Gestiunea memoriei
Heap-uri și
Fișiere mapate în memorie

CURS NR. 8

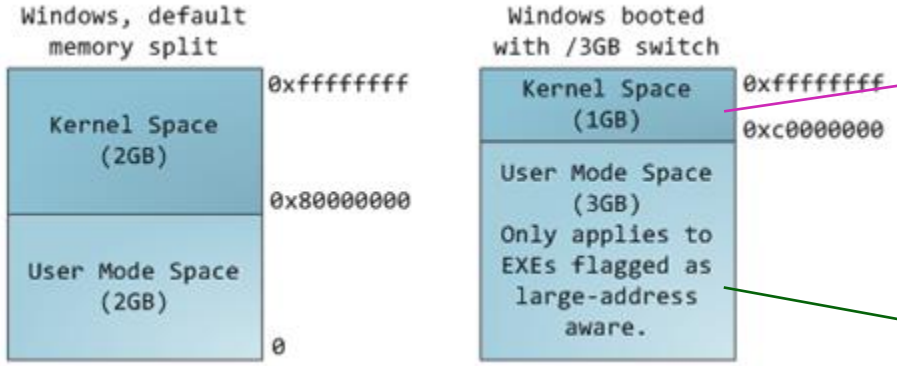
Gestiunea memoriei

Aplicațiile au nevoie de gestiunea dinamică a memoriei

- Când lucrăm cu structuri de date a căror număr sau dimensiune nu este cunoscută înainte de rulare

Fiecare proces are **propriul** spațiu de adrese virtual

- Win32: adrese pe 32 de biți => spațiu de adrese de 2^{32} de octeți (4 GB)
  - Cel puțin jumătate (2 GB sau 3 GB) este în spațiul utilizator – restul (2 GB sau 1 GB) este spațiul kernel



Spațiu kernel

kernel, hal, drivere
stiva, heap din sp. kernel
tabele de pagini, cache de fișiere
structuri de date și obiecte kernel

Spațiu utilizator

segment de cod și date din executabile
stiva și heap-ul din sp. utilizator
blocuri de mediu/control al proceselor și threadurilor (PEB, TEB)

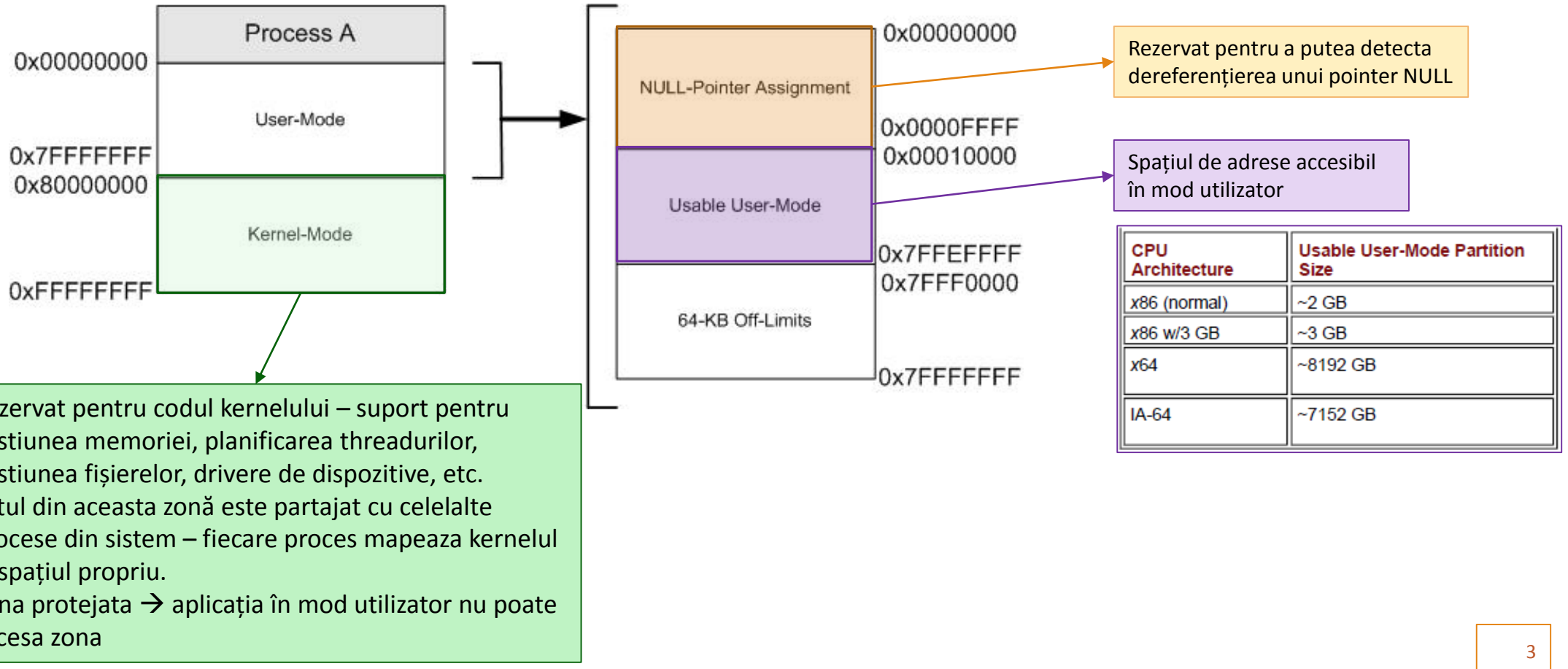
- Win64: adrese pe 64 de biți => spațiu de adrese de 2^{64} de octeți (16 EB - exabytes)

Atenție: este vorba de dimensiunea spațiului de adrese virtual – nu spațiul fizic alocat!!!

Arhitectura gestiunii memoriei

Partiționarea spațiului de adrese

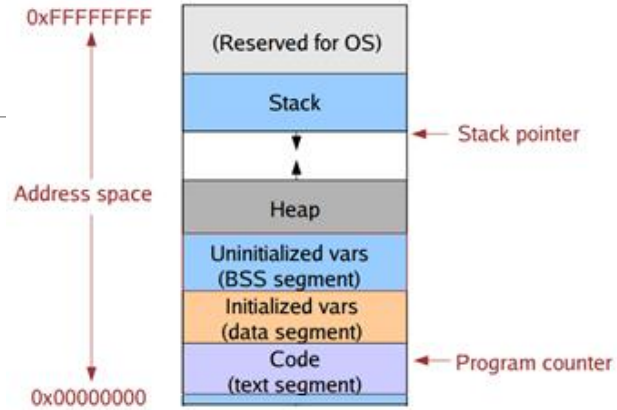
- Partiționarea este dependentă de sistemul de operare – diferite kerneluri Windows partiționează diferit spațiul de adrese



- Rezervat pentru codul kernelului – suport pentru gestiunea memoriei, planificarea threadurilor, gestiunea fișierelor, drivere de dispozitive, etc.
- Totul din aceasta zonă este partajat cu celelalte procese din sistem – fiecare proces mapeaza kernelul în spațiul propriu.
- Zona protejata → aplicația în mod utilizator nu poate accesa zona

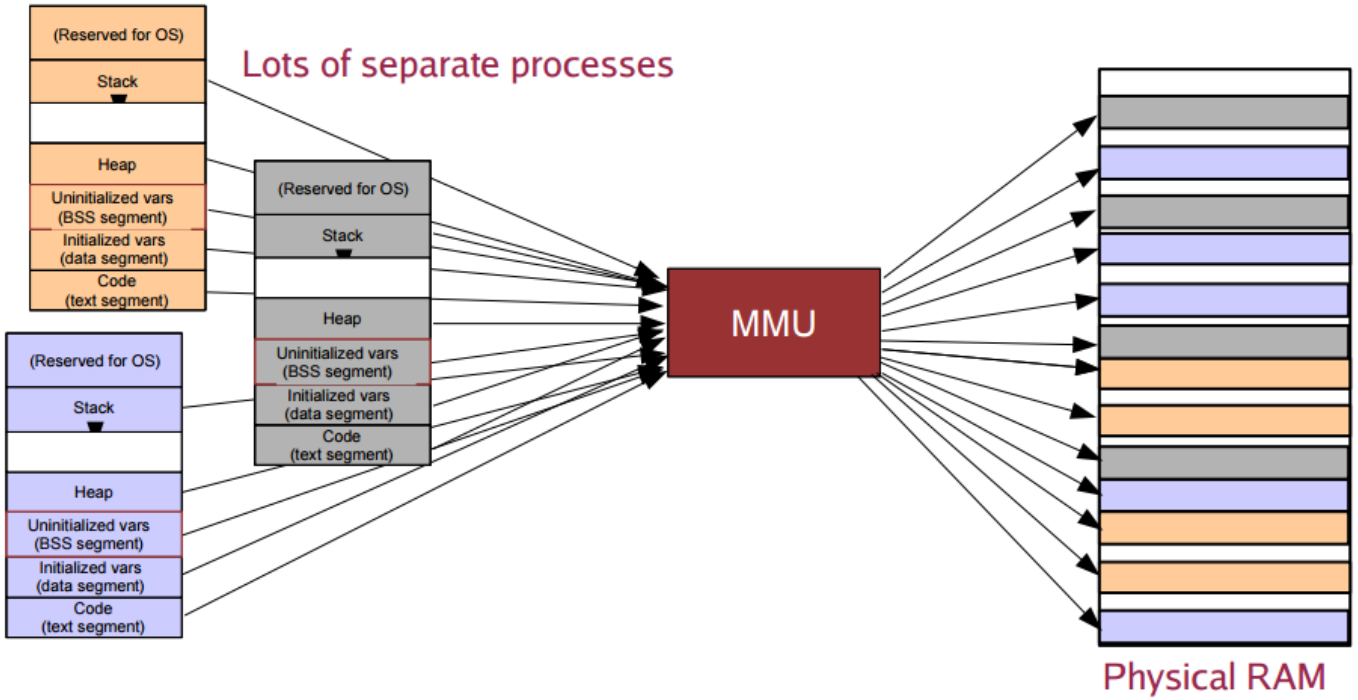
Arhitectura gestiunii memoriei

Partiționarea spațiului de adrese



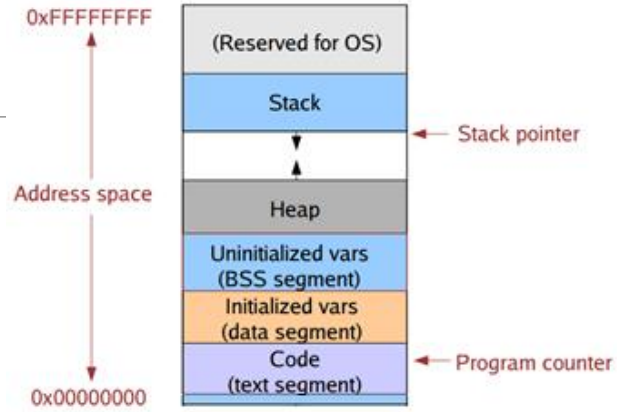
Maparea adreselor

- Spațiul de adrese logic nu este neaparat în zona contiguă în memoria fizică
- MMU (Memory Management Unit) este responsabilă de maparea adreselor logice la adrese fizice



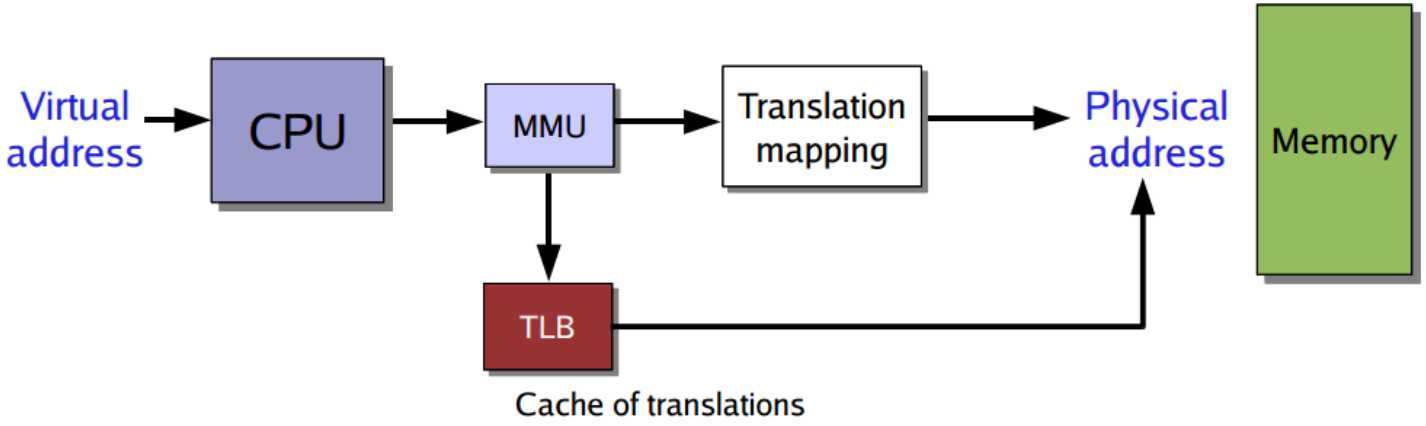
Arhitectura gestiunii memoriei

Partiționarea spațiului de adrese



Maparea adreselor

- Spațiul de adrese logic nu este neaparat în zona contiguă în memoria fizică
- MMU (Memory Management Unit) este responsabilă de maparea adreselor logice la adrese fizice



Gestiunea memoriei – paginare cu memorie virtuală

Paginare – spațiul de adrese se împarte în pagini, memoria fizică se împarte în cadre de pagină

