému zdroju. Mutex obsahuje *usage count*, hodnotu identifikujúcu vlákno (*thread ID*), ktoré ho v danom okamihu vlastní a hodnotu určujúcu koľkokrát je mutex vlastnený daným vláknom (*recursion counter*). Chovanie mutexov je identické s chovaním kritických sekcií, ale mutexy sú objektami jadra, kým kritické sekcie sú objektami užívateľského módu. To znamená, že mutexy sú pomalejšie ako kritické sekcie, ale tiež to znamená, že prístup k mutexu môže získať aj vlákno z iného procesu a je možné aj špecifikovať dobu, ktorú je dané vlákno ochotné čakať na uvoľnenie mutexu.

Na vytvorenie mutexu slúži funkcia *CreateMutex*:

```
HANDLE CreateMutex(
    PSECURITY_ATTRIBUTES sa,
    BOOL initialOwner,
    PCTSTR name
    );
```

Parameter *initialOwner* určuje počiatočný stav mutexu. Ak je hodnota `FALSE` (väčšinou), hodnoty *thread ID* a *recursion counter* sú nastavené na 0 – mutex nie je vlastnený žiadnym vláknom a je preto signalizovaný. Ak je hodnota `TRUE`, *thread ID* je nastavené na hodnotu identifikátora vlákna a *recursion counter* na hodnotu 1 – mutex je vlastnený vláknom a je preto nesignalizovaný.

V okamihu, keď vlákno získa prístup k mutexu, vie, že má výlučný prístup. Všetky ostatné vlákna usilujúce o získanie mutexu sú umiestnené do čakajúceho stavu.Keď vlákno s aktuálnym prístupom ku zdieľanému zdroju už prístup nepotrebuje, musí mutex uvoľniť volaním funkcie *ReleaseMutex*:

```
BOOL ReleaseMutex(HANDLE mutexHandle);
```

Pri mutexoch sa však môže stať, že vlákno, ktoré ich aktuálne vlastní sa náhle ukončí. Nakoľko v tomto prípade vlákno nestihne mutex uvoľniť, mutex je stále v nesignalizovanom stave a jeho *thread ID* má hodnotu už ukončeného vlákna. Túto situáciu rieši systém, ktorý automaticky považuje mutex za *abandoned* a uvoľní jeho *threadID* aj *recursion counter* hodnoty, čím je mutex opäť signalizovaný. *Wait* funkcia nejakého vlákna, ktorá čaká práve na tento mutex, v tomto prípade nevráti hodnotu `WAIT_OBJECT_0`, ale hodnotu `WAIT_ABANDONED`.

Použitá literatúra:

RICHTER, Jeffrey. *Programming Applications for Microsoft Windows*. 4. vyd. 1999. ISBN 1-57231-996-8

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