2. PLIKI MAPOWANE W PAMIĘCI RAM

Tworzenie **O**biektu**M**apowany**Plik**, którego zawartość fizyczna rezyduje w pliku dyskowym.

Tworzenie pliku mapowanego w pamięci:

1. Utwórz/Otwórz plik dyskowy, który będzie mapowany na PAO ➜ ***CreateFile***
2. Utwórz w jądrze **O**biekt**M**apowanie**Pliku** ➜ ***CreateFileMapping***,
3. Zmapuj całość/część **OMP**lik na przestrzeń adresową PAO ➜ ***MapViewOfFile***.

Usuwanie pliku mapowanego w pamięci:

1. Anuluj mapowanie OMPliku na przestrzeń adresową procesu, ➜ **UnmapViewOfFile( )**
2. Zamknij OMPliku ➜ **CloseHandle**( )
3. Zamknij ObiektPlik ➜ **CloseHandle**( )

HANDLE **CreateFileMapping(** // **tylko** tworzy w jądrze ObiektMapowanyPlik

❶	HANDLE	*hFile*,	// handle to file to map, from **CreateFile**
❷	LPSECURITY_ATTRIBUTES	*lpFileMappingAttributes*,	// optional security attributes, **NULL**
❸	DWORD	*flProtect*,	// protection for mapping object
❹	DWORD	*dwMaximumSizeHigh*,	// high-order 32 bits of object size
❺	DWORD	*dwMaximumSizeLow*,	// low-order 32 bits of object **size**
❻	LPCTSTR	***lpName***	// **name** of file-mapping object

);

Zwraca uchwyt do ObiektMapowaniePliku lub w przypadku niepowodzenia **NULL**.

CreateFile w razie niepowodzenia zwraca **INVALID_HANDLE_VALUE (-1).**

***hFile*:** wskaźnik na plik dla którego tworzy się ObiektMapowanyPlik (zwraca wywołanie *CreateFile),*

Jeżeli hFile = (HANDLE) 0xFFFFFFFF to plik będzie współdzielony oraz 3-ci i 4-ty parametr muszą być ustawione.

***lpFileMappingAttributes*:** -wskaźnik do struktury SECURITY_ATTRIBUTES, zazwyczaj **NULL** - standardowe zabezpieczenia i brak dziedziczenia zwróconego uchwytu.

***flProtect*:** PAGE_READONLY -tylko czytanie,
ustawić GENERIC_READ w *CreateFile*;

PAGE_READWRITE -pisanie i czytanie,
ustawić GENERIC_READ I GENERIC_WRITE w *CreateFile*;;

PAGE_WRITECOPY - pisanie i czytanie. Pisanie tworzy prywatną kopię strony.
☞Ustawić GENERIC_READ albo GENERIC_READ | GENERIC WRITE w *CreateFile*;

Dodatkowo można dołączyć atrybuty:

SEC_NOCACHE -żadna ze stron ***pliku mapowanego w pamięci*** **nie może** być buforowana. W trakcie operacji Pisania system będzie aktualizował zawartość tego pliku na dysku częściej niż zwykle.

SEC_IMAGE -informacja, że mapowany plik jest obrazem przenośnego, wykonywalnego pliku.
System sprawdza wówczas jego zawartość, aby ustalić atrybuty ochrony, które ma przypisać różnym stronom mapowanego obrazu (sekcja kodu pliku jest mapowana z atrybutem PAGE_EXECUTE_READ, sekcja danych z atrybutem PAGE_READWRITE).
SEC_IMAGE każe zamapować obraz pliku i ustawić odpowiednio ochronę strony.

***dwMaximumSizeHigh, dwMaximumSizeLow*:** -maksymalne rozmiary pliku w bajtach.
Potrzeba dwóch 32-bitowych wartości, gdyż Windows obsługuje pliki potrzebujące 64 bity.
➔ Dla plików nie większych niż **4GB**, ***dwMaximumSizeHigh*** zawsze **0**.
☛Jeśli obiekt mapowanie odzwierciedla aktualny rozmiar pliku, można podać **0** dla obu.
☛Gdy wyłącznie czytanie lub nie dochodzi zmiany wielkości, można ustawić oba parametry na **0**.

***lpName*:** -nazwa **O**biektu**M**apowany**Plik**.
Gdy plik mapowany w pamięci **nie** jest **współdzielony** przez inne procesy parametr ten ustawia się na **NULL**.

