Správa pamäte

Architektúra pamäte používaná operačným systémom je najdôležitejším kľúčom k pochopeniu ako operačný systém pracuje. Windows ponúka 3 mechanizmy pre prácu s pamäťou:

- Virtuálna pamäť, vhodná najmä na správu veľkých polí objektov a štruktúr.
- Súbory namapované do pamäte (*memory-mapped files*), vhodné najmä na správu obsiahlych prúdov dát (zvyčajne zo súborov) a na zdieľanie dát medzi viacerými procesmi.
- Haldy, vhodné na správu veľkého množstva malých objektov.

Virtuálna pamäť

Každému procesu je pridelený vlastný virtuálny adresový priestor. Vlákno v rámci určitého procesu má prístup k pamäti patriacej tomuto procesu. Pamäť patriaca ostatným procesom je pred daným vláknom ukrytá a nedostupná.

Stránky v adresovom priestore procesu môžu byť buď *free*, *reserved* alebo *committed*. Po vytvorení procesu je väčšina jeho adresového priestoru voľná. Pre využitie tohto voľného priestoru je nutné si pamäť najskôr rezervovať a potom potvrdiť túto rezerváciu. Tieto akcie je možné vykonať aj naraz. Rezerváciu a potvrdenie umožní volanie funkcie `VirtualAlloc`:

```
LPVOID VirtualAlloc(
    LPVOID address,
    SIZE_T regionSize,
    DWORD allocationType,
    DWORD protection
    );
```

Vlákno si môže rezervovať pamäť pre budúce využitie. Pokus o pristúp k tejto rezervovanej pamäti však vyústi do chyby *access violation*, pretože táto pamäť nie je ešte zobrazená do fyzickej pamäte. Aby sa mohlo pristupovať k rezervovanej pamäti, je nutné ju najskôr potvrdiť. *Reserved* stránky tak zmenia označenie na *committed*. Tieto stránky sú v čase prístupu k nim preložené na platné stránky vo fyzickej pamäti. Po skončení práce s danou pamäťou je nutné volať funkciu `VirtualFree`, ktorá túto pamäť uvoľní:

```
BOOL VirtualFree(
    LPVOID address,
    SIZE_T regionSize,
    DWORD freeType
    );
```

Zásobník vlákna

Vždy keď je vytvorené nové vlákno, systém rezervuje oblasť v adresovom priestore procesu pre zásobník vlákna. Tejto rezervovanej oblasti je pridelená aj nejaká fyzická pamäť. Každé vlákno má svoj vlastný zásobník. Implicitne systém rezervuje 1 MB adresového priestoru a priradí mu 2 stránky fyzického priestoru. Toto nastavenie je, samozrejme, možné zmeniť, a to pri volaní funkcii `CreateThread`, prípadne

_beginthreadex. Každá z týchto funkcií má parameter, ktorý vie zmeniť veľkosť oblasti adresového priestoru vyhradenej pre zásobník vlákna. Ak je tento parameter 0, veľkosť oblasti bude implicitná. Pre potreby tohto textu bude implicitná veľkosť zásobníka nasledovná: 1 MB rezervovanej oblasti s postupným priraďovaním jednej stránky fyzickej pamäte.

Memory Address	State of Page
0x080FF000	Top of stack: committed page
0x080FE000	Committed page with guard protection attribute flag
0x080FD000	Reserved page
...	...
0x08002000	Reserved page
0x08001000	Reserved page
0x08000000	Bottom of stack: reserved page

Obr. 4-1: *Zásobník vlákna po vytvorení.*

Obr. 4-1 zobrazuje ako môže vyzerať oblasť zásobníku (rezervácia začína na adrese 0x08000000) na počítači s veľkosťou stránkok 4 kB.

Po zarezervovaní tejto oblasti, systém priradí fyzickú pamäť vrchným dvom stránkam oblasti. Skôr, než je vláknu dovolené pokračovať, systém nastaví ukazateľ zásobníku vlákna tak, aby ukazoval na koniec vrchnej stránky oblasti zásobníku (adresa veľmi blízko adresy 0x08100000). Na tejto stránke začne vlákno používať svoj zásobník. Druhá stránka v poradí je tzv. *guard page*. Popri tom ako vlákno rozširuje svoj strom volaní volaním ďalších funkcií, potrebuje stále viac miesta vo svojom zásobníku.