- Dimensiuni egale – 4KB, 8KB
- Gestiunea paginilor – prin tabela de pagini (o tabela per proces)
  - Conține mapare număr pagină – număr cadru de pagină

Memoria virtuală – memorie pe disc

- Programul nu mai este constrâns de limitele memoriei fizice - se pot rula programe de dimensiuni mai mari decât memoria fizică
- Termenul folosit în Windows: `pagefile` (sau paging file)
  - În mod transparent este gestionat ca o extensie a memoriei interne

TLB (Translation look-aside buffer)

- Memoria asociativă de viteză mare
- Cache al tabelei de pagini (reține maări accesate recent)

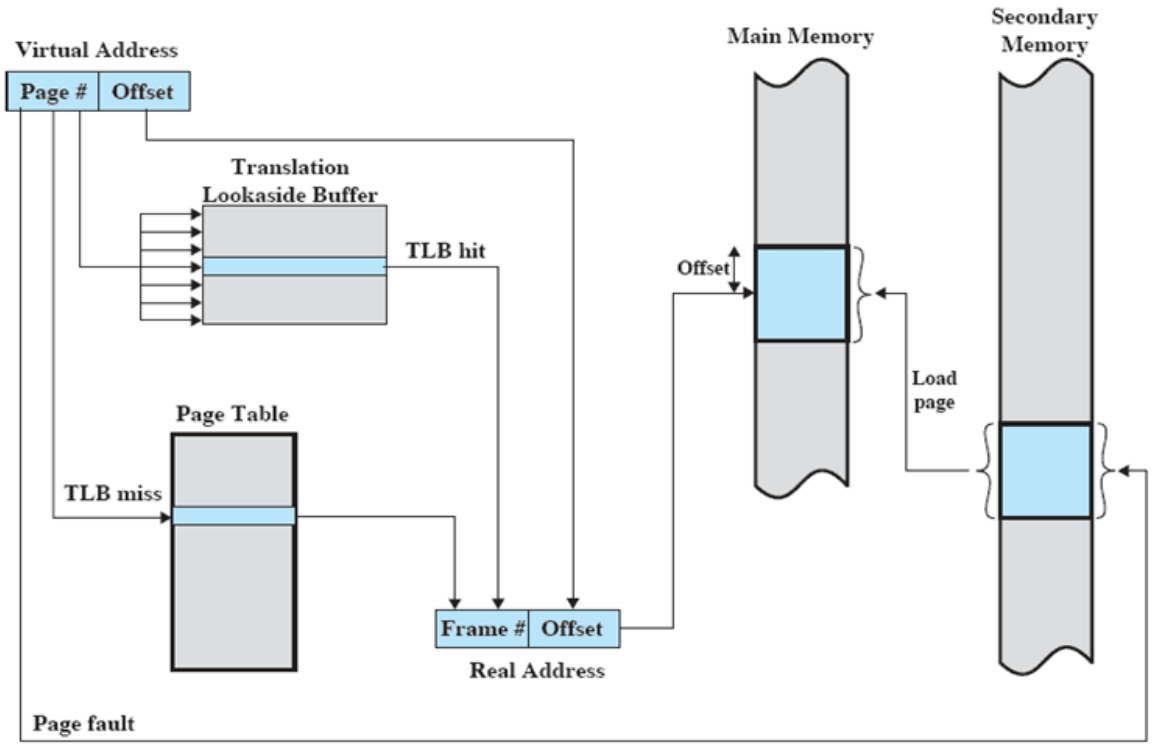


Figure 8.7 Use of a Translation Lookaside Buffer

Stallings, pg. 351

TLB și memoria CACHE

Conceptul de cache

- Zonă de stocare temporară, unde informația accesată frecvent poate fi stocată pentru acces rapid
  - Data din cache este o copie a datei originale - acesta ar fi mult mai costisitor de accesat

TLB este un cache pentru translatare mai rapidă

- Eficientizează mecanimului de mapare din spațiul de adrese al procesului în spațiul de adrese al memoriei fizice
- Stochează mapări recente (similar cu intrările în tabela de pagini)
  - adresă virtuală : adresă fizică (plus biți de control)
- Căutare eficientă – Content-Addressable Memory (CAM)

Memoria CACHE propriu-zisă: pentru acces rapid la date

- Eficientizează accesul la datele / instrucțiunile din memoria fizică accesate recent
- Viteză mare de acces

TLB și memoria CACHE

Stallings, pg. 353

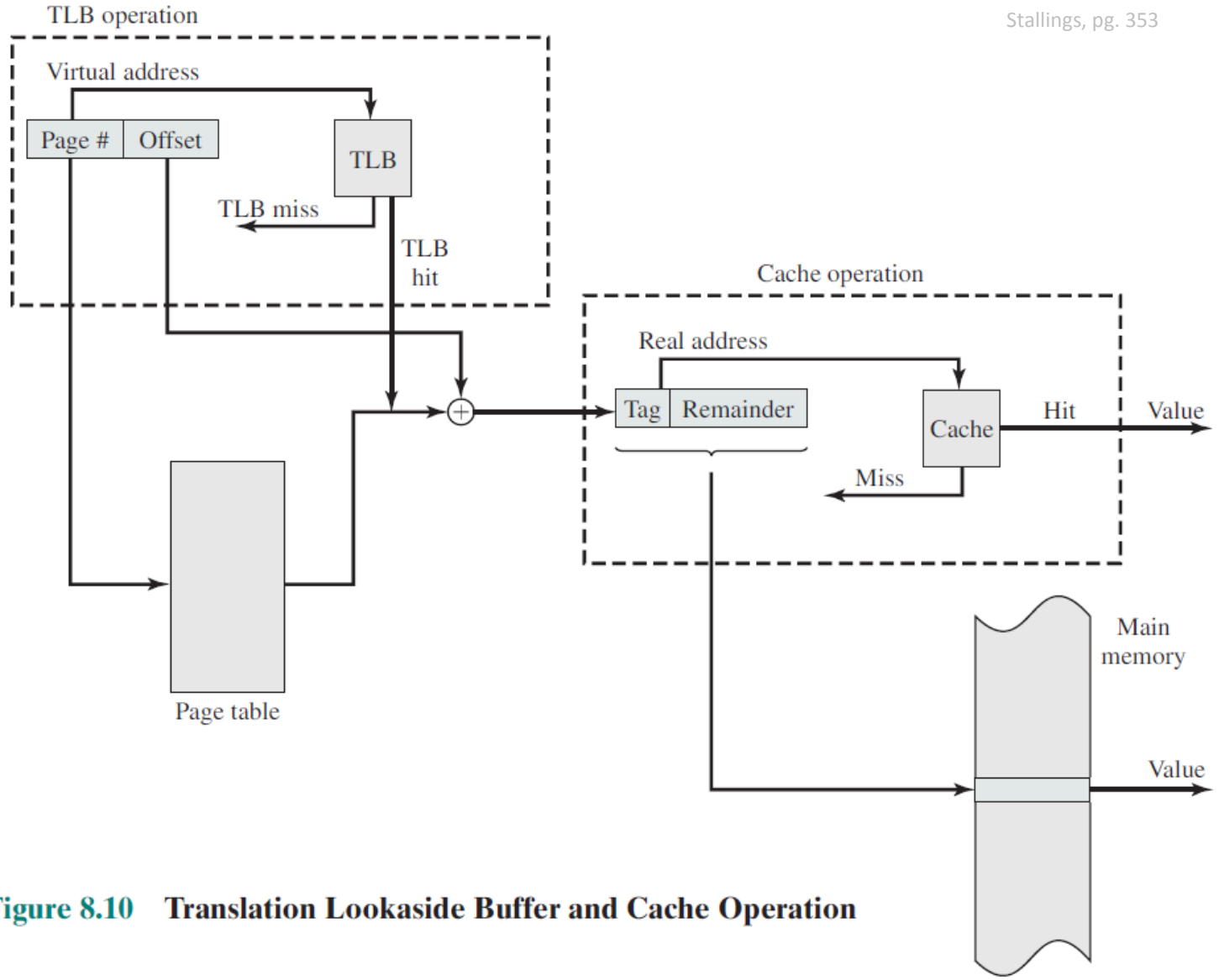


Figure 8.10 **Translation Lookaside Buffer and Cache Operation**

Concepte și funcții

Pagină (din spațiul virtual) și cadru de pagină (din memoria fizică)

Rezervare – alocarea unei zone de memorie din spațiul accesibil user-mode ← `VirtualAlloc`

- Granularitate de alocare – depinde de platformă – de regulă 64 KB
  - Zona rezervată va începe de la o adresă multiplu de 64 KB
- Dimensiunea zonei rezervate – trebuie să fie multiplu de dimensiune de pagină – depinde de platformă - 4 KB (sau 8 KB)
- Sistemul de operare nu este constrâns de respectarea limitei de granularitate când rezervă spațiu în cadrul procesului
  - De ex pt. PEB (Process Environment Block) sau TEB (Thread Environment Block), etc.

  dar trebuie să respecte dimnesiunea minimă rezervată – 1 pagină

- Ex. Dacă dorim rezervarea unei zone de 10 KB → sistemul rotunjește în sus cererea spre un multiplu de dimensiune de pagină
  - 12 KB (pe sisteme x86 și x64)
  - 16 KB (pe sisteme IA-64)

Alocarea spațiului fizic (committing) ← `VirtualAlloc`

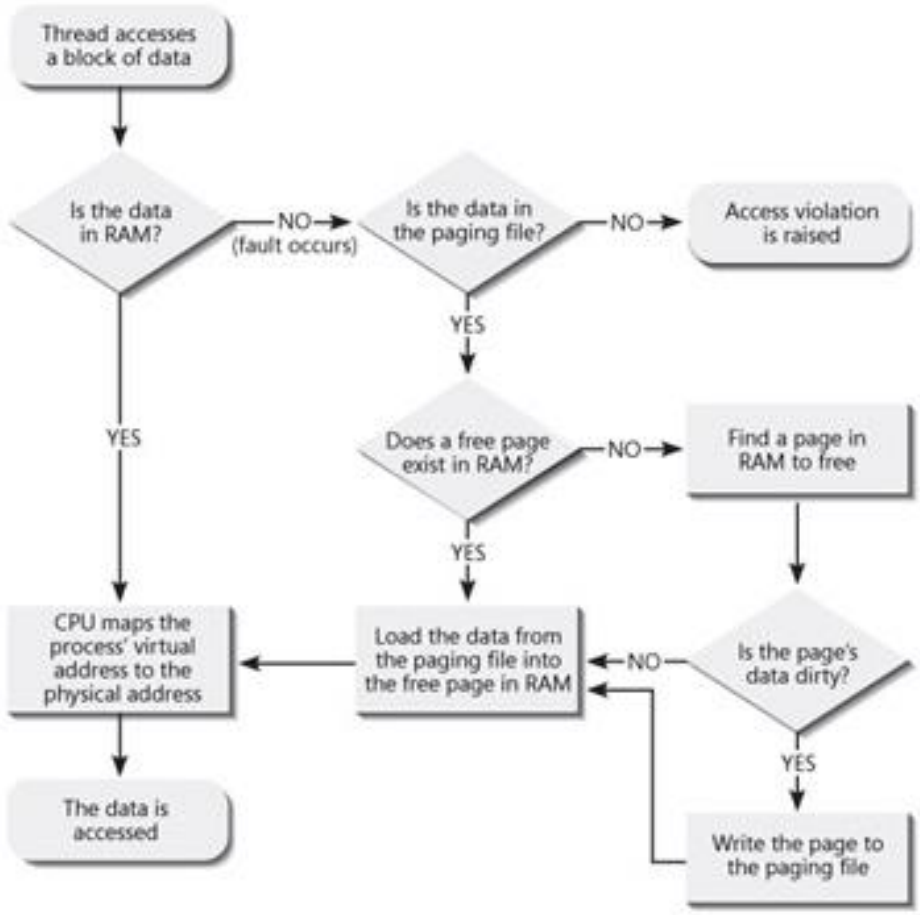
- Pentru a utiliza un spațiu rezervat, trebuie alocat spațiu fizic și apoi mapat la zona rezervată
- alocarea se face în pagini întregi
- Din memoria internă sau din paging file (memoria virtuală)

Eliberarea (release)- eliberarea zonei rezervate sau alocate ← `VirtualFree`

Accesul la un bloc de date

Acces folosind memorie virtuală - paging file

Windows via C/C++, pg. 380



- Paging file nu este singurul suport secundar folosit ca memorie virtuală
- Fișiere mapate în memorie
  - Fișiere de date sau fișiere executabile sunt folosite ca suport fizic de stocare asociat spațiului de adrese al procesului

Exemplu: lansarea în execuție a unei aplicații

- Sistemul determină dimensiunea segmentului de cod și date
- Sistemul rezervă memorie din spațiul de adrese
- Spațiul fizic asociat zonei rezervate este fișierului .exe

- De ex. Segmentul de cod nu ocupă spațiu din paging file, ci mapează în spațiul de adrese segmentul din imaginea (conținutul) fișierului executabil
  - Este partajat de toate instanțele aceleiași aplicații

- Permite ca aplicațiile să fie încărcate foarte rapid și menține paging file-ul la dimensiuni relativ mici