Wywołując **CreateFileMapping:**
-z flagą *PAGE_READWRITE*, system sprawdza, czy plik danych na dysku ma przynajmniej taki rozmiar jak w parametrach *dwMaximumSizeHigh* i *dwMaxitnumSizeLow.*
Jeśli plik jest mniejszy to system wydłuży go;
wydłużenie zapewnia istnienie odpowiednio dużej pamięci fizycznej, gdy plik będzie używany jako plik mapowany w pamięci.
-z flagą PAGE_READONLY lub PAGE_WRITECOPY, rozmiar przekazywany do funkcji musi być większy niż rozmiar fizyczny pliku dyskowego.

Funkcja **CreateFileMapping** tworzy tylko ObiektMapowanyPlik,
- **nie rezerwuje** obszaru przestrzeni adresowej,
- **nie mapuje** pamięci pliku na ten obszar.

Po utworzeniu **ObiektMapowanyPlik** należy **zarezerwować** obszar przestrzeni adresowej PAO na dane pliku i przydzielić te dane jako pamięć fizyczną mapowaną na obszar PAO.

LPVOID MapViewOfFile(// przydziela **pamięć fizyczną RAM** mapowanemu plikowi

HANDLE	*hFileMappingObject*,	// file-mapping object to map into address space
DWORD	*dwDesiredAccess*,	// access mode
DWORD	*dwFileOffsetHigh*,	// high-order 32 bits of file offset
DWORD	*dwFileOffsetLow*,	// low-order 32 bits of file offset
DWORD	*dwNumberOfBytesToMap*	// number of bytes to map

);

Zakończona sukcesem zwraca adres bazowy **pb** w PAO, mapowanego obszaru.

***hFileMappingObject*:** uchwyt na **O**biekt**M**apowanu**Plik**, zwrócony przez *CreateFileMapping*

***dwDesiredAccess*:** w jaki sposób będzie się korzystać z danych pliku:

FILE_MAP_WRITE, FILE_MAP_READ,
FILE_MAP_ALL_ACCESS, **FILE_MAP_COPY**

Wcześniej należy ustawić odpowiednie parametry w **CreateFileMapping.**

dwFileOffsetHigh, dwFileOffsetLow*:** informują system, który bajt pliku ma być zmapowany jako pierwszy bajt ***widoku.

☛ Nie trzeba od razu mapować całego pliku.
☛ Można zmapować jego fragment zwany ***widokiem***.

Przesunięcie musi być wielokrotnością systemowej ziarnistości alokacji (*allocation granularity*), w systemach 32-bitowych 64 KB.

dwNumberOfBytesToMap: jaka część pliku będzie zmapowana na przestrzeń adresową (obszar przestrzeni adresowej, który ma być zarezerwowany).
Dla wartości **0** system spróbuje zmapować ***widok*** od ***Offsetu*** aż do końca pliku.

Wywołując funkcję **MapViewOfFile** z flagą **FILE_MAP_COPY**, system przydzieli pamięć fizyczną z systemowego **pliku wymiany**.
Czytając mapowany widok pliku, system nie korzysta ze stron przydzielonych w pliku wymiany.
Gdy wątek procesu po raz pierwszy zapisuje coś pod adresem pamięci w *ramach mapowanego widoku pliku*, system natychmiast:
- skopiuje stronę oryginalnych danych do jednej ze stron przydzielonych w pliku wymiany,
- zmapuje tę kopię na przestrzeń adresową procesu.

Od tej chwili wątki będą sięgać do jego **lokalnej** kopii danych i nie będą mogły odczytać ani zmodyfikować oryginału.

Po wykorzystaniu zmapowanego obszaru pliku w PAO należy go zwolnić.

BOOL **UnmapViewOfFile**(LPCVOID *lpBaseAddress*);

Funkcja zwalnia obszar PAO, kopiując wprowadzone w nim zmiany do **pliku** na **dysku**
Zakończona sukcesem zwraca wartość niezerową.

lpBaseAddress: adres bazowy zwalnianego obszaru - address where mapped view begins
Jest to wartość zwrócona wcześniej przez funkcję **MapViewOfFile**.

Uwaga: system buforuje strony z danymi pliku i nie aktualizuje na bieżąco dyskowego obrazu pliku podczas pracy z jego mapowanym widokiem.

