```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAXPAROLA 30
#define MAXRIGA 80

int main(int argc, char *argv[])
{
  int freq[MAXPAROLA] ; /* vettore di contatori
  delle frequenze delle lunghezze delle parole */
  char riga[MAXRIGA] ;
  int i, inizio, lunghezza ;
  FILE * f ;

  for(i=0; i<MAXPAROLA; i++)
    freq[i]=0 ;

  if(argc != 2)
  {
    fprintf(stderr, "ERRORE, serve un parametro con il nome del file\n");
    exit(1);
  }
  f = fopen(argv[1], "rt") ;
  if(f==NULL)
  {
    fprintf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
    exit(1);
  }

  while( fgets( riga, MAXRIGA, f ) != NULL )
```

Managing main memory

Memory Mapping

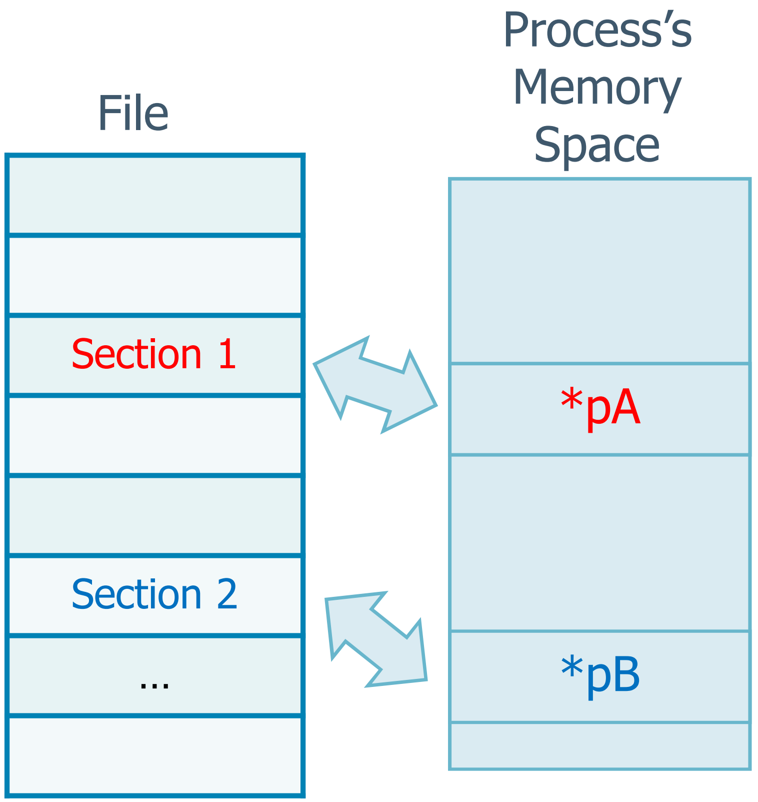
Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

Memory Management

- Windows provides memory-mapped files to
  - Associate a process's address space with a file
  - Allow the OS to manage all data movement between the file and memory while the user just cope with memory address space
  - Permit the programmer to manipulate data structures without file I/O functions
    - ReadFile, WriteFile, SetFilePointer, etc.