Kedykoľvek sa vlákno pokúsi pristúpiť k pamäti na stránke *guard page*, je na to upozornený systém. Systém odpovie pridelením ďalšej stránky pamäte hneď pod stránkou *guard page*. Potom systém odoberie príznak *guard page* aktuálnej *guard page* stránke a pridelí ho novej stránke. Táto technika umožňuje rozšíriť zásobník iba vtedy, keď to vlákno vyžaduje.

Zásobník vlákna sa postupom času dostane do stavu zobrazenom na **Obr. 4-2**. Predpokladajme, že v tomto okamihu ukazateľ zásobníku vlákna ukazuje na adresu 0x08003004. Ak teraz vlákno zavolá ďalšiu funkciu, systém mu musí prideliť viac fyzickej pamäti. Ak je však pridelená pamäť na adrese 0x08001000, systém sa nechová tak, ako pri prideľovaní pamäte vo zvyšku zásobníku. Tak ako očakávate, stránke na adrese 0x08002000 bude odobraný príznak *guard page* a stránke začínajúcej na adrese 0x08001000 bude pridelená fyzická adresa. Rozdiel je v tom, že systém tejto novej fyzickej stránke nepriradí *guard* atribút. To znamená, že rezervovaná oblasť adresového priestoru pre zásobník vlákna v tomto okamihu obsahuje všetku fyzickú pamäť, ktorá jej môže byť pridelená. Najspodnejšia stránka zásobníka je vždy rezervovaná. Plný zásobník je zobrazený na **Obr. 4-3**.

Memory Address	State of Page
0x080FF000	Top of stack: committed page
0x080FE000	Committed page
…	...
0x08003000	Committed page
0x08002000	Committed page with guard protection attribute flag
0x08001000	Reserved page
0x08000000	Bottom of stack: reserved page

Obr. 4-2: *Takmer plná oblasť zásobníku vlákna*

Pri priradení fyzickej pamäte stránke na adrese 0x08001000 systém vykoná ešte jednu akciu – vyhodí výnimku EXCEPTION_STACK_OVERFLOW. Z tejto výnimky sa správne ošetrený program dokáže zotaviť. Ak však vlákno bude používať zásobník aj po tejto výnimke a použije celú pamäť na stránke 0x08001000, pokúsi sa pristúpiť k pamäti na stránke začíajúcej na adrese 0x08000000. Keď sa vlákno pokúsi pristúpiť k tejto rezervovanej, ale nepridelenej pamäti, systém vyhodí výnimku *access violation*. V tomto okamihu preberá kontrolu systém a ukončí celý proces – nie iba vlákno. Systém dokonca užívateľa na túto skutočnosť ani neupozorní – celý proces jednoducho zmizne.

Memory Address	State of Page
0x080FF000	Top of stack: committed page
0x080FE000	Committed page
…	...
0x08003000	Committed page
0x08002000	Committed page
0x08001000	Committed page
0x08000000	Bottom of stack: reserved page

Obr. 4-3: *Plný zásobník vlákna*

Teraz je vhodné vysvetliť, prečo najspodnejšia stránka zásobníku vlákna musí byť vždy rezervovaná. Rezervovaním tejto stránky sú chránené iné dáta používané procesom pred náhodným prepísaním. Je

totižto možné, že na adrese 0x07FFF000 (jedna stránka pod dnom zásobníka) má iná oblasť adresového priestoru pridelenú fyzickú pamäť. Keby aj stránka na adrese 0x08000000 obsahovala fyzickú pamäť, systém by nezachytil pokus vlákna pristúpiť k rezervovanej oblasti zásobníka. Ak by šiel zásobník pod svoju rezervovanú oblasť, mohol by prepísať iné dáta v adresovom priestore procesu – veľmi tažko odhaliteľná chyba.