Funkcja **FlushViewOfFile** zmusza system aby strony były natychmiast zapisywane na dysku.

```
BOOL FlushViewOfFile(
    LPCVOID     lpBaseAddress,          // start address of byte range to flush
    DWORD       dwNumberOfBytesToFlush  // number of bytes in range
);
```

Zakończona sukcesem zwraca wartość niezerową

lpBaseAddres: adres bajtu w widoku pliku mapowanego, zaokrąglony w dół do granicy strony.

dwNumberOfBytesToFlush: liczba bajtów zrzucanych na dysk.
System zaokrągla tę liczbę w górę, tak aby całkowita liczba bajtów była wielokrotnością objętości strony.
Wywołanie *FlushViewOfFile* bez wcześniejszej zmiany danych na stronie, powoduje natychmiastowy powrót w miejsce wywołania, nie zapisując niczego na dysku.

2.1. Dostęp do *istniejącego* już ObiektuMapowaniePliku

```
HANDLE OpenFileMapping(
    DWORD   dwDesiredAccess,  // access mode: FILE_MAP_WRITE, FILE_MAP_READ,
                              //              FILE_MAP_ALL_ACCESS, FILE_MAP_COPY
    BOOL    bInheritHandle,   // TRUE indicates that the new process inherits the handle.
    LPCTSTR lpName            // pointer to name of file-mapping object
);
```

Funkcja najpierw sprawdza zabezpieczenia, a dopiero potem zwraca wartość uchwytu.
Jeśli użytkownik ma prawo dostępu do istniejącego **O**MapowaniePliku, funkcja zwraca uchwyt.
Jeśli nie ma prawa funkcja zwróci NULL, a wywołanie *GetLastError* wartość **ERROR_ACCESS_DENIED (5)**.

Poniższe wywołanie umożliwia dostęp do istniejącego Obiektu z prawem czytania.

```
HANDLE hFileMapping = OpenFileMapping(FILE_MAP_READ, FALSE, "MyFileMapping");
```

2.2. Pliki mapowane w pamięci reprezentujące plik wymiany

Można utworzyć plik mapowany w pamięci, reprezentujący **systemowy plik wymiany**.
Należy wywołać ***CreateFileMapping*** z parametrem ***hFile*** = **INVALID_HANDLE_VALUE**.

```
HANDLE hFile = (HANDLE) 0xffffffff;          // = -1
```

system przydzieli pamięć fizyczną w **systemowym pliku wymiany**.
O ilości przydzielonej pamięci decydują parametry ***CreateFileMapping***.

Po utworzeniu **O**biekt**M**apowany**P**lik i zmapowaniu jego widoku na przestrzeń adresową procesu, może on używać go, jakby to był obszar pamięci RAM.
Chcąc dzielić te dane z innymi procesami, należy wywołac ***CreateFileMapping*** z ustawionym parametr ***lpName*** na "**łańcuch_tekstowy**".

Gdy jakiś proces zechce sięgnąć do tej pamięci, wystarczy, że wywoła *CreateFileMapping* lub *OpenFileMapping* i przekaże tę samą nazwę przez ***lpName***.

Należy sprawdzać, czy wartość zwracana przez *CreateFile* nie wskazuje na błąd.
Jeśli wywołanie ***CreateFile*** kończy się niepowodzeniem, funkcja zwraca **INVALID_HANDLE_VALUE**.

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

☞ **Programista nie sprawdził** czy **udało się utworzyć plik.**

W efekcie wywołanie ***CreateFileMapping*** może przekazać przez parametr ***hFile*** wartość: **INVALID_HANDLE_VALUE**,
co spowoduje zmapowanie **pliku wymiany** a nie planowany pliku dyskowego.
Gdy dojdzie do usunięcia ObiektMapowanyPlik, dane zapisane w jego pamięci (w pliku wymiany) zostaną zniszczone przez system.