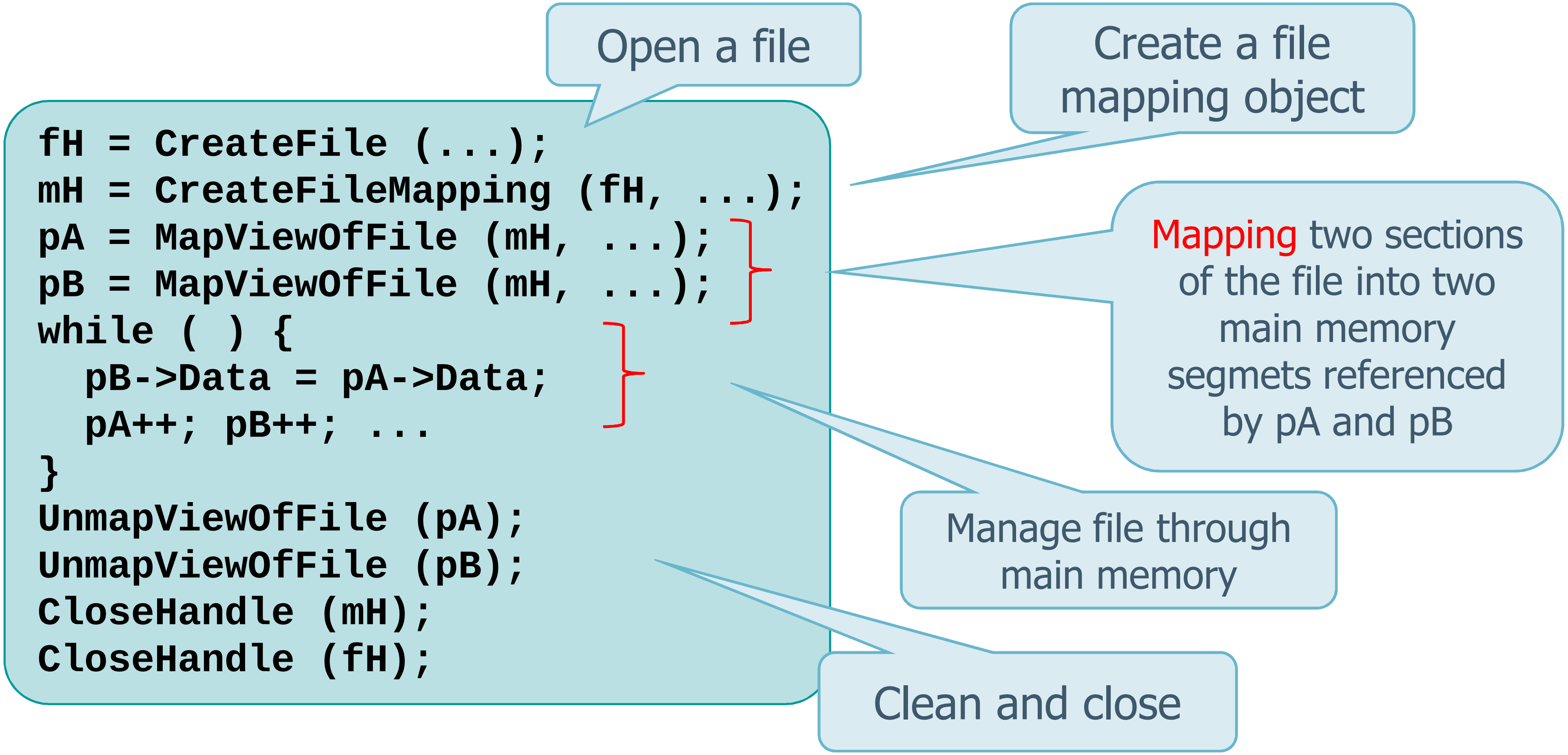


Memory Management

- The advantages to mapping the virtual memory space directly to normal files include
  - Applications can be significantly **faster**
    - The program can maintain dynamic data structures conveniently in permanent files
    - Memory-based algorithms can process file data
      - **In-memory** algorithms (string processing, sorts, search trees) can directly process data
      - The file may be much larger than the available physical memory
  - There is no need to manage buffers and the file data they contain
  - Multiple processes can **share** memory, and the file views will be coherent

Logic

- Memory-mapped file used inside a **single** process



```
fH = CreateFile (...);
mH = CreateFileMapping (fH, ...);
pA = MapViewOfFile (mH, ...);
pB = MapViewOfFile (mH, ...);
while ( ) {
  pB->Data = pA->Data;
  pA++; pB++; ...
}
UnmapViewOfFile (pA);
UnmapViewOfFile (pB);
CloseHandle (mH);
CloseHandle (fH);
```

Open a file

Create a file mapping object

Mapping two sections of the file into two main memory segmets referenced by pA and pB

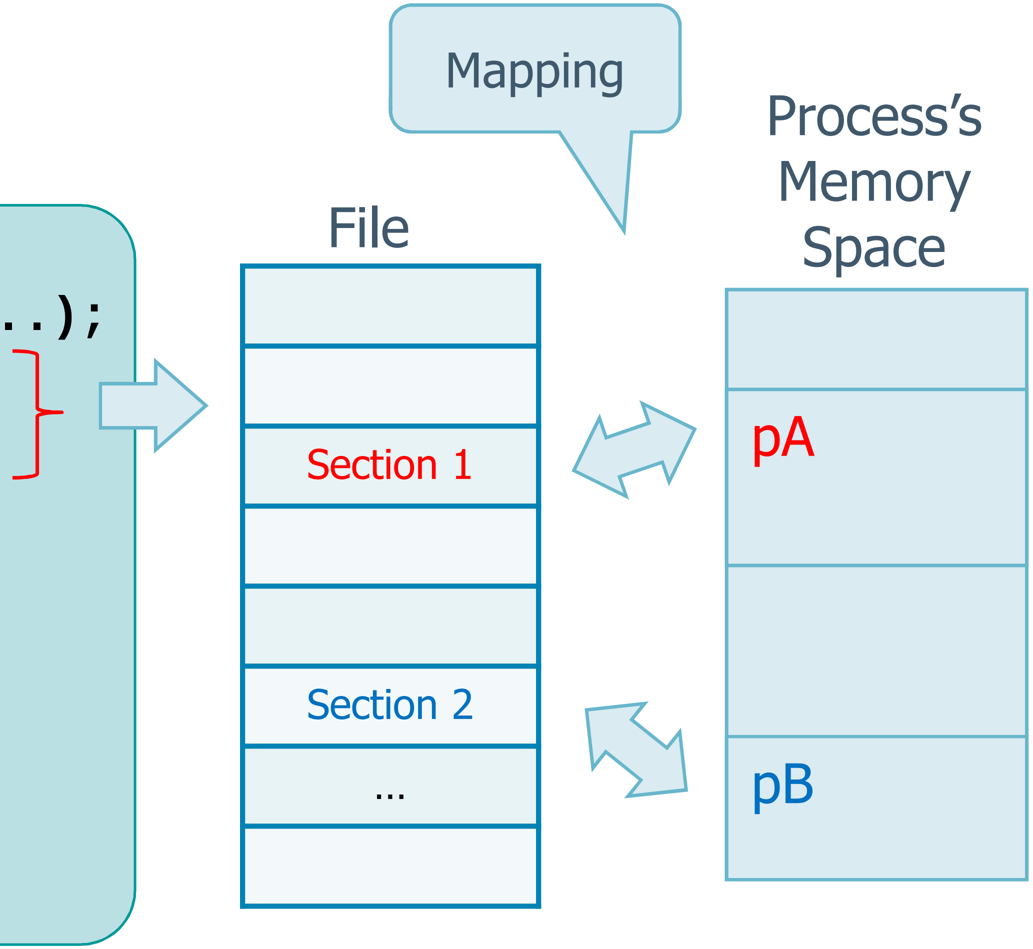
Manage file through main memory

Clean and close

Logic