Súbory namapované do pamäte

Tak ako virtuálna pamäť, aj súbory namapované do pamäte umožňujú rezervovať oblasť adresového priestoru a prideliť jej fyzickú pamäť. Rozdiel je v tom, že fyzická pamäť v tomto prípade je súborom, ktorý už je na disku a nie stránkovacím súborom systému. Keď je daný súbor zobrazený, môžete k nemu pristupovať, akoby bol celý súbor načítaný do pamäti.

Súbory namapované do pamäte sa používajú v 3 rôznych prípadoch:

- Systém používa do pamäte namapované súbory na načítanie a vykonanie EXE a DLL súborov. Významne tým šetrí priestor stránkovacieho súboru a čas potrebný na vykonanie programu.
- Súbory namapované do pamäte môžete použiť aj na prístup k dátovému súboru na disku. Nemusíte sa potom zaoberať vstupno-výstupnými operáciami a načítavaním obsahu súboru.
- Tieto súbory je možné použiť aj k zdieľaniu dát medzi viacerými procesmi. Windows, samozrejme, ponúka aj iné spôsoby výmeny dát medzi procesmi, ale tieto metódy sú naimplementované práve s použitím súborov namapovaných do pamäte.

EXE a DLL súbory namapované do pamäte

Keď vlákno zavolá funkciu `CreateProcess`, systém vykoná nasledujúce kroky:

1. Systém lokalizuje .exe súbor špecifikovaný vo volaní `CreateProcess` funkcie. Ak súbor nenájde, proces sa nevytvorí a funkcia `CreateProcess` vráti hodnotu `FALSE`.
2. Systém vytvorí nový objekt jadra procesu.
3. Systém vytvorí vlastný adresový priestor pre tento nový proces.
4. Systém rezervuje dostatočne veľkú oblasť adresového priestoru tak, aby obsiahla .exe súbor.
5. Systém zaznamená, že fyzická pamäť pre rezervovanú oblasť je v danom .exe súbore namiesto stránkového súboru systému.

Po namapovaní .exe súboru do adresového priestoru procesu, systém pristúpi k tej časti .exe súboru, ktorá obsahuje zoznam funkcií z DLL súborov, ktoré volá kód .exe súboru. Systém potom volá funkciu `LoadLibrary` pre každé z týchto DLL súborov. Pri každom jednom volaní funkcie `LoadLibrary` sú vykonané kroky podobné bodom 4 a 5. Tentokrát sa však miesto .exe súboru pracuje s DLL súborom.

Ak z nejakého dôvodu nie je možné namapovať .exe súbor a všetky požadované DLL súbory, systém užívateľovi zobrazí správu a uvoľní adresový priestor procesu a objekt jadra procesu. Funkcia `CreateProcess` vráti hodnotu `FALSE`.

Po namapovaní .exe a všetkých DLL súborov do adresového priestoru procesu, systém môže začať vykonávať kód .exe súboru. Po namapovaní .exe súboru sa prípadné stránkovanie, *buffering* a *caching* stará systém.

Dátové súbory namapované do pamäte

Operačný systém umožňuje namapovať do adresového priestoru vášho procesu aj dátové súbory, čím sa práca s veľkým množstvom dát stáva omnoho pohodlnejšou. Aby bolo možné namapovaný súbor používať, je nutné urobiť nasledujúce kroky:

1. Vytvorte alebo otvorte objekt jadra súboru, ktorý identifikuje súbor na disku, ktorý chcete namapovať – zavolajte funkciu *CreateFile*.
2. Vytvorte objekt namapovaného súboru, ktorý oznámi systému veľkosť súboru a spôsob, akým chcete k súboru pristupovať – zavolajte funkciu *CreateFileMapping*.
3. Nechajte systém, aby do adresového priestoru vášho procesu namapoval všetky časti objektu namapovaného súboru – zavolajte funkciu *MapViewOfFile*.