Program **Mapp1** zapisuje do pliku dyskowego tekst: **qwert asdfgh**.
Następnie mapuje plik w przestrzeni PAO i dopisuje do niego tekst: **123456** .

```
#include <windows.h>
#include <cstdlib>
#include<cstdio>
using namespace std;
#define BYTEStoREAD  35
int main()                                                    // Mapp1
{
DWORD readed = 0,  writed = 0;
char buf[BYTEStoREAD],   buf1[BYTEStoREAD];
char nameF[30] = "dlaMapp1.txt",    *pb ;
 HANDLE  f,  ff, fm;
  f = CreateFile(nameF, GENERIC_READ|GENERIC_WRITE,0,0, CREATE_ALWAYS, 0,0);
     WriteFile(f, "qwert asdfgh", 12, &writed, NULL);
                                   printf("rozmiar pliku: %d \n", GetFileSize(f, NULL));

//--------------Utworzenie obiektu reprezentującego plik zmapowany
  fm = CreateFileMapping(
     f,                          // wskaznik na ObiektPlikowy od  CreateFile
     NULL,                       // standardowe atrybuty bezpieczenstwa
     PAGE_READWRITE,             // czytanie i pisanie (powiazany z 2-gim par. CreateFile)
     0,                          // maly plik
     sizeof(buf),                // rozmiar pliku
     NULL );                     // bez wlasnej nazwy

//-------Przydzielenie obszaru w PAO na odwzorowanie pliku
  pb = (char*)MapViewOfFile(
     fm,                                 // uchwyt zwrócony przez  CreateFileMapping
     FILE_MAP_WRITE,                     // tylko do pisania
     0,                                  // maly plik
     0,                                  // mapowanie od poczatek pliku
     0  );                               // mapowania calego pliku
memcpy(pb + writed, "123456", 6);    // modyfikacja pliku w PAO

UnmapViewOfFile((void *)pb);  // zwolnienie pamieci przeznaczonej na odwzorowanie pliku
CloseHandle(fm);   CloseHandle(f);  ;    // zamkniecie uchwytow do Obiektów jądra

puts("Odczyt funkcjami systemu Windows po modyfikacji:");
   ff = CreateFile(nameF, GENERIC_READ, 0,0, OPEN_EXISTING, 0,0);
                                   printf("rozmiar pliku: %d \n", GetFileSize(ff, NULL));
   ReadFile(ff, buf1, BYTEStoREAD, &readed, NULL);
   puts(buf1);
   CloseHandle(ff);
//getchar();
 return 0;
}
```

```
rozmiar pliku: 12
Odczyt funkcjami systemu Windows po modyfikacji
rozmiar pliku: 35
qwert asdfgh123456
```


Program **Mapp2** tworzy roboczy plik tekstowy i *coś* w nim zapisuje.
Następnie używając techniki Mapowania Pliku: -kopiuje fragmenty pliku do tablicy tekstowej,
-nadpisuje fragment pliku.

```
#include <windows.h>
#include <cstdlib>
#include<cstdio>
using namespace std;
int main()                    // Mapp2  mapowanie pliku danych w przestrzeni procesu
{
HANDLE  hF = NULL;                  // uchwyt do pliku
HANDLE  hMapFile = NULL;            // uchwyt do obiektu reprezentujacego plik zmapowany
char *pMapFile;                     // wskaznik na poczatek obszaru zmapowanego pliku w PAO
char str[3][7], nameF[30] = "dlaMapp2.txt";
char Buf[] = "abcdefghijklmnoprstuvwz";
DWORD writed = 0;
int size = sizeof(str[0]) - 1, i,    sizeBuf = sizeof(Buf)-1;
hF = CreateFile( nameF, GENERIC_READ|GENERIC_WRITE, 0,0, OPEN_ALWAYS, 0, 0 );
     if (hF == INVALID_HANDLE_VALUE) { printf("CreateFile error: %d.\n", GetLastError() );  getchar(); return(1); }
BOOL wynik = WriteFile(hF, Buf, sizeBuf, &writed, NULL);
                          if (!wynik) {   printf("ReadFile error %d.\n", GetLastError() ) ; getchar(); return (3); }
DWORD  sizeF = GetFileSize(hF, 0);    printf("sizeFile = %d\n", sizeF);
hMapFile = CreateFileMapping(hF,  NULL, PAGE_READWRITE, 0,  sizeF+8, NULL);
          if(hMapFile==NULL) { printf("CreateFileMapping error: %d\n", GetLastError() );  getchar();  return 1; }
pMapFile = (char *)MapViewOfFile(hMapFile,  FILE_MAP_ALL_ACCESS, 0,  0,  0);
                    if(pMapFile==NULL) { printf("Brak możliwości przydzielenia PAO.");  getchar();  return 1; }
     puts(pMapFile);   //  ⬅  wydruk zawartości pliku
// mod. 1:  umieszczenie fragmentów pliku w postaci łańcuchów,  w tablicy tekstowej ----------
   for (i=0; i<3; i++) {
                          memcpy(str[i],  pMapFile + i*size,  size);
                          str[i][6]='\0';   }
   printf("\n Zawartosc tablicy z fragmentami pliku:\n");  for (i=0; i<3; i++)    puts(str[i]  );
// mod. 2:  nadpisanie fragmentu pliku ----------------------------------------------------------------
                          memcpy(pMapFile+1*size,  "123456",  size);
UnmapViewOfFile(pMapFile);
CloseHandle(hF);  CloseHandle(hMapFile);

// --odczyt z pliku po modyfikacji funkcjami jezyka C
//  nie stosować w programach docelowych
puts("---odczyt z pliku po modyfikacji:");
    char zn;
    FILE *pF1 = fopen(nameF, "r");
    while ((zn=getc(pF1)) !=EOF) putc(zn, stdout);
    fclose(pF1);
return 0;
}
```

```
sizeFile = 23
abcdefghijklmnoprstuvwz

Zawartosc tablicy z fragmentami pliku:
        abcdef
        ghijkl
        mnoprs

----odczyt z pliku po modyfikacji:
abcdef123456mnoprstuvwz
```