Arhitectura gestiunii memoriei

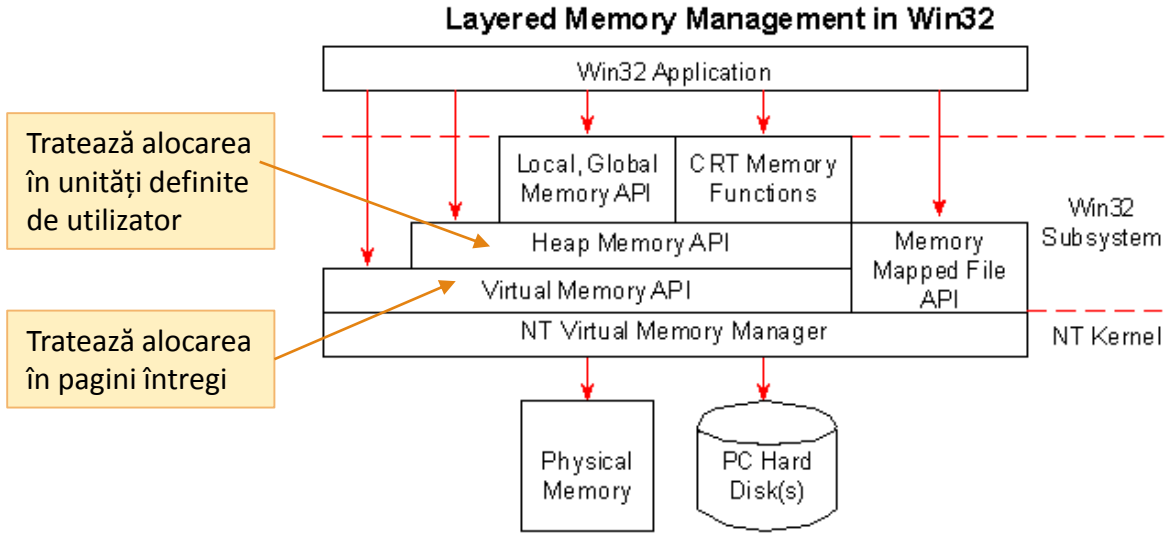
Gestiunea memoriei este sarcina SO

- Tehnici complexe (paginare, translatarea adreselor, algoritmi de înlocuire a paginilor, etc.)
- Amintim doar câteva aspecte
  - Sistemul are un volum relativ mic de memorie internă fizică
  - Fiecare proces are un spațiu de adrese propriu – care poate depăși dimensiunea memoriei fizice
    - Multiple procese coexistă în sistem simultan
  - SO mapează adresele virtuale la adrese fizice
    - Cele mai multe pagini virtuale nu sunt încărcate în memoria fizică → apar page-fault-uri

Trei grupuri de funții pentru gestiunea memoriei

- Funcții pentru heap-uri
- Funcții pentru fișiere mapate în memorie
- Funcții pentru memoria virtuală

API-uri de gestiune a memoriei



Informații despre sistem

Pentru explorarea structurii de memorie putem apela la funcția `GetSystemInfo`

- Oferă informații despre sistem: dimensiunea de pagină, granularitatea de alocare, adresa fizică a aplicației
  numărul de procesoare,

```
void WINAPI GetSystemInfo(
           LPSYSTEM_INFO lpSystemInfo );
```

Pointer la structura SYSTEM_INFO în care se returnează informațiile

```
typedef struct _SYSTEM_INFO {
   union {
      DWORD  dwOemId;
      struct {
         WORD wProcessorArchitecture;
         WORD wReserved;
      };
   };
   DWORD      dwPageSize;
   LPVOID     lpMinimumApplicationAddress;
   LPVOID     lpMaximumApplicationAddress;
   DWORD_PTR dwActiveProcessorMask;
   DWORD      dwNumberOfProcessors;
   DWORD      dwProcessorType;
   DWORD      dwAllocationGranularity;
   WORD       wProcessorLevel;
   WORD       wProcessorRevision;
} SYSTEM_INFO;
```

```
#include <windows.h>
#include <stdio.h>

void main() {
   SYSTEM_INFO siSysInfo;

   // Copy the hardware information to the SYSTEM_INFO structure.

   GetSystemInfo(&siSysInfo);

   // Display the contents of the SYSTEM_INFO structure.

   printf("Hardware information: \n");
   printf("  OEM ID: %u\n", siSysInfo.dwOemId);
   printf("  Number of processors: %u\n", siSysInfo.dwNumberOfProcessors);
   printf("  Page size: %u\n", siSysInfo.dwPageSize);
   printf("  Processor type: %u\n", siSysInfo.dwProcessorType);
   printf("  Minimum application address: %lx\n", siSysInfo.lpMinimumApplicationAddress);
   printf("  Maximum application address: %lx\n", siSysInfo.lpMaximumApplicationAddress);
   printf("  Active processor mask: %u\n", siSysInfo.dwActiveProcessorMask);
}
```

```
Hardware information:
  OEM ID: 0
  Number of processors: 4
  Page size: 4096
  Processor type: 586
  Minimum application address: 10000
  Maximum application address: 7ffeffff
  Active processor mask: 15
Press any key to continue . . .
```

Utilizarea memoriei virtuale în aplicații

Windows furnizează trei mecanisme pentru gestiunea memoriei din aplicațiile utilizator

- **Memoria virtuală**
  - Potrivită pentru gestiunea tablourilor de obiecte sau structuri de dimensiuni mari
- **Fișiere mapate în memorie**
  - Potrivite pentru gestiunea fluxurilor mari de date (de regulă din fișiere) și pentru partajarea datelor între procese distincte care rulează pe același sistem
- **Heap-uri**
  - Potrivite pentru gestiunea unui număr mare de obiecte mici

Memory set	System resource affected
Virtual memory functions	A process's virtual address space System pagefile System memory Hard disk space
Memory-mapped file functions	A process's virtual address space System pagefile Standard file I/O System memory Hard disk space
Heap memory functions	A process's virtual address space System memory Process heap resource structure
Global heap memory functions	A process's heap resource structure
Local heap memory functions	A process's heap resource structure
C run-time reference library	A process's heap resource structure

Memoria virtuală

Rezervarea SAU/ ȘI alocarea unei regiuni în spațiul de adrese

```
LPVOID WINAPI VirtualAlloc(
          _In_opt_ LPVOID lpAddress,
          _In_     SIZE_T dwSize,
          _In_     DWORD  flAllocationType,
          _In_     DWORD  flProtect );
```

Adresa de început unde se rezervă regiunea. NULL dacă nu avem preferințe

Dimnesiunea regiunii în octeți

MEM_COMMIT (alocarea spațiului de memorie – alocare efectivă doar la accesare; conținut resetat la 0)
MEM_RESERVE (rezervarea spațiului de memorie – fără alocare efectivă a spațiului fizic)
Altele: **MEM_RESET** și **MEM_RESET_UNDO**
Poate fi combinat cu **MEM_TOP_DOWN** și altele ...

rezervare + alocare:
MEM_RESERVE | MEM_COMMIT

Modul de protecție a paginilor care vor fi alocate

PAGE_READONLY , **PAGE_READWRITE** , **PAGE_EXECUTE** , **PAGE_EXECUTE_READWRITE** , și altele

- Zona rezervată va începe de la o adresă multiplu de granularitate de alocare (64 KB)
  - Ex. Dacă dorim să rezervăm spațiu la adresa 19,668,992 (300 × 65,536 + 8192) → sistemul rotunjește în jos la cel mai apropiat multiplu de 64KB, și de acolo va rezerva spațiu: 19,660,800 (300 × 65,536)
- Dimensiunea zonei rezervate – trebuie să fie multiplu de dimensiune de pagină – depinde de platformă - 4 KB (sau 8 KB)
  - Ex. Dacă dorim să rezervăm 62 KB → se rezervă 64 KB pe sisteme care folosesc pagini de 4 KB, 8 KB sau 16 KB
- Returnează adresa de început a regiunii rezervate
  - Dacă am specificat o adresă de început, și s-a găsit spațiu liber, va returna aceea adresă – rotunjită în jos la multiplu de 64KB

Memoria virtuală

- Exemplu VirtualAlloc.
  - După ce am rezervat un spațiu de 512 KB începând de la adresa 5,242,880, alocarea efectivă (commit) a unui spațiu de 6 KB începând la o distanță de 2 KB de la începutul zonei rezervate se face astfel:

```
VirtualAlloc((PVOID) (5242880 + (2 * 1024)), 6 * 1024, MEM_COMMIT, PAGE_READWRITE);
```

8 KB se alocă – multiplu de pagină

Eliberarea / dealocarea unei regiuni din spațiul de adrese

```
BOOL WINAPI VirtualFree(
          _In_ LPVOID lpAddress,
          _In_ SIZE_T dwSize,
          _In_ DWORD  dwFreeType );
```

Adresa de început a zonei de pagini care va fi eliberată
Dacă dwFreeType este **MEM_RELEASE** → se specifică adresa de început a regiunii rezervate

Dimnesiunea regiunii în octeți
Dacă dwFreeType este **MEM_RELEASE** → se specifică 0 (se eliberează toată regiunea rezervată)
Dacă dwFreeType este **MEM_DECOMMIT**

- se dealocă toate paginile care conțin cel puțin un octet din intervalul de la lpAddress până la lpAddress+dwSize
- Și se specifică 0 (se dealocă toată regiunea alocată)

Tipul operației: **MEM_RELEASE** (eliberarea zonei rezervate) **MEM_DECOMMIT** dealocarea zonei alocate)

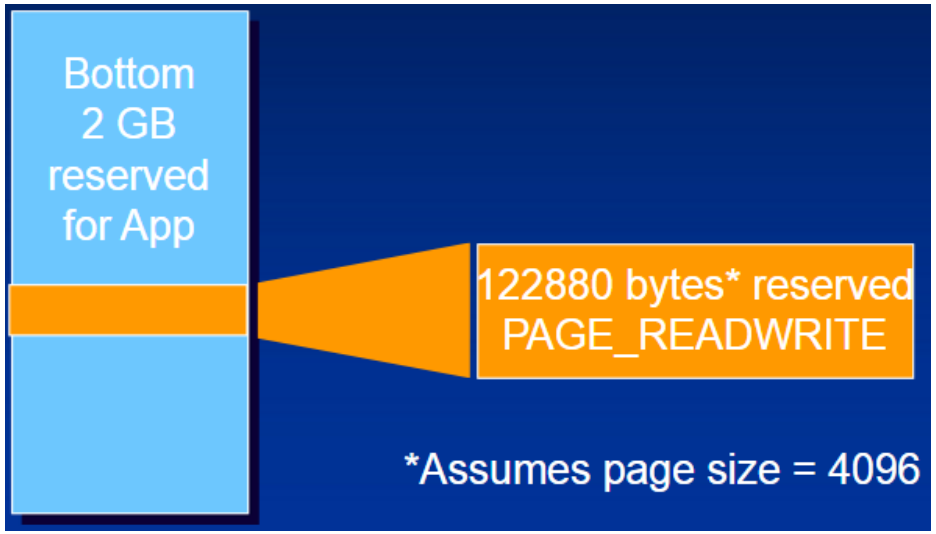
- Returnează o valoare nenulă în caz de succes, 0 în caz de eșec.

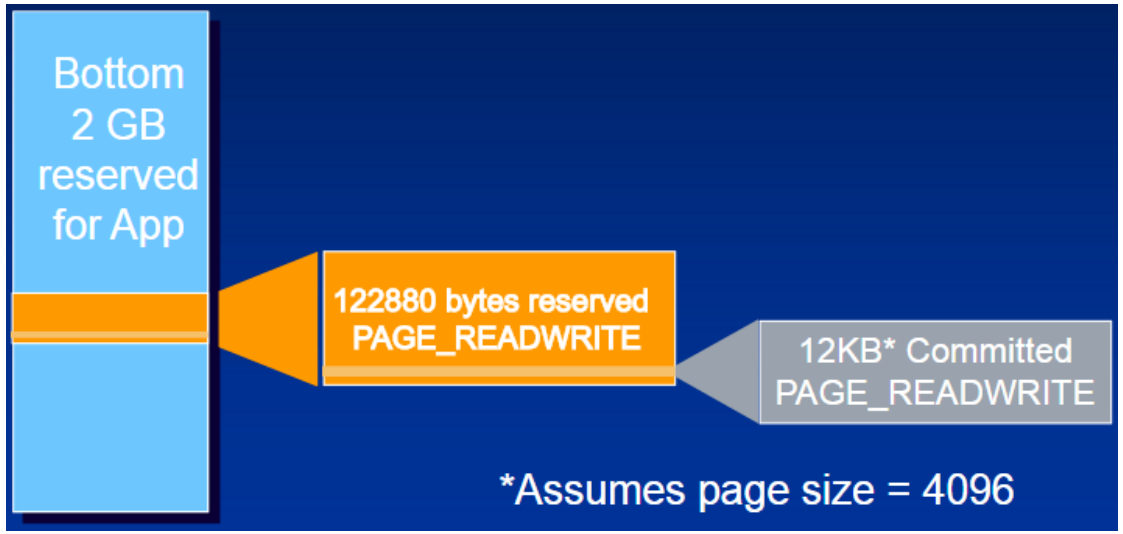
Vezi exemplu pe MSDN

Exemple scurte

```
LPVOID lpMem = VirtualAlloc(
        NULL,
        120000,
        MEM_RESERVE,
        PAGE_READWRITE);
```

```
LPVOID lpBlock = VirtualAlloc(
        lpMem + 6 * 1024,
        7 * 1024,
        MEM_COMMIT,
        PAGE_READWRITE);
```