Po skončení používania pamäťového namapovaného súboru, je nutné urobiť nasledujúce kroky:

1. Nechajte systém odstrániť namapovanie objektu namapovaného súboru z adresového priestoru vášho procesu – zavolajte funkciu *UnmapViewOfFile*.
2. Zatvorte objekt jadra namapovaného súboru - *CloseHandle*.
3. Zatvorte objekt jadra súboru - *CloseHandle*.

Celý popísaný postup má štruktúru podobnú nasledujúcemu pseudokódu:

```
HANDLE hFile = CreateFile(...);
HANDLE hFileMapping = CreateFileMapping(hFile, ...);
PVOID fileView = MapViewOfFile(hFileMapping, ...);

// use the memory-mapped file

UnmapViewOfFile(fileView);
CloseHandle(hFileMapping);
CloseHandle(hFile);
```

Daný pseudokód zobrazuje predpokladaný spôsob práce s namapovanými súbormi. Neukazuje však, že systém po zavolaní funkcie *MapViewOfFile* zvýši *usage count* objektu súboru a objektu namapovaného súboru. Tento vedľajší efekt znamená, že daný kód možeme prepísať nasledovne:

```
HANDLE hFile = CreateFile(...);
HANDLE hFileMapping = CreateFileMapping(hFile, ...);
CloseHandle(hFile);
PVOID fileView = MapViewOfFile(hFileMapping, ...);
CloseHandle(hFileMapping);

// use the memory-mapped file

UnmapViewOfFile(fileView);
CloseHandle(hFileMapping);
```

Pokiaľ však chcete vytvoriť dodatočný objekt namapovaného súboru, nesmiete zavolať funkciu *CloseHandle* skôr – *handle* objektu súboru budete potrebovať pri ďalšom volaní funkcií *CreateFile-Mapping* a *MapViewOfFile*.

Namapované súbory a koherencia

Systém umožňuje namapovať niekoľko pohľadov tých istých dát zo súboru. Napríklad môžete namapovať prvých 10KB súboru do jedného pohľadu a potom ďalšie 4KB toho istého súboru do iného pohľadu. Kým mapujete ten istý objekt namapovaného súboru, systém zaručí, že tieto dáta sú koherentné. Ak napríklad aplikácia zmení dáta v jednom pohľade, všetky ďalšie pohľady sú aktualizované tak, aby odzrkadľovali dané zmeny. To je možné z dôvodu, že aj keď je niektorá stránka namapovaná do virtuálneho adresového priestoru procesu viackrát, systém má dané dáta v skutočnosti v RAM len jedenkrát. Ak majú viaceré procesy namapované pohľady toho istého súboru, dáta sú stále koherentné, pretože v dátovom súbore je stále len jedna inštancia každej stránky RAM – jedná sa len o namapovanie stránok RAM do rôznych adresových priestorov procesov.

Pri práci so súbormi však neexistuje dôvod, pre ktorý by iná aplikácia nemohla zavolať funkciu `CreateFile`, aby otvorila súbor, ktorý už má namapovaný iný proces. Tento nový proces potom môže pomocou funkcií `ReadFile` a `WriteFile` čítať a zapisovať do daného súboru. Samozrejme, kedykoľvek proces volá tieto funkcie, musí dáta čítať alebo zapisovať do pamäťového bufferu. Tento buffer musí byť vytvorený týmto procesom, nesmie to byť pamäť používaná namapovaným súborom. Problém nastane ak majú dve aplikácie otvorený ten istý súbor: jeden proces môže zavolať funkciu `ReadFile` na prečítanie časti súboru, zmeniť tieto dáta a zapísať ich naspäť s použitím funkcie `WriteFile` bez toho, aby objekt namapovaného súboru druhého vlákna bol upozornený na túto skutočnosť. Z tohto dôvodu sa odporúča pri volaní funkcie `CreateFile` pre súbory, ktoré budú namapované, špecifikovať parameter `dwShareMode` hodnotou 0. Tým oznámite systému, že si želáte výlučný prístup k tomuto súboru a žiadny iný proces ho nesmie otvoriť.