2.3. Adres bazowy pliku mapowanego

Funkcja *MapViewOfFileEx* pozwala sugerować konkretny adres na mapowanie pliku.

LPVOID **MapViewOfFileEx**(

HANDLE *hFileMappingObject*,

DWORD *dwDesiredAccess*,

DWORD *dwFileOffsetHigh*,

DWORD *dwFileOffsetLow*,

DWORD *dwNumberOfBytesToMap*,

LPVOID ***lpBaseAddress*** // suggested **starting address** for mapped view

);

Ostatni parametr służy do przekazania adresu bazowego przewidzianego na mapowanie pliku.

Adres ten musi wypadać na granicy ***ziarnistości alokacji*** (64 KB); w przeciwnym razie *funkcja* zwraca błąd = NULL

zaś *GetLastError* ERROR_MAPPED_ALIGNMENT (1132).

Jeśli system nie może zmapować pliku we wskazanym miejscu (np.: plik jest zbyt duży) funkcja zwraca NULL, i nie próbuje szukać innej, nadającej się przestrzeni adresowej.

Wywołując *MapViewOfFile**Ex*** należy podać adres należący do strefy trybu użytkownika aktualnego procesu, gdyż inaczej funkcja zwróci NULL.

Ustawienie parametr ***lpBaseAddress*** na NULL powoduje, że funkcja działa jak *MapViewOfFile.*

MapViewOfFileEx jest przydatna przy dzieleniu danych z innymi procesami poprzez **plik mapowany w pamięci**.

Może by potrzebny taki plik dostępny pod ***określonym adresem***.

2.4. Uwagi o zamykaniu **O**biekt**M**apowanie**Pliku** i ObiektuPliku

Praca z plikami mapowanymi w pamięci polega na:

-utworzeniu/Otwarciu **O**biektu**M**apowaniu**Pliku,**

-użyciu tego Obiektu do zmapowania widoku **pliku** na przestrzeń adresową procesu.

Aby zamknąć OMPliku, wystarczy wywołać dwukrotnie funkcję ***CloseHandle*** - po jednym razie na każdy uchwyt.

Wywołanie funkcji ***MapViewOfFile*** zwiększa licznik użyć ObiektuPliku oraz OMPliku.

```
HANDLE hFile = CreateFile(...);
HANDLE hFileMap = CreateFileMapping(hFile, . ..);
PVOID pvFile = MapViewOfFile(hFileMap, ...);
// kod korzystający z pliku mapowanego w pamięci.

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

```
HANDLE hFile = CreateFile(...);
HANDLE hFileMap = CreateFileMapping(hFile, . ..);
        CloseHandle (hFile) ;
PVOID pvFile = MapViewOfFile(hFileMap, ...);
        CloseHandle(hFileMapping);
// kod korzystający z pliku mapowanego w pamięci.