Heap-uri

Heap-ul permite alocarea și eliberarea dinamică a memoriei și este utilizat când

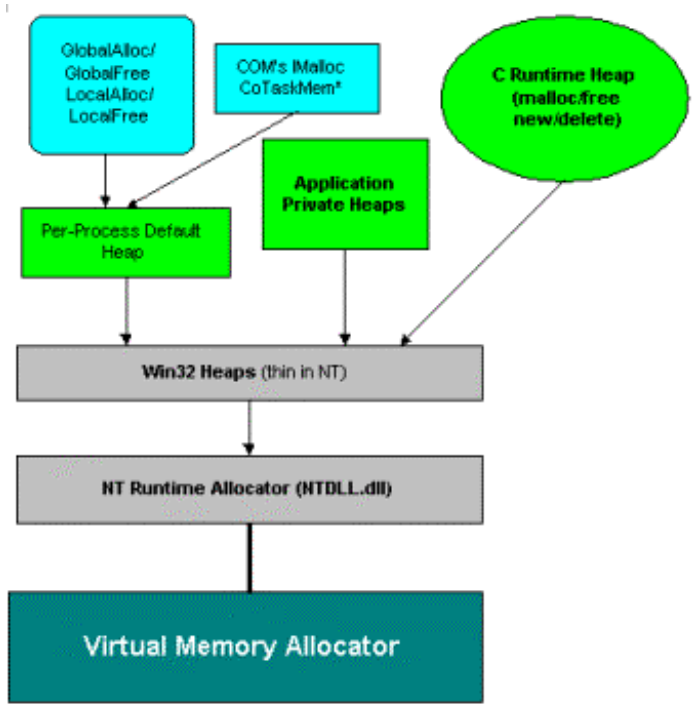
- Numărul sau dimensiunea structurilor de date nu sunt cunoscute înainte de rulare sau
- Structura este prea mare pentru alocarea pe stivă

Heap-ul este mecanismul potrivit pentru gestiunea unui număr mare de obiecte mici

- Granularitate mai fină (permite alocarea unor blocuri de memorie de dimeniuni mai mici) față de funcțiile de gestiunea a memoriei virtuale care alocă cel puțin o pagină de memorie
- Nu implică cunoașterea detaliilor legate de modul de gestiune a memoriei realizat de SO

Fiecare proces are propriul heap implicit și poate crea noi heap-uri

- Memoria dinamică se alocă din aceste heap-uri



Funcțiile `GlobalAlloc` și `LocalAlloc` sunt echivalente și gestionează heap-ul implicit al procesului

CRT creează un heap privat, dar folosește heap-ul implicit pentru alocarea dinamică a memoriei folosind funcțiile malloc, calloc, realloc

Aplicația poate crea multiple heap-uri private

Heap-uri

Un singur heap poate fi suficient – heap-ul implicit

- Caz în care gestiunea se face folosind funcțiile de lucru cu heap-ul implicit (GlobalAlloc, LocalAlloc, etc.) sau funcțiile din biblioteca C (malloc, calloc, realloc, free)

Dar utilizarea mai multor heap-uri are avantaje importante

- **Corectitudine** și **performanță**
  - Fiecare thread poate avea propriul heap
  - Simplifică sincronizarea
  - Poate beneficia de localitatea referințelor
  - Nu apare limitarea la dimensiunea unui singur heap

- **Alocare și dealocare eficientă**
  - Alocarea unor elemente de dimensiune fixă într-un heap mai mic poate fi mai eficient decât alocarea unor elemente de dimensiuni diferite într-un singur heap mai mare
  - Se reduce fragmentarea
  - Întregul heap – cu toate structurile de date conținute – poate fi dealocat pentr-un singur apel de funcție

Heap-ul este un obiect Windows, deci are handle – dar NU este un obiect kernel !

Heap-uri

Obținerea handle-ului către heap-ul implicit al procesului

- Handle-ul poate fi folosit apoi în alte funcții de gestiune a heap-urilor

```
HANDLE WINAPI GetProcessHeap(void);
```

- Returnează NULL în caz de eșec (și nu INVALID_HANDLE_VALUE ca la fișiere)

Obținerea numărului și handle-urilor către toate heap-urile active ale procesului

```
DWORD WINAPI GetProcessHeaps(
           DWORD   NumberOfHeaps,
           PHANDLE ProcessHeaps );
```

Numărul maxim de handle-uri heap care pot fi stocate în bufferul indicat de al doilea parametru

Bufferul care reține handle-urile

Returnează numărul de heap-uri active

```
#include <windows.h>
#include <tchar.h>

#define MAX_HEAPS 100

int _tmain() {
   DWORD NumberOfHeaps;
   DWORD HeapsIndex;
   HANDLE hDefaultProcessHeap;
   HANDLE aHeaps[MAX_HEAPS];

   // Get a handle to the default process heap.
   hDefaultProcessHeap = GetProcessHeap();
   if (hDefaultProcessHeap == NULL) {
      _tprintf(TEXT("Failed to retrieve the default process heap with LastError %d.\n"),
                     GetLastError());
      return 1;
   }
   _tprintf(TEXT("Default heap at address: %#p.\n"),hDefaultProcessHeap);
   // Retrieve handles to the process heaps and print them to stdout.
   NumberOfHeaps = GetProcessHeaps(MAX_HEAPS, aHeaps);
   if (NumberOfHeaps == 0) {
      _tprintf(TEXT("Failed to retrieve heaps with LastError %d.\n"),
                     GetLastError());
      return 1;
   }

   _tprintf(TEXT("Process has %d heaps.\n"), NumberOfHeaps);
   for (HeapsIndex = 0; HeapsIndex < NumberOfHeaps; ++HeapsIndex) {
      _tprintf(TEXT("Heap %d at address: %#p.\n"),
               HeapsIndex, aHeaps[HeapsIndex]);
   }

   return 0;
}
```

```
Default heap at address: 00910000.
Process has 4 heaps.
Heap 0 at address: 00910000.
Heap 1 at address: 74300000.
Heap 2 at address: 00AD0000.
Heap 3 at address: 00EB0000.
Press any key to continue . . .
```

Heap-uri

Creearea unui nou heap

```
HANDLE WINAPI HeapCreate(
          DWORD  flOptions,
          SIZE_T dwInitialSize,
          SIZE_T dwMaximumSize );
```

HEAP_CREATE_ENABLE_EXECUTE (permite executarea codului plasat în heap)
HEAP_GENERATE_EXCEPTIONS (sistemul va genera excepție pentru indicarea eșecului în funcțiile HeapAlloc și HeapReAlloc – în loc de a returna NULL)
HEAP_NO_SERIALIZE (Elimină serializarea. Serializarea asigură excludere mutuală în cazul accesului concurent)

Dimnesiunea inițială în octeți a heap-ului
– se rotunjește în sus la un multiplu de dimensiune de pagină

Dimensiunea maximă a heap-ului (în octeți)

- Dimensiunile sunt de tipul SIZE_T care se traduce în întreg pe 32 sau 64 de biți (în funcție de flag-ul de compilare _WIN32 sau _WIN64)
- Dacă necesarul de memorie (cerut de funcțiile `HeapAlloc` sau `HeapReAlloc`) depășește dimensiunea inițială alocată, sistemul **extinde heap-ul** cu noi pagini până la dimensiunea maximă a heap-ului (`dwMaximumSize`)
- Dacă `dwMaximumSize` este nenul, atunci heap-ul este limitat în dimensiune – are **dimensiune maximă fixă** și
  - Cel mai mare **bloc de memorie care poate fi alocat** din heap
    - Pentru procese pe 32 biți: un pic mai puțin de 512 KB
    - Pentru procese pe 64 biți: un pic mai puțin de 1024 KB
  - Cererile pentru dimensiuni mai mari eșuează, chiar dacă dimensiunea maximă a heap-ul este mai mare
- Dacă `dwMaximumSize` este nul (0), atunci heap-ul poate crește în dimensiune până la limitele memoriei disponibile (spațiului de adrese virtual nealocat încă altor heap-uri sau fișiere swap)
  - Pot fi alocate blocuri de dimensiuni mai mari decât în cazul heap-urilor cu dimensiuni fixe

Heap-urile nu au atribute de securitate ← nu sunt obiecte kernel

- În schimb obiectele de memorie mapată pot fi securizate – după cum vom vedea în curând

Heap-uri

Distrugerea heap-ului

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

→ Handle-ul heap-ului care va fi distrus. Handle returnat de HeapCreate

- Atenție: **să nu distrugem heap-ul implicit** al procesului (nu apelăm HandleDestroy pe handle-ul returnat de GetProcessHeap)
- Distrugerea heap-ului
  - Dealocă zona de memorie virtuală și memoria fizică asociată
  - Dealocă toate structurile de date alocate în heap
    - **Nu este necesară traversarea** și eliberarea element-cu-element

Alocarea unui bloc de memorie în heap

```
LPVOID WINAPI HeapAlloc(
          HANDLE hHeap,
          DWORD  dwFlags,
          SIZE_T dwBytes );
```

→ Handle-ul heap-ului din care se alocă blocul de memorie

→ Opțiuni de alocare

HEAP_GENERATE_EXCEPTIONS (sistemul va genera excepție pentru indicarea eșecului – în loc de a returna NULL)
HEAP_NO_SERIALIZE (Elimină serializarea. Serializarea asigură excludere mutuală în cazul accesului concurent)
HEAP_ZERO_MEMORY (zona alocată va fi inițializată la zero)

↓ Dimensiunea blocului în octeți
- Maxim 0x7FFF8 pentru heap-uri cu dimensiune maximă fixată

- Returnează adresa blocului de memorie alocat, sau NULL (dacă nu este setat flagul pt generarea excepțiilor)
  - Adresa pe 32 sau 64 de biți – depinde de opțiunile de compilator
  - În caz de eroare nu se poate utiliza funcția GetLastError

Heap-uri

Realocarea memoriei – schimbarea dimensiunii blocului de memorie alocat

```
LPVOID WINAPI HeapReAlloc(
          HANDLE hHeap,
          DWORD  dwFlags,
          LPVOID lpMem,
          SIZE_T dwBytes );
```

Handle-ul heap-ului din care se realocă blocul de memorie

Opțiuni de realocare

- **HEAP_GENERATE_EXCEPTIONS** (sistemul va genera excepție pentru indicarea eșecului – în loc de a returna NULL)
- **HEAP_NO_SERIALIZE** (Elimină serializarea. Sserializarea asigură excludere mutuală în cazul accesului concurent)
- **HEAP_REALLOC_IN_PLACE_ONLY** (fără relocarea blocului)
- **HEAP_ZERO_MEMORY** (zona extinsă va fi inițializată la zero)

Noua dimensiune a blocului, în octeți
- Extindere sau micșorare
- Maxim 0x7FFF8 pentru heap-uri cu dimensiune maximă fixată

Adresa blocului de memoriei – returnat de HeapAlloc sau HeapReAlloc

- Returnează adresa blocului de memorie ralocat, sau NULL (dacă nu este setat flagul pt generarea excepțiilor)
  - Adresa poate fi identică cu adresa veche a blocului – dacă nu s-a mutat blocul de memorie pentru a satisface realocarea
  - Dacă eșuează, blocul original este în continuare accesibil

Dealocarea memoriei

```
BOOL WINAPI HeapFree(
          HANDLE hHeap,
          DWORD  dwFlags,
          LPVOID lpMem );
```

Handle-ul heap-ului în care blocul de memorie va fi eliberat. Același heap în care a fost alocat

Opțiune de dealocare **0** sau **HEAP_NO_SERIALIZE** (Elimină serializarea)

Adresa blocului de memoriei – returnat de HeapAlloc sau HeapReAlloc

- În caz de eroare returnează FALSE și se poate folosi funcția GetLastError (aici nu se aplică flagul de generare a excepțiilor)

Heap-uri

Dimensiunea blocului de memorie din heap

```
SIZE_T WINAPI HeapSize(
          HANDLE  hHeap,
          DWORD   dwFlags,
          LPCVOID lpMem );
```

Handle-ul heap-ului în care se află blocul de memorie

Opțiune **0** sau **HEAP_NO_SERIALIZE** (Elimină serializarea)

Adresa blocului de memorie pentru care interogăm dimensiunea – returnat de HeapAlloc sau HeapReAlloc

- Returnează dimensiunea blocului de memorie, în octeți, sau -1 în caz de eșec (nu se poate folosi GetLastError)
- Dacă blocul nu face parte din heap-ul specificat, comportamentul nu este definit

Alte funcții de lucru cu heap-uri

- Funcțiile de alocare `LocalAlloc` și `GlobalAlloc` respectiv funcțiile de eliberare a memoriei `LocalFree` și `GlobalFree`
  - Implementate ca funcții wrapper care apelează `HeapAlloc` în spate, folosind handle-ul heap-ului implicit al procesului
- Funcția `VirtualAlloc` permite specificarea unor opțiuni adiționale pentru alocarea memoriei
  - Dar utilizează o granularitate de alocare de o pagină - în schimb funcțiile pentru heap-uri permit o granularitate mai fină
- Funcțiile `HeapLock` și `HeapUnlock` sunt folosite pentru serializarea accesului la heap
- Funcțiia `HeapCompact` returnează dimensiunea celui mai mare bloc de memorie nealocat
- Funcțiia `HeapWalk` enumeră blocurile din heap

Exemple scurte

Crearea unui heap cu rezervarea unui spațiu de 1 MB (dimensiunea maximă) și alocarea inițială a 2 pagini de memorie

```
HANDLE hHeap = HeapCreate(0, 0x2000, 0x100000);
```

0x2000 este 8192 = 2 * 4096 = 2 * 4KB (o pagină are dimensiunea 4 KB)