Read-only súbory sú vhodnými kandidátmi na namapované súbory, nakoľko nemajú problém s koherenciou. Pamäťové namapované súbory by nikdy nemali byť použité na zdieľanie zapisovateľných súborov cez sieť, pretože systém nemôže garantovať koherentné pohľady dát. Ak niektorý počítač zaktualizuje obsah súboru, iný počítač s pôvodnými dátami v pamäti o zmene nebude vedieť.

Zdieľanie dát medzi procesmi s využitím súborov namapovaných do pamäte

Windows vyniká v ponuke mechanizmov, ktoré umožňujú aplikáciám rýchlo a ľahko zdieľať dáta a informácie. Tieto mechanizmy zahŕňajú RPC, COM, OLE, DDE, *window messages, clipboard, mailslots, pipes, sockets*,... Vo Windows je mechanizmom pre zdieľanie dát na jednom počítači na najnižšej úrovni práve mapovanie súborov do pamäte. Aj všetky vyššie spomenuté mechanizmy, pokiaľ sa jedná o procesy komunikujúce na tom istom počítači, na najnižšej úrovni používajú práve mapovanie súborov.

Takýto spôsob zdieľania dát je dosiahnutý tým, že dva alebo viac procesov namapujú pohľady toho istého objektu namapovaného súboru, čo znamená, že zdieľajú rovnaké stránky fyzickej pamäte. Vo výsledku, keď jeden proces zapíše dáta do pohľadu zdieľaného objektu mapovaného súboru, ostatné procesy okamžite vidia zmenu v ich pohľadoch. Ak viaceré procesy zdieľajú ten istý objekt namapovaného súboru, všetky musia používať pre tento objekt to isté meno.

Tak ako pre všetky objekty jadra, aj pre objekty mapovaných súborov môžete použiť 3 techniky zdieľania týchto objektov medzi viacerými procesmi: dedičnosť *handles*, pomenovanie objektov a duplikáciu *handles*.

Haldy

Posledným mechanizmom pre prácu s pamäťou je použitie háld. Haldy sú vhodné najmä pre správu mnohých menších blokov dát. Využijú sa najmä pri práci so spojenými zoznamami alebo stromami.

Halda je oblasť rezervovaného adresového priestoru. Iniciálne, väčšine stránok rezervovanej oblasti nie je pridelená fyzická pamäť. Čím viac alokovania z pamäte haldy, tým viac pridelených fyzických stránok. Táto fyzická pamäť je vždy pridelená zo stránkovacieho súboru systému. Pri uvoľnení blokov vo vnútri haldy, je uvoľnená pridelená fyzickej pamäte.

Implicitná halda procesu

Pri inicializácií procesu systém v adresovom priestore procesu vytvorí haldu. Táto halda sa nazýva implicitná halda procesu (*default heap*). Implicitne má táto halda veľkost 1MB a využívajú ju mnohé Windows funkcie. Ak napríklad niektorá Windows funkcia potrebuje použiť dočasný blok pamäte, alokuje tento blok práve z implicitnej haldy procesu.

Nakoľko implicitná halda procesu je používaná mnohými Windows funkciami a vaša aplikácia môže mať niekoľko vlákien, ktoré všetky môžu volať rôzne Windows funkcie, prístup k implicitnej halde procesu je sériový. Inými slovami, systém garantuje, že v danom čase smie alokovať alebo uvoľňovať blok pamäte z implicitnej haldy práve jedno vlákno. Ak sa v tom istom čase pokúsia o alokovanie dve rôzne vlákna, jedno z nich bude donútené čakať. Tento sériový prístup však znamená pomalší výkon. Ak je vo vašej aplikácií len jedno vlákno, urýchlite prístup k halde vytvorením vlastnej haldy, ktorá nemusí mať sériový prístup. Windows funkcie však aj naďalej budú pracovať s implicitnou haldou procesu.

Jeden proces môže mať aj niekoľko háld. Tieto haldy môžu byť vytvárané, alebo zničené v priebehu životného cyklu procesu. Implicitná halda je však vytvorená skôr, než sa proces začne vykonávať a je automaticky zničená, keď proces skončí. Implicitnú haldu nemôžete zničiť.