UnmapViewOfFile(pvFile) ;
```

Tworząc dodatkowe OMPliku na podstawie tego **samego pliku** lub mapując więcej niż jeden widok tego samego OMPliku, nie można zbyt szybko wywołać *CloseHandle* –

będą potrzebne te uchwyty w dodatkowych wywołaniach *CreateFileMapping* i *MapviewOfFile.*

W programie **Mapp1,2** funkcje języka C++ **memcpy( )** i **memmove( )**, działające na PAO , wykorzystano do modyfikacji danych w dyskowym pliku tekstowym, poprzez jego **zmapowanie**.

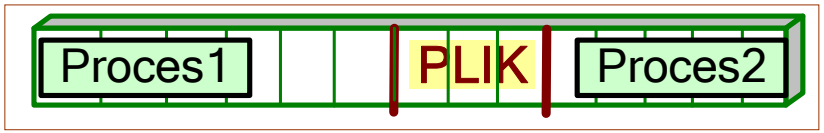
void ***memcpy(**void ***s1,** const void ***s2,** size_t **n)**;

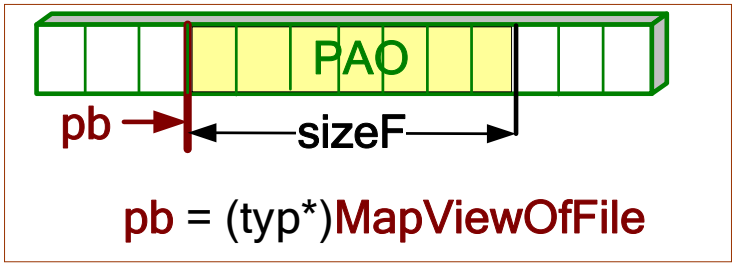
Kopiuje **n** bajtów z miejsca wskazywanego przez **s2** w miejsce wskazywane przez **s1**.

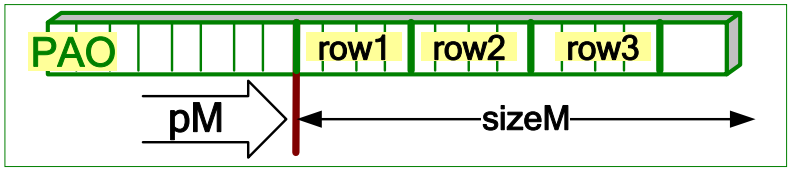
void ***memmove**(void ***s1,** const void ***s2,** size_t **n**);

Kopiuje **n** bajtów z miejsca wskazywanego przez **s2** w miejsce przez **s1**; korzysta z buforu, aby możliwe było kopiowanie do obszaru, który zachodzi na obszar źródłowy. Zwraca wartość **s1**

KU PAMIęCI







W języku C++ dostępny jest operator **New**

double ***B** = **new** double **[rozmiar]**;

double (***pM**)[50]; **pM** = **new** double [**row**][50];

Uwaga:

Funkcje typu:

FILE *pF1 = fopen(nameF, "r"),

zn=getc(pF1), putc(zn, stdout),

fclose(pF1)

i podobne są funkcjami języka C/C++, obsługującymi operacje plikowe.

W programach docelowych nie stosować w operacjach plikowych funkcji języka C/C++, lecz wyłącznie funkcje Systemu Operacyjnego.

Zadanie 2.1

W programie **Mapp2** odczyt kontrolny zawartości plików funkcjami języka C++, **zastąpić funkcjami Systemu Operacyjnego**.

Zadanie 2.2

Plik dyskowy zawiera wektor V[12] liczb typu **double**, zapisany w postaci binarnej.

Napisać program, *wykorzystujący technikę mapowania pliku*, który dopisze do wektora:

- cztery nowe liczby na końcu wektora,
- trzy nowe liczby w środku wektora.

➔ Operacje dyskowe realizować wyłącznie funkcjami systemu operacyjnego.

Zadanie 2.3

Plik dyskowy zawiera macierz M[8][11] liczb typu **double**, zapisaną w postaci binarnej.

Napisać program, *wykorzystujący technikę mapowania pliku*, który dopisze do macierzy:

-dwa nowe wiersze na końcu macierzy,

-dwa nowe wiersze w środku macierzy (np. za wierszem 4-tym).

➔ Operacje dyskowe realizować wyłącznie funkcjami systemu operacyjnego.