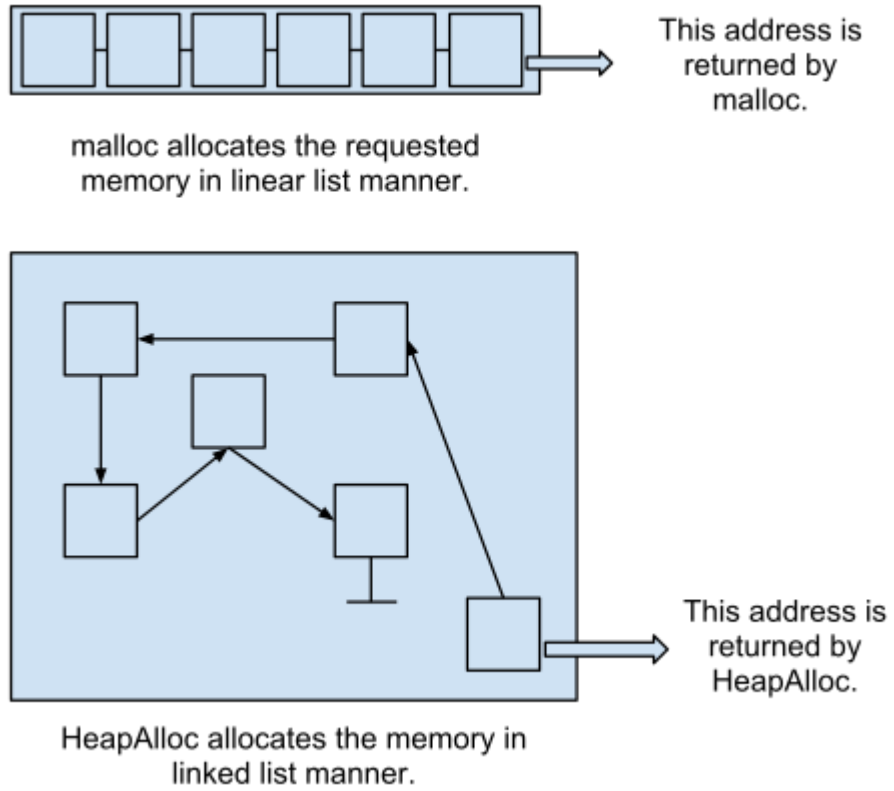
Utilizarea funcțiilor GlobalAlloc și GlobalFree – care folosesc heap-ul implicit al procesului

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

void main() {
   PSECURITY_DESCRIPTOR pSD;

   pSD = (PSECURITY_DESCRIPTOR)GlobalAlloc(GMEM_FIXED, sizeof(PSECURITY_DESCRIPTOR));

   // Handle error condition
   if (pSD == NULL) {
      _tprintf(TEXT("GlobalAlloc failed (%d)\n"), GetLastError());
      return;
   }

   //see how much memory was allocated
   _tprintf(TEXT("GlobalAlloc allocated %d bytes\n"), GlobalSize(pSD));

   // Use the memory allocated

   // Free the memory when finished with it
   GlobalFree(pSD);
}
```

```
GlobalAlloc allocated 4 bytes
```

HeapAlloc vs. malloc

- Funcția malloc este portabilă – face parte din standardul C
  - Alocă memorie din heap-ul implicit al procesului
  - Apelăm free pentru eliberare
- HeapAlloc
  - Apel sistem specific win32
  - Se pot crea mai multe heap-uri
  - Poate fi setat să
    - genereze excepții
    - Inițializeze zona la 0 (zero)
  - Zona alocată nu poate fi mutată (de ex. la redimensionare)
  - Apelăm HeapFree pentru eliberare
- LocalAlloc și GlobalAlloc
  - Alocă memorie din heap-ul implicit al procesului
  - Zona este mutabilă în cadrul heap-ului
  - Are încărcare (overhead) mai mare decât funcțiile HeapAlloc
  - Apelăm LocalFree / GlobalFree pentru eliberare



Exemplu: sortarea liniilor din fișier folosind arbore binar de căutare

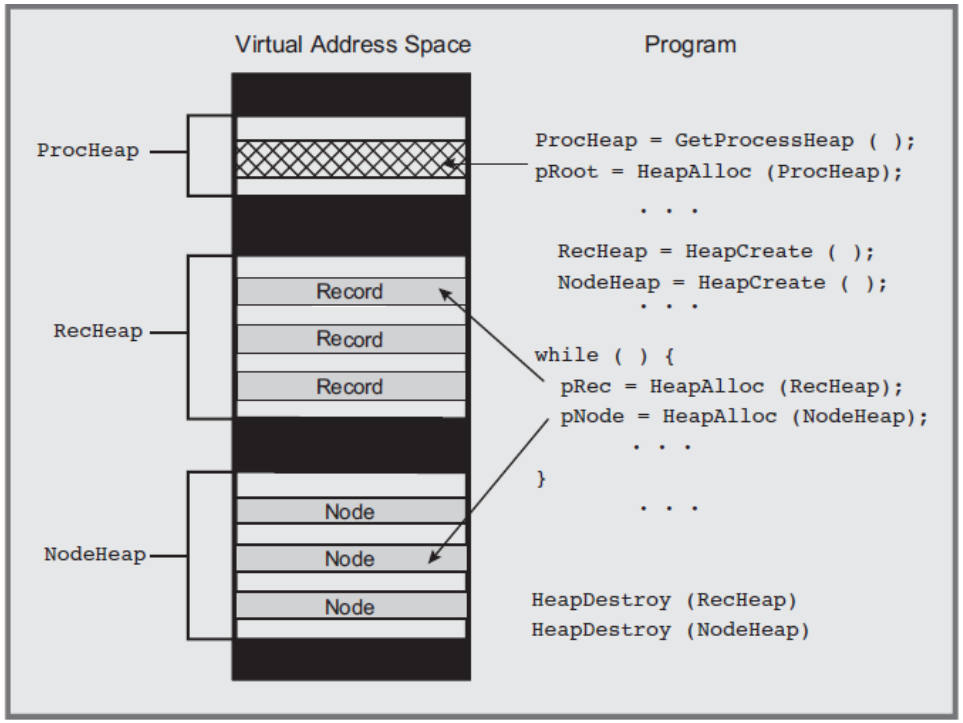
Programul sortBT.c implementează sortarea liniilor dintr-un fișier text

- folosind un arbore binar de căutare și două heap-uri
  - Cheile sunt păstrate în heap-ul: NodeHeap – care reprezintă arborele binar de căutare
    - Cheile sunt considerate primii 8 octeți din linie
    - Fiecare nod are pointeri la fiul stâng și drept și un pointer la înregistrarea data din heap-ul: RecHeap
  - Data este o înregistrare care conține o linie de text din fișierul de intrare. Datele sunt păstrate în RecHeap

  - Observăm că
    - NodeHeap are blocuri de dimensiune fixă
    - RecHeap are blocuri de dimensiuni diferite
      - șiruri de dimensiuni variate

- Exemplul ilustrează
  - Utilizarea heap-urilor
  - Procesarea erorilor de alocare prin tratarea excepțiilor
  - Funcțiile de lucru cu ABC

  - Limitare
    - Potrivit doar pentru fșiere de dimensiuni reduse
      - Păstrează în memoria virtuală fișierul și cheile



Memory Management in Multiple Heaps

Exemplu: sortarea liniilor din fișier folosind arbore binar de căutare

```
/* Chapter 5, sortBT command.   Binary Tree version. */
/* Technique:
1.Scan the input file, placing each key (which is fixed size)
into the binary search tree and each record onto the data heap.
2. Traverse the search tree and output the records in order.
3. Destroy the heap and repeat for the next file. */

#include "Everything.h"

#define KEY_SIZE 8

/* Structure definition for a tree node. */
typedef struct _TREENODE {
      struct _TREENODE *Left, *Right;
      TCHAR key[KEY_SIZE];
      LPTSTR pData;
   } TREENODE, *LPTNODE, **LPPTNODE;

#define NODE_SIZE sizeof (TREENODE)
#define NODE_HEAP_ISIZE 0x8000
#define DATA_HEAP_ISIZE 0x8000
#define MAX_DATA_LEN 0x1000
#define TKEY_SIZE KEY_SIZE * sizeof (TCHAR)
#define STATUS_FILE_ERROR 0xE0000001    // Customer exception
```

```
LPTNODE FillTree(HANDLE, HANDLE, HANDLE);
BOOL Scan(LPTNODE);
int KeyCompare(LPCTSTR, LPCTSTR), iFile; /* for access in exception handler */
BOOL InsertTree(LPPTNODE, LPTNODE);

int _tmain(int argc, LPTSTR argv[])
{
   HANDLE hIn = INVALID_HANDLE_VALUE, hNode = NULL, hData = NULL;
   LPTNODE pRoot;
   BOOL noPrint;
   CHAR errorMessage[256];
   int iFirstFile = Options(argc, argv, _T("n"), &noPrint, NULL);

   if (argc <= iFirstFile)
      ReportError(_T("Usage: sortBT [options] files"), 1, FALSE);
   /* Process all files on the command line. */
```

```
   for (iFile = iFirstFile; iFile < argc; iFile++)
   __try {
      /* Open the input file. */
      hIn = CreateFile(argv[iFile], GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
      if (hIn == INVALID_HANDLE_VALUE)
         RaiseException(STATUS_FILE_ERROR, 0, 0, NULL);

      __try {
         /* Allocate the two growable heaps. */
         hNode = HeapCreate( HEAP_GENERATE_EXCEPTIONS | HEAP_NO_SERIALIZE, NODE_HEAP_ISIZE, 0);
         hData = HeapCreate( HEAP_GENERATE_EXCEPTIONS | HEAP_NO_SERIALIZE, DATA_HEAP_ISIZE, 0);

         /* Process the input file, creating the tree. */
         pRoot = FillTree(hIn, hNode, hData);

         /* Display the tree in key order. */
         if (!noPrint) {
            _tprintf(_T("Sorted file: %s\n"), argv[iFile]);
            Scan(pRoot);
         }
      }
      __finally {
         /* Heaps and file handle are always closed */
         /* Destroy the two heaps and data structures. */
         if (hNode != NULL) HeapDestroy(hNode);
         if (hNode != NULL) HeapDestroy(hData);
         hNode = NULL; hData = NULL;
         if (hIn != INVALID_HANDLE_VALUE) CloseHandle(hIn);
         hIn = INVALID_HANDLE_VALUE;
      }
   } /* End of main file processing loop and try block. */

    /* Handle the exceptions that we can expect - Namely, file open error or out of memory. */
   __except ((GetExceptionCode() == STATUS_FILE_ERROR ||
              GetExceptionCode() == STATUS_NO_MEMORY) ?
              EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
   {
      _stprintf(errorMessage, _T("\n%s %s"), _T("sortBT error on file:"), argv[iFile]);
      ReportError(errorMessage, 0, TRUE);
   }
   return 0;
}
```

Exemplu: sortarea liniilor din fișier folosind arbore binar de căutare

```
LPTNODE FillTree(HANDLE hIn, HANDLE hNode, HANDLE hData)
/* Scan the input file, creating a binary search tree in the
hNode heap with data pointers to the hData heap. */
/* Use the calling program's exception handler. */
{
   LPTNODE pRoot = NULL, pNode;
   DWORD nRead, i;
   BOOL atCR;
   TCHAR dataHold[MAX_DATA_LEN];
   LPTSTR pString;

   /* Open the input file. */
   while (TRUE) {
      pNode = HeapAlloc(hNode, HEAP_ZERO_MEMORY, NODE_SIZE);
      pNode->pData = NULL;
      (pNode->Left) = pNode->Right = NULL;
      /* Read the key. Return if done. */
      if (!ReadFile(hIn, pNode->key, TKEY_SIZE, &nRead, NULL) ||
          nRead != TKEY_SIZE)
         /* Assume end of file on error.All records must be just the right size */
         return pRoot;/* Read the data until the end of line. */
      atCR = FALSE; /* Last character was not a CR. */

      for (i = 0; i < MAX_DATA_LEN; i++) {
         ReadFile(hIn, &dataHold[i], TSIZE, &nRead, NULL);
         if (atCR && dataHold[i] == LF) break;
         atCR = (dataHold[i] == CR);
      }
      dataHold[i - 1] = _T('\0');

      /* dataHold contains the data without the key.
         Combine the key and the Data. */

      pString = HeapAlloc(hData, HEAP_ZERO_MEMORY,
                          (SIZE_T)(KEY_SIZE + _tcslen(dataHold) + 1) * TSIZE);
      memcpy(pString, pNode->key, TKEY_SIZE);
      pString[KEY_SIZE] = _T('\0');
      _tcscat(pString, dataHold);
      pNode->pData = pString;
      /* Insert the new node into the search tree. */
      InsertTree(&pRoot, pNode);

   } /* End of while (TRUE) loop */
   return NULL; /* Failure */
}
```

```
BOOL InsertTree(LPPTNODE ppRoot, LPTNODE pNode)

   /* Insert the new node, pNode, into the binary search tree, pRoot. */
{
   if (*ppRoot == NULL) {
      *ppRoot = pNode;
      return TRUE;
   }
   if (KeyCompare(pNode->key, (*ppRoot)->key) < 0)
      InsertTree(&((*ppRoot)->Left), pNode);
   else
      InsertTree(&((*ppRoot)->Right), pNode);
   return TRUE;
}
```

```
int KeyCompare(LPCTSTR pKey1, LPCTSTR pKey2)

/* Compare two records of generic characters.
The key position and length are global variables. */
{
   return _tcsncmp(pKey1, pKey2, KEY_SIZE);
}
```

```
static BOOL Scan(LPTNODE pNode)