Každá halda je identifikovateľná pomocou svojej *handle*. Všetky funkcie alokujúce alebo uvoľňujúce bloky pamäte vo vnútri haldy potrebujú túto *handle* ako parameter. *Handle* k implicitnej halde procesu je možné získať volaním funkcie `GetProcessHeap`:

```
HANDLE GetProcessHeap();
```

Funkcie pre prácu s haldami

Pre prácu s haldami existuje množstvo Windows funkcií. Dodatočnú haldu pre svoj proces vytvoríte volaním funkcie `HeapCreate`:

```
HANDLE HeapCreate(
    DWORD options,
    SIZE_T initialSize,
    SIZE_T maximumSize
    );
```

Prvý parameter funkcie môže nadobudnúť hodnoty 0, `HEAP_NO_SERIALIZE`, `HEAP_GENERATE_EXCEPTIONS` alebo kombináciu týchto hodnôt. Hodnota `HEAP_NO_SERIALIZE` zmanená, že prístup k halde nebude sériový. S použitím tejto hodnoty však treba narábať opatrne, aby potom pri prístupe viacerých vlákien k halde nedošlo k poškodeniu haldy. Hodnota `HEAP_GENERATE_EXCEPTIONS` spôsobí, že systém vyhodí výnimku pri každom zlyhaní pokusu o alokovanie, prípadne realokovanie bloku pamäti z haldy. O použití štruktúrovaných výnimiek však až neskôr v priebehu semestra...

Alokovanie bloku pamäte z haldy je zabezpečené volaním funkcie `HeapAlloc`:

```
PVOID HeapAlloc(
    HANDLE hHeap,
    DWORD flags,
    SIZE_T bytes
    );
```

Pri volaní tejto funkcie s hodnotou parametra flags `HEAP_ZERO_MEMORY`, alokovaný blok pamäte bude naplnený nulami.

Pri pokuse o alokovanie bloku pamäte z haldy, funkcia `HeapAlloc` vykonáva nasledujúce kroky:

1. Prejde spojový zoznam alokovaných a voľných pamäťových blokov.
2. Nájde adresu voľného bloku.
3. Alokuje nový blok preznačením voľného bloku na blok alokovaný.
4. Pridá nový vstup do spojového zoznamu pamäťových blokov.

Veľkosť alokovaného bloku pamäte je, samozrejme, možné aj zmeniť, a to volaním funkcie *`HeapReAlloc`*:

```
PVOID HeapReAlloc(
    HANDLE hHeap,
    DWORD flags,
    PVOID mem,
    SIZE_T bytes
    );
```

Po alokovaní bloku pamäte, môžete volať funkciu `HeapSize` na získanie aktuálnej veľkosti bloku:

```
SIZE_T HeapSize(
    HANDLE hHeap,
    DWORD flags,
    LPCVOID mem
    );
```

Ak už pamäťový blok nepotrebujete, môžete ho uvoľniť volaním funkcie `HeapFree`:

```
BOOL HeapFree(
    HANDLE hHeap,
    DWORD flags,
    PVOID mem
    );
```

Ak vaša aplikácia už ďalej nepotrebuje haldu, ktorú vytvorila, môže ju zničiť volaním funkcie `HeapDestroy`:

```
BOOL HeapDestroy(HANDLE hHeap);
```

Volanie funkcie *`HeapDestroy`* uvoľní všetky pamäťové bloky obsiahnuté v halde a tiež spôsobí, že fyzická pamäť a rezervovaný adresový priestor využívaný haldou bude uvoľnený späť systému. Ak je funkcia úspešná, vráti hodnotu TRUE. Ak explicitne nezničíte haldu pred ukončením procesu, systém ju zničí za vás. Halda je však zničená až keď proces končí, t.j. ak nejaké vlákno vytvorí haldu a skončí, halda sa nezničí skončením tohto vlákna.

Systém neumožní zničiť implicitnú haldu procesu, až kým proces úplne neskončí. Ak zavoláte funkciu *`HeapDestroy`* s *handle* implicitnej haldy, systém bude toto volanie jednoducho ignorovať.

Použitá literatúra:

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