/* Scan and print the contents of a binary tree. */
{
   if (pNode == NULL)
      return TRUE;
   Scan(pNode->Left);
   _tprintf(_T("%s\n"), pNode->pData);
   Scan(pNode->Right);
   return TRUE;
}
```

Fișiere mapate în memorie

Prin intermediul fișierelor mapate în memorie, sistemul permite aplicațiilor să acceseze fișiere de pe disc în același mod cum accesează memoria alocată dinamic – adică prin pointeri

- Se poate mapa o parte sau întregul fișier într-o anumită zonă a spațiul de adrese al procesului
  - Scrierea în fișier se traduce în asignarea unei valori la valoarea referită de un pointer
  - Atenție însă – scrierea efectivă în fișierul de pe disk este transparentă utilizatorului și se face de regulă cu întârziere – pentru îmbunătățirea performanței
    - Se poate impune scrierea imediată în fișier – cu cost în performanță

Avantajele fișierelor mapate în memorie

- Nu este nevoie de operații directe de lucru cu fișierele (ReadFile sau WriteFile), care sunt de regulă mult mai lente
- Nu este nevoie de gestiunea expicită a bufferelor pentru încărcarea fișierului în memorie – sistemul se ocupă în mod eficient
- Structurile de date create în memorie sunt salvate în fundal pe disc
- Se pot folosi algoritmi eficienți pe datele din memorie – chiar dacă fișierul este foarte mare, mai mare decât memoria fizică
  - Cu penalizare datorată paginării
- Mai multe procese pot partaja memorie prin maparea unor zone din spațiul lor de adrese în același fișier sau în fișierul de paginare

SO folosește maparea memoriei în încărcarea DLL-urilor și a fișierelor executabile

Fișiere mapate în memorie

Pentru înțelegerea conceptului analizăm 4 metode diferite de inversare a tuturor octeților dintr-un fișier de intrare

- Metoda 1: un fișier, un buffer
  - Presupune alocarea unui buffer suficient de mare
  - Se citește întregul fișier în memorie
  - Se inversează octeții prin interschimbare (primul cu ultimul, al doilea cu penultimul, etc. până se ajunge la mijlocul fișierului)
  - Se suprascrie fișierul cu datele din memorie

Dezavantaje:
- Nu funcționează pt fișiere mari
- Nu protejează fișierul original împortiva coruperii – ex. prin întreruperea suprascrierii, etc.

- Metoda 2: două fișiere, un buffer
  - Se creează un nou fișier și se alocă un buffer de dimnesiuni reduse (ex. 8KB)
  - Se citește în buffer ultima parte din fișier, se inversează conținutul bufferului, se scrie bufferul în fișierul nou creat
  - Căutarea, citirea, inversarea și scrierea se repetă până se ajunge la începutul fișierului original
  - După procesare fișierele se închid și fișierul original se șterge

Dezavantaje:
- Mai lentă decât prima metodă (la fiecare pas se caută poziția înainte de citire)
- Utilizare ineficientă a discului, mai ales în cazul fișierelor mari

- Metoda 3: un fișier două buffere
  - Se alocă două buffere de dimensiuni reduse (ex. 8MB)
  - În primul buffer se citesc date de la începutul fișierului, în al doilea se citesc de la finalul fișierului
    - Se inversează conținutul bufferelor și se interschimbă bufferele la scriere: primul buffer la final, al doilea la început, etc.
  - Avantaj în utilizarea discului și al memoriei

Dezavantaje:
- Dificulatate în implementare
- Nu protejează fișierul original împortiva coruperii

- Metoda 4: un fișier, nici un buffer – metoda cu mapare în memorie
  - Se deschide fișierul și se mapează în memorie
  - Se inversează conținutul unei zone din spațiul de adrese virtual
  - Avantaje: **sistemul gestionează partea de caching**
    - programatorul nu trebuie să aloce/elibereze explicit memorie, să încarce conținutul fișirerului sau să scrie înapoi conținutul memoriei în fișier

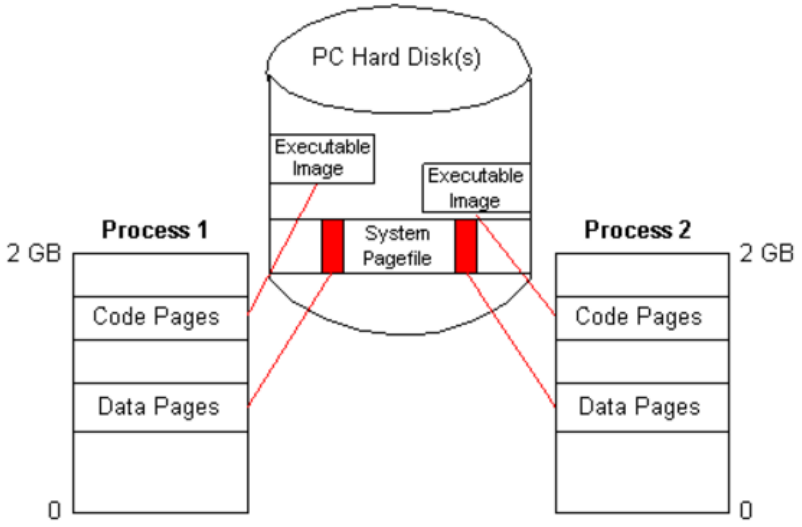
Dezavantaje:
- Problema coruperii datelor dacă intervine o întrerupere (ex. pană de curent, etc.)

Fișiere mapate în memorie

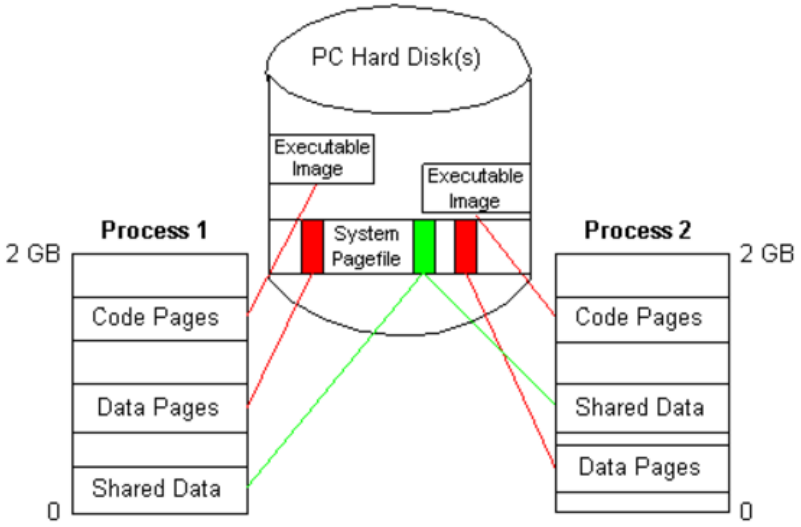
Sistemul folosește maparea fișierelor pentru părțile de cod și date ale aplicațiilor

- Codul și datele sunt reprezentate de pagini în memorie și ambele au ca backup în spate un fișier
  - Copia de siguranță a codului este fișierul executabil
  - Copia de siguranță a datelor este pagefile (fișierul de paginare) al sistemului

Prin extinderea acestui principiu aplicațiile pot partaja date prin copiile de siguranță din fișierul de paginare



Memory used to represent pages of code in processes for Windows NT are backed directly by the application's executable module while memory used for pages of data are backed by the system pagefile.



Processes share memory by mapping independent views of a common region in the system pagefile.

Maparea unui fișier de date în spațiul de adrese al procesului se face pe același principiu

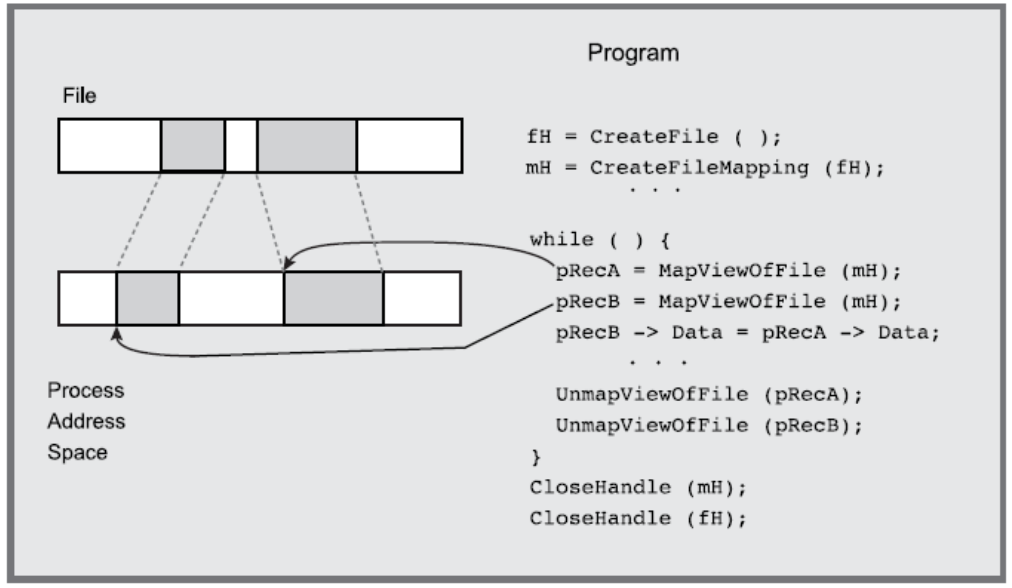
Fișiere mapate în memorie

Pași pentru utilizarea fișierelor mapate în memorie

- Crearea sau deschiderea unui obiect kernel care identifică acel fișier pe disc pe care dorim să îl mapăm în memorie
- Crearea unui obiect kernel de mapare fișier care indică sistemului dimensiunea memoriei necesare pentru mapare și modul de acces
- Maparea în întregime sau parțială a obiectului kernel mapare de fișier în spațiul de adrese al procesului

Pași pentru eliberarea zonei ocupate de fișierul mapat în memorie

- Demaparea obiectului kernel de mapare din spațiul de adrese al procesului
- Închiderea obiectului kernel de mapare
- Închiderea obiectului kernel fișier



A File Mapped into Process Address Space

Fișiere mapate în memorie

Crearea obiectului kernel mapare de fișier

- Creează sau deschide un obiect mapare de fișier cu nume / fără nume pentru un fișier dat

```
HANDLE WINAPI CreateFileMapping(
          HANDLE                hFile,
          LPSECURITY_ATTRIBUTES lpAttributes,
          DWORD                 flProtect,
          DWORD                 dwMaximumSizeHigh,
          DWORD                 dwMaximumSizeLow,
          LPCTSTR               lpName );
```

Handle-ul fișierului deschis. INVALID_HANDLE_VALUE se referă la pagefile
Modul de deschidere trebuie să corespundă cu `flProtect`

Pointer la atributele de securitate (NULL pentru atribute implicite)

Protecția paginilor din obiectul de mapare

- **PAGE_READONLY** (Mapare read-only sau copy-on-write)
- **PAGE_READWRITE** (Mapare read-only, copy-on-write, read/write)
- **PAGE_WRITECOPY** (Mapare read-only sau copy-on-write)
- **PAGE_EXECUTE_READ** (+ execuție)
- **PAGE_EXECUTE_READWRITE** (+ execuție)
- **PAGE_EXECUTE_WRITECOPY** (+ execuție)

Dimensiunea maximă a obiectului de mapare – specificat ca două valori pe 32 de biți (low și high)

Numele obietului de mapare.
Dacă NULL – obiect anonim
Dacă numele există, cere acces

- Fișierul de paginare (pagefile) poate fi folosit pentru partajarea memoriei între procese fără crearea unui fișier dedicat pentru acesta
  - Se specifică INVALID_HANDLE_VALUE pentru hFile
- Obiectul de mapare poate fi securizat – are atribute de securitate (ne reamintim că heap-urile nu aveau)
- Copy-on-write: când memoria mapată este modificată, o copie privată procesului este scrisă în fișierul de paginare, fișierul original nu este modificat
- Dacă parametrii care descriu dimensiunea maximă a obiectului de mapare (`dwMaximumSizeHigh` și `dwMaximumSizeLow`) sunt ambele 0, atunci dimensiunea este setată la dimensiunea curentă a fișierului identificat de hFile
  - Dacă fișierul are dimensiunea 0 apelul eșuează

- Returnează handle către obiectul de mapare nou creat, sau dacă există deja, atunci cel existent (ERROR_ALREADY_EXISTS)
  - NULL în caz de eșec

Fișiere mapate în memorie

- Exemplu

```
int _tmain() {

	// Before executing the line below, C:\ does not have a file called "MMFTest.Dat"
	HANDLE hFile = CreateFile(TEXT("C:\\MMFTest.Dat"), GENERIC_READ | GENERIC_WRITE,
		FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

	// Before executing the line below, the MMFTest.Dat file does exist but has a file size of 0 bytes.
	HANDLE hFileMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, 100, NULL);
	// After executing the line above, the MMFTest.Dat file has a size of 100 bytes.

	// Cleanup
	CloseHandle(hFileMap);
	CloseHandle(hFile);

	// When the process terminates, MMFTest.Dat remains on the disk
	// with a size of 100 bytes.
	return(0);
}
```

Se verifică dacă fișierul de pe disc permite acest mod de acces

Fișierele de date mapate de regulă nu trebuie partajate, de aceea nu s-a dat un nume – este mapare anonimă

Dacă dimensiunea fișierului este mai mică decât dimensiunea cerută pentru mapare, se va extinde dimensiunea fișierului de pe disc

Dacă se creează obiectul de mapare doar în citire, atunci dimensiunea mapării nu poate depăși dimensiunea fișierului

Fișiere mapate în memorie

Deschiderea unui obiectului de mapare existent – pe baza numelui

- Obiectul de mapare trebuie să aibă nume – nu poate fi anonim

```
HANDLE WINAPI OpenFileMapping(
          DWORD   dwDesiredAccess,
          BOOL bInheritHandle,
          LPCTSTR lpName );
```

Modul de acces la obiectul de mapare. Se verifică dacă drepturile cu care s-a creat obiectul permit accesul

Moștenibilitatea handle-ului.
Mai multe vezi la procese

Numele obiectului de mapare creat prin CreateFileMapping

- Returnează handle către obiectul de mapare cu numele specificat, sau NULL

Maparea unei vederi a obiectului de mapare în spațiul de adrese al procesului

```
LPVOID WINAPI MapViewOfFile(
          HANDLE hFileMappingObject,
          DWORD  dwDesiredAccess,
          DWORD  dwFileOffsetHigh,
          DWORD  dwFileOffsetLow,
          SIZE_T dwNumberOfBytesToMap );
```

Handle-ul obiectului de mapare – returnat de CreateFileMapping sau OpenFileMapping

Modul de acces la obiectul de mapare
Se verifică dacă drepturile cu care
s-a creat obiectul permit

FILE_MAP_ALL_ACCESS (mapează o imagine în citire/scriere)
FILE_MAP_COPY (mapează o imagine în copy-on-write)
FILE_MAP_READ (mapează o imagine doar în citire)
FILE_MAP_WRITE (mapează o imagine în citire/scriere)
Pot fi combinate cu:
FILE_MAP_EXECUTE (mapează o imagine în execuție)

Offsetul din fișier de unde începe imaginea mapată –
specificat ca două valori pe 32 de biți (low și high)

Numărul de octeți de mapat din
fișier începând de la offset
- 0 indică până la finalul fișierului

- Offsetul trebuie să fie un multiplu al granularității de alocare (de regulă 64 KB, poate fi obținut prin GetSystemInfo)
- Returnează adresa de început unde s-a mapat vederea fișierului în memorie, sau NULL în caz de eșec

Fișiere mapate în memorie

Exemplu de mapare a unei vederi folosind FILE_MAP_COPY

- Modificările se efectuează într-o copie alocată în paging file

```
// Open the file that we want to map.
HANDLE hFile = CreateFile(pszFileName, GENERIC_READ | GENERIC_WRITE, 0, NULL,
                          OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

// Create a file-mapping object for the file.
HANDLE hFileMapping = CreateFileMapping(hFile, NULL, PAGE_WRITECOPY, 0, 0, NULL);

// Map a copy-on-write view of the file; the system will commit enough
// physical storage from the paging file to accommodate the entire file.
// All pages in the view will initially have PAGE_WRITECOPY access.
PBYTE pbFile = (PBYTE)MapViewOfFile(hFileMapping, FILE_MAP_COPY, 0, 0, 0);

// Read a byte from the mapped view.
BYTE bSomeByte = pbFile[0];

// When reading, the system does not touch the committed pages in the
// paging file. The page keeps its PAGE_WRITECOPY attribute.
// Write a byte to the mapped view.
pbFile[0] = 0;
// When writing for the first time, the system grabs a committed page
// from the paging file, copies the original contents of the page at
// the accessed memory address, and maps the new page (the copy) into
// the process' address space. The new page has an attribute of PAGE_READWRITE.
// Write another byte to the mapped view.
pbFile[1] = 0;
// Because this byte is now in a PAGE_READWRITE page, the system
// simply writes the byte to the page (backed by the paging file).

// When finished using the file's mapped view, unmap it.
// UnmapViewOfFile is discussed in the next section.
UnmapViewOfFile(pbFile);
// The system decommits the physical storage from the paging file.
// Any writes to the pages are lost.

// Clean up after ourselves.
CloseHandle(hFileMapping);
CloseHandle(hFile);
```

Fișiere mapate în memorie

Eliberarea memoriei ocupate de vederea fișierului mapat

```
BOOL WINAPI UnmapViewOfFile(
          LPCVOID lpBaseAddress );
```

Adresa de început a imaginii mapate – returnată de MapViewOfFile

- Chiar dacă handle-ul fișierului mapat este închis de program, sistemul menține fișierul deschis până când se demapează ultima imagine a acelui fișier din memorie
- Reamintim că scrierile paginilor modificate (dirty page) înapoi în fișier se fac cu întârziere – modificările se mențin în cache

Scrierea pe disc a unei porțiuni din imaginea fișierului mapat

```
BOOL WINAPI FlushViewOfFile(
          LPCVOID lpBaseAddress,
          SIZE_T  dwNumberOfBytesToFlush );
```

Adresa de început a imaginii mapate – returnată de MapViewOfFile

Numărul de octeți de scris înapoi pe disc
- Dacă e 0 se scrie toată imaginea mapată înapoi pe disc

- Datorită actualizării "leneșe" a fișierului de pe disc pe baza vederii mapate în memorie, procesele care accesează concurent fișierul folosind operații I/O convenționale (ReadFile, etc.) pot vedea o imagine necoerentă cu ultimele actualizări produse în imaginea mapată în memorie (coerența este asigurată dacă accesul este doar prin imaginea mapată)
  - Nu se recomandă accesul la fișierul mapat și prin operații I/O clasice

Închiderea handle-ului de obiect de mapare

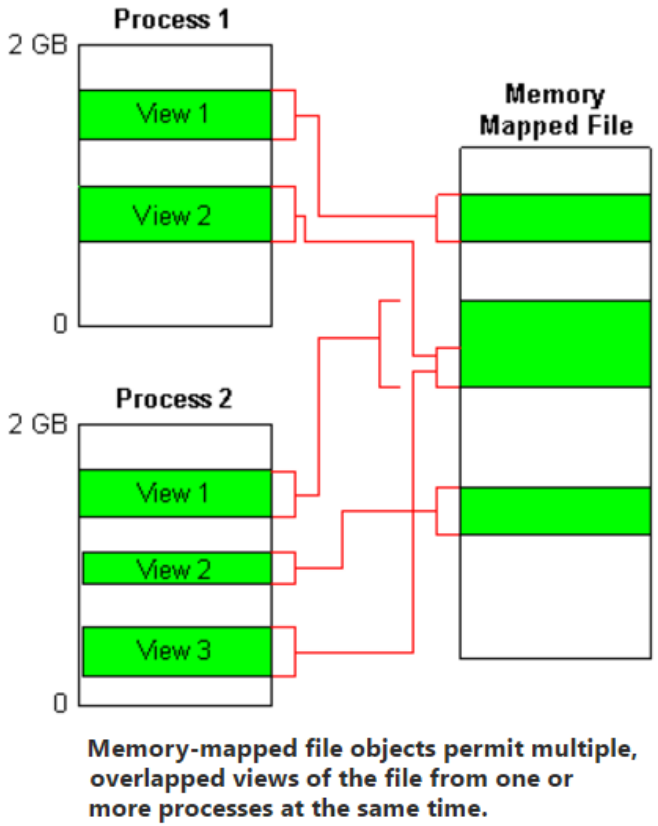
```
BOOL WINAPI CloseHandle(
          HANDLE hObject );
```

Handle-ul obiectului de mapare

Handle-ul returnat de CreateFileMapping se poate închide imediat ce MapViewOfFile revine cu succes – dacă nu dorim să mapăm și alte imagini

Fișiere mapate în memorie

Multiple vederi ale aceluiași obiect de mapare de fișier pot coexista și se pot suprapune

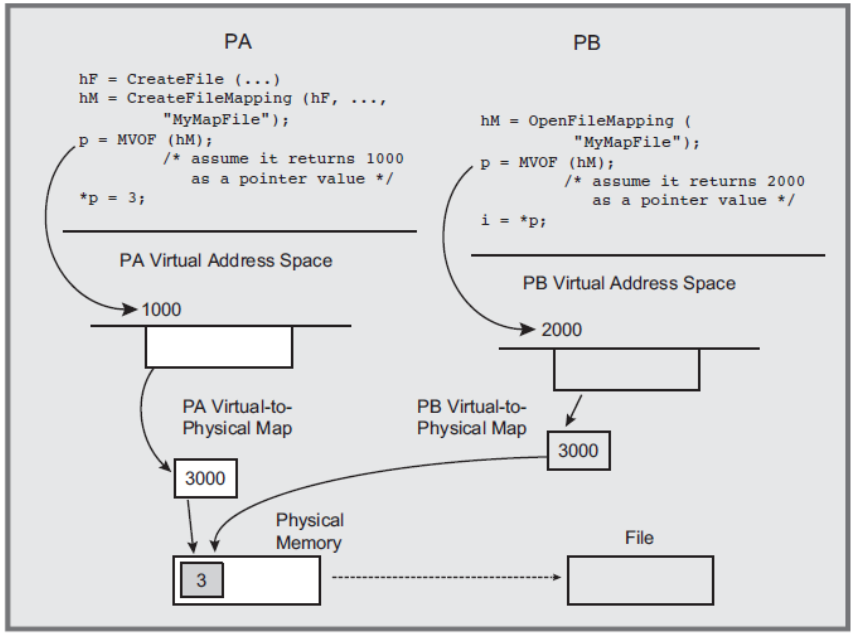


Memory-mapped file objects permit multiple, overlapped views of the file from one or more processes at the same time.

În cazul unui singur proces cu vederi suprapuse, sunt mai multe adrese virtuale în spațiul de adrese al procesului care referă aceeași locație din memoria fizică

Fiecare pagină a fișierului mapat în memorie partajat este reprezentat de o singură pagină din memoria fizică

- Deci toate vederile sunt coerente unele cu altele



Shared Memory

Problemele de sincronizare care pot apare vor fi studiate în curând

Fișiere mapate în memorie

În rezumat: ordinea de creare/mapare și închidere/demapare

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

// Use the memory-mapped file.

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

Sau, putem închide handle-urile, deoarece sistemul menține fișierul deschis până la închiderea ultimei vederi mapate a fișierului

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

// Use the memory-mapped file.

UnmapViewOfFile(pvFile);
```

Limitări

- Limitare specifică Win32
  - Nu permite maparea fișierelor foarte mari (2-3 GB) în spațiul de adrese al procesului
  - Zonele contigue sunt mult mai mici decât maximul teoretic
- Limitări atât în Win32 cât și Win64
  - Zona mapată nu poate fi extinsă – trebuie să știm dimensiunea maximă încă de la crearea mapării
  - Alocarea dinamică a memoriei în regiunea mapată nu este facilitată (în această privință heap-urile sunt mai bine dotate)

Exemplu: inversarea fișierului

```
BOOL FileReverse(PCTSTR pszPathname, PBOOL pbIsTextUnicode) {

   *pbIsTextUnicode = FALSE;  // Assume text is Unicode

   // Open the file for reading and writing.
   HANDLE hFile = CreateFile(pszPathname, GENERIC_WRITE | GENERIC_READ, 0,
      NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

   if (hFile == INVALID_HANDLE_VALUE) {
      chMB("File could not be opened.");
      return(FALSE);
   }

   // Get the size of the file (I assume the whole file can be mapped).
   DWORD dwFileSize = GetFileSize(hFile, NULL);

   // Create the file-mapping object. The file-mapping object is 1 character
   // bigger than the file size so that a zero character can be placed at the
   // end of the file to terminate the string (file). Because I don't yet know
   // if the file contains ANSI or Unicode characters, I assume worst case
   // and add the size of a WCHAR instead of CHAR.
   HANDLE hFileMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE,
      0, dwFileSize + sizeof(WCHAR), NULL);

   if (hFileMap == NULL) {
      chMB("File map could not be opened.");
      CloseHandle(hFile);
      return(FALSE);
   }

   // Get the address where the first byte of the file is mapped into memory.
   PVOID pvFile = MapViewOfFile(hFileMap, FILE_MAP_WRITE, 0, 0, 0);

   if (pvFile == NULL) {
      chMB("Could not map view of file.");
      CloseHandle(hFileMap);
      CloseHandle(hFile);
      return(FALSE);
   }
```

Exemplu: inversarea fișierului

```
// Does the buffer contain ANSI or Unicode?
   int iUnicodeTestFlags = -1;   // Try all tests
   *pbIsTextUnicode = IsTextUnicode(pvFile, dwFileSize, &iUnicodeTestFlags);

   if (!*pbIsTextUnicode) {
      // For all the file manipulations below, we explicitly use ANSI
      // functions because we are processing an ANSI file.

      // Put a zero character at the very end of the file.
      PSTR pchANSI = (PSTR) pvFile;
      pchANSI[dwFileSize / sizeof(CHAR)] = 0;

      // Reverse the contents of the file.
      _strrev(pchANSI);

      // Convert all "\n\r" combinations back to "\r\n" to
      // preserve the normal end-of-line sequence.
      pchANSI = strstr(pchANSI, "\n\r"); // Find first "\r\n".

      while (pchANSI != NULL) {
         // We have found an occurrence....
         *pchANSI++ = '\r';   // Change '\n' to '\r'.
         *pchANSI++ = '\n';   // Change '\r' to '\n'.
         pchANSI = strstr(pchANSI, "\n\r"); // Find the next occurrence.
      }

   }
```

```
else {
      // For all the file manipulations below, we explicitly use Unicode
      // functions because we are processing a Unicode file.

      // Put a zero character at the very end of the file.
      PWSTR pchUnicode = (PWSTR) pvFile;
      pchUnicode[dwFileSize / sizeof(WCHAR)] = 0;

      if ((iUnicodeTestFlags & IS_TEXT_UNICODE_SIGNATURE) != 0) {
         // If the first character is the Unicode BOM (byte-order-mark),
         // 0xFEFF, keep this character at the beginning of the file.
         pchUnicode++;
      }

      // Reverse the contents of the file.
      _wcsrev(pchUnicode);

      // Convert all "\n\r" combinations back to "\r\n" to
      // preserve the normal end-of-line sequence.
      pchUnicode = wcsstr(pchUnicode, L"\n\r"); // Find first '\n\r'.

      while (pchUnicode != NULL) {
         // We have found an occurrence....
         *pchUnicode++ = L'\r';   // Change '\n' to '\r'.
         *pchUnicode++ = L'\n';   // Change '\r' to '\n'.
         pchUnicode = wcsstr(pchUnicode, L"\n\r"); // Find the next occurrence.
      }
   }
```

```
// Clean up everything before exiting.
   UnmapViewOfFile(pvFile);
   CloseHandle(hFileMap);

   // Remove trailing zero character added earlier.
   SetFilePointer(hFile, dwFileSize, NULL, FILE_BEGIN);
   SetEndOfFile(hFile);
   CloseHandle(hFile);

   return(TRUE);
}
```

Exemplu: criptarea folosind fișiere mapate în memorie

Exemplul mapează fișierul de intrare și cel de ieșire în memorie și procesează octeții secvențial

- Există un compromis între complexitatea implementării folosind fișiere mapate în memorie și simplitatea procesării ulterioare

```
/* cci_fMM.c function: Memory Mapped implementation of the
simple Caeser cipher function. */

#include "Everything.h"

BOOL cci_f(LPCTSTR fIn, LPCTSTR fOut, DWORD shift) {
   BOOL complete = FALSE;
   HANDLE hIn = INVALID_HANDLE_VALUE, hOut = INVALID_HANDLE_VALUE;
   HANDLE hInMap = NULL, hOutMap = NULL;
   LPTSTR pIn = NULL, pInFile = NULL, pOut = NULL, pOutFile = NULL;

   __try {
      LARGE_INTEGER fileSize;

      /* Open the input file. */
      hIn = CreateFile(fIn, GENERIC_READ, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
      if (hIn == INVALID_HANDLE_VALUE)
           ReportException(_T("Failure opening input file."), 1);

      /* Get the input file size. */
      if (!GetFileSizeEx(hIn, &fileSize))
           ReportException(_T("Failure getting file size."), 4);
      /* This is a necessar, but NOT sufficient, test for mappability on 32-bit systems */
      if (fileSize.HighPart > 0 && sizeof(SIZE_T) == 4)
           ReportException(_T("This file is too large to map on a Win32 system."), 4);

      /* Create a file mapping object on the input file. Use the file size. */
      hInMap = CreateFileMapping(hIn, NULL, PAGE_READONLY, 0, 0, NULL);
           if (hInMap == NULL)
                ReportException(_T("Failure Creating input map."), 2);

      /* Map the input file */
      /* This program works by mapping the input and output files in their entirity.
      *  You could enhance this program by mapping one block at a time for each file,
      *  much as blocks are used in the ReadFile/WriteFile implementations. This would
      *  allow you to deal with very large files on 32-bit systems. */
      pInFile = MapViewOfFile(hInMap, FILE_MAP_READ, 0, 0, 0);
      if (pInFile == NULL)
            ReportException(_T("Failure Mapping input file."), 3);


      /*  Create/Open output file. MUST have Read/Write access for the mapping to succeed. */
     hOut = CreateFile(fOut, GENERIC_READ | GENERIC_WRITE,
                    0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
      if (hOut == INVALID_HANDLE_VALUE) {
         complete = TRUE; /* Do not delete an existing file. */
         ReportException(_T("Failure Opening output file."), 5);
      }

      /* Map the output file. CreateFileMapping will expand the file if smaller than the mapping. */
      hOutMap = CreateFileMapping(hOut, NULL, PAGE_READWRITE, fileSize.HighPart, fileSize.LowPart, NULL);
      if (hOutMap == NULL)
          ReportException(_T("Failure creating output map."), 7);
      pOutFile = MapViewOfFile(hOutMap, FILE_MAP_WRITE, 0, 0, (SIZE_T)fileSize.QuadPart);
      if (pOutFile == NULL)
          ReportException(_T("Failure mapping output file."), 8);

      /* Now move the input file to the output file, doing all the work in memory. */
      __try {
          CHAR cShift = (CHAR)shift;
          pIn = pInFile;     pOut = pOutFile;
          while (pIn < pInFile + fileSize.QuadPart) {
             *pOut = (*pIn + cShift);    pIn++;     pOut++;
          }
          complete = TRUE;
       }
       __except (GetExceptionCode() == EXCEPTION_IN_PAGE_ERROR ?
                 EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
       {
          complete = FALSE;
          ReportException(_T("Fatal Error accessing mapped file."), 9);
       }

       /* Close all views and handles. */
       UnmapViewOfFile(pOutFile); UnmapViewOfFile(pInFile);
       CloseHandle(hOutMap); CloseHandle(hInMap);   CloseHandle(hIn); CloseHandle(hOut);
       return complete;
   }
   __except (EXCEPTION_EXECUTE_HANDLER) {
   if (pOutFile != NULL) UnmapViewOfFile(pOutFile); if (pInFile != NULL) UnmapViewOfFile(pInFile);
   if (hOutMap != NULL) CloseHandle(hOutMap); if (hInMap != NULL) CloseHandle(hInMap);
   if (hIn != INVALID_HANDLE_VALUE) CloseHandle(hIn); if (hOut != INVALID_HANDLE_VALUE) CloseHandle(hOut);

   /* Delete the output file if the operation did not complete successfully. */
   if (!complete)
   DeleteFile(fOut);
   return FALSE;
  }
}
```

Exemplu: sortarea liniilor într-un fișier mapat în memorie

```
/* Chapter 5. sortFL.  File sorting with memory mapping.
sortFL file */

#include "Everything.h"

/* Definitions of the record structure in the sort file. */
#define DATALEN 56
#define KEY_SIZE 8

typedef struct _RECORD {
   TCHAR key[KEY_SIZE];
   TCHAR data[DATALEN];
} RECORD;

#define RECSIZE sizeof (RECORD)
typedef RECORD * LPRECORD;
int KeyCompare(LPCTSTR, LPCTSTR);

int _tmain(int argc, LPTSTR argv[]) {
   /* The file is the first argument. Sorting is done in place. */
   HANDLE hFile = INVALID_HANDLE_VALUE, hMap = NULL;
   HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
   LPVOID pFile = NULL;
   DWORD result = 2;
   TCHAR tempFile[MAX_PATH];
   LPTSTR pTFile;
   LARGE_INTEGER fileSize;
   BOOL NoPrint;
   int iFirstFile;

   iFirstFile = Options(argc, argv, _T("n"), &NoPrint, NULL);
   if (argc <= iFirstFile)
      ReportError(_T("Usage: sortFL [options] files"), 1, FALSE);

   __try {/* Copy the input file to a temp output file that will be sorted.
             Do not alter the input file. */
      _stprintf_s(tempFile, MAX_PATH, _T("%s.tmp"), argv[iFirstFile]);
      CopyFile(argv[iFirstFile], tempFile, TRUE);

      result = 1; /* tmp file is new and should be deleted. */
      /* Open the file (use the temporary copy). */
      hFile = CreateFile(tempFile, GENERIC_READ | GENERIC_WRITE,
                         0, NULL, OPEN_EXISTING, 0, NULL);
     if (hFile == INVALID_HANDLE_VALUE)
          ReportException(_T("File open failed."), 2);
      /* Get file size. If the file is too large, catch that when it is mapped. */
      if (!GetFileSizeEx(hFile, &fileSize)) ReportException(_T("GetFileSizeEx failed."), 2);
      if (fileSize.QuadPart == 0) { /* There is nothing to sort */ CloseHandle(hFile); return 0; }

      /* Create a file mapping object. Use the file size but add space for the null character. */
      fileSize.QuadPart += 2;
      if (fileSize.HighPart > 0 && sizeof(SIZE_T) == 4)
          ReportException(_T("This file is too large to map on a Win32 system."), 4);
      hMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE, fileSize.HighPart, fileSize.LowPart, NULL);
      if (hMap == NULL) ReportException(_T("Create File map failed."), 3);

      pFile = MapViewOfFile(hMap, FILE_MAP_ALL_ACCESS, 0, 0, 0);
      if (pFile == NULL) ReportException(_T("MapView failed"), 4);

      /* Now sort the file. Perform the sort with the C library - in mapped memory. */
      __try {
           qsort(pFile, (SIZE_T)fileSize.QuadPart / RECSIZE, RECSIZE, KeyCompare);

           /* Print out the entire sorted file. Treat it as one single string. */
           pTFile = (LPTSTR)pFile;
           pTFile[fileSize.QuadPart / TSIZE] = _T('\0');
           if (!NoPrint) PrintMsg(hStdOut, pFile);
      }
      __except (GetExceptionCode() == EXCEPTION_IN_PAGE_ERROR ? EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
      {  ReportError(_T("Fatal Error accessing mapped file."), 9, TRUE); }

         result = 0; /* Indicate successful completion. */
         ReportException(EMPTY, 5); /* Force an exception to clean up. */
         return 0;
       } /* End of inner try-except. */
     __except (EXCEPTION_EXECUTE_HANDLER) {
     /* Catch any exception here. Indicate any error. This is the normal termination. Free all the resources. */

     if (pFile != NULL) UnmapViewOfFile(pFile);
     if (hMap != NULL) CloseHandle(hMap);
     if (hFile != INVALID_HANDLE_VALUE) CloseHandle(hFile);
     if (result != 2) DeleteFile(tempFile);
     return result;
  }
} /* End of _tmain */

int KeyCompare(LPCTSTR pKey1, LPCTSTR pKey2) {
   /* Compare two records of generic characters. The key position and length are global variables. */
   return _tcsncmp(pKey1, pKey2, KEY_SIZE);
}
```

Fișiere mapate în memorie

Partajarea memoriei între două sau mai multe procese

- Pentru ca un proces să poată deschide un obiect de mapare de fișier cu OpenFileMapping, maparea trebuie să creată un nume (să nu fie anonim)
  - Ultimul parametru din CreateFileMapping indică numele
- La crearea obiectului mapare de fișier cu funcția CreateFilaMapping se pot crea mapări care mențin copiile de siguranță în fișierul de paginare – nu într-un fișier specific indicat de programator
  - În loc de handle-ul de fișier se pune INVALID_HANDLE_VALUE
- Atenție la testarea succesului pentru funcția de creare a fișierului
  - În caz de eșec se returnează INVALID_HANDLE_VALUE
  - Această valoare furnizată ca prim parametru la CreateFileMapping face maparea să folosească fișierul de paginare al sistemului - și nu fișierul creat cu CreateFile

```
HANDLE hFile = CreateFile(...);
HANDLE hMap = CreateFileMapping(hFile, ...);
if (hMap == NULL)
    return(GetLastError());
...
```

```
// Create a paging file-backed MMF to contain the edit control text.
// The MMF is 4 KB at most and is named MMFSharedData.
s_hFileMap = CreateFileMapping(INVALID_HANDLE_VALUE, NULL,
   PAGE_READWRITE, 0, 4 * 1024, TEXT("MMFSharedData"));

if (s_hFileMap != NULL) {

   if (GetLastError() == ERROR_ALREADY_EXISTS) {
      chMB("Mapping already exists - not created.");
      CloseHandle(s_hFileMap);

   } else {

      // File mapping created successfully.

      // Map a view of the file into the address space.
      PVOID pView = MapViewOfFile(s_hFileMap,
         FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);

      if (pView != NULL) { ..........
```

Procesul care creează obiectul de mapare

```
// See if a memory-mapped file named MMFSharedData already exists.
HANDLE hFileMapT = OpenFileMapping(FILE_MAP_READ | FILE_MAP_WRITE,
   FALSE, TEXT("MMFSharedData"));

if (hFileMapT != NULL) {
   // The MMF does exist, map it into the process's address space.
   PVOID pView = MapViewOfFile(hFileMapT,
      FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);

   if (pView != NULL) { ........................
```

Procesul care deschide obiectul de mapare

Pointeri cu bază

În fișierul mapat în memorie se pot crea structuri de date care conțin pointeri

- Pointerii sunt relativi la adresa de memorie returnată de MapViewOfFile → la o nouă mapare aceste adrese ar fi lipsite de semnificație

Soluție: utilizarea pointerilor cu bază (adresare bazată pe un pointer)

- De fapt un pointer cu bază este un offset (pe 32 sau 64 de biți) față de un pointer considerat baza
- Permite salvarea structurilor de date dinamice pe disc și reîncărcarea acestora într-o altă locație de memorie astfel încât valorile pointerilor (legăturile) să rămână valide

- Specificare `type __based (base) declarator`

- Exemple:

```
LPTSTR pInFile = NULL;
DWORD __based (pInFile) *pSize;
TCHAR __based (pInFile) *pIn;
```

Contrar convențiilor Windows,
* este impus de sintaxă

```
void *vpBuffer;
```

Pointerul de bază: primește valoare în timpul rulării programului

```
struct llist_t {
    void __based(vpBuffer) *vpData;
    struct llist_t
         __based(vpBuffer) *llNext;
};
```

Lista înlănțuită se realocă relativ la valoarea pointerului de bază

Materiale de studiu

- Jeffrey RICHTER & Christophe NASARRE, Windows via C, C++, 5th edition, Microsoft Press, 2008
  - Capitolele 13, 14 și 15

- Johnson HART, Windows System Programming, 4th edition, Addison Wesley, 2010
  - Capitolul 5

  - Exemplele incluse sunt din cartea de mai sus – vezi arhiva de pe moodle: WSP4_Examples.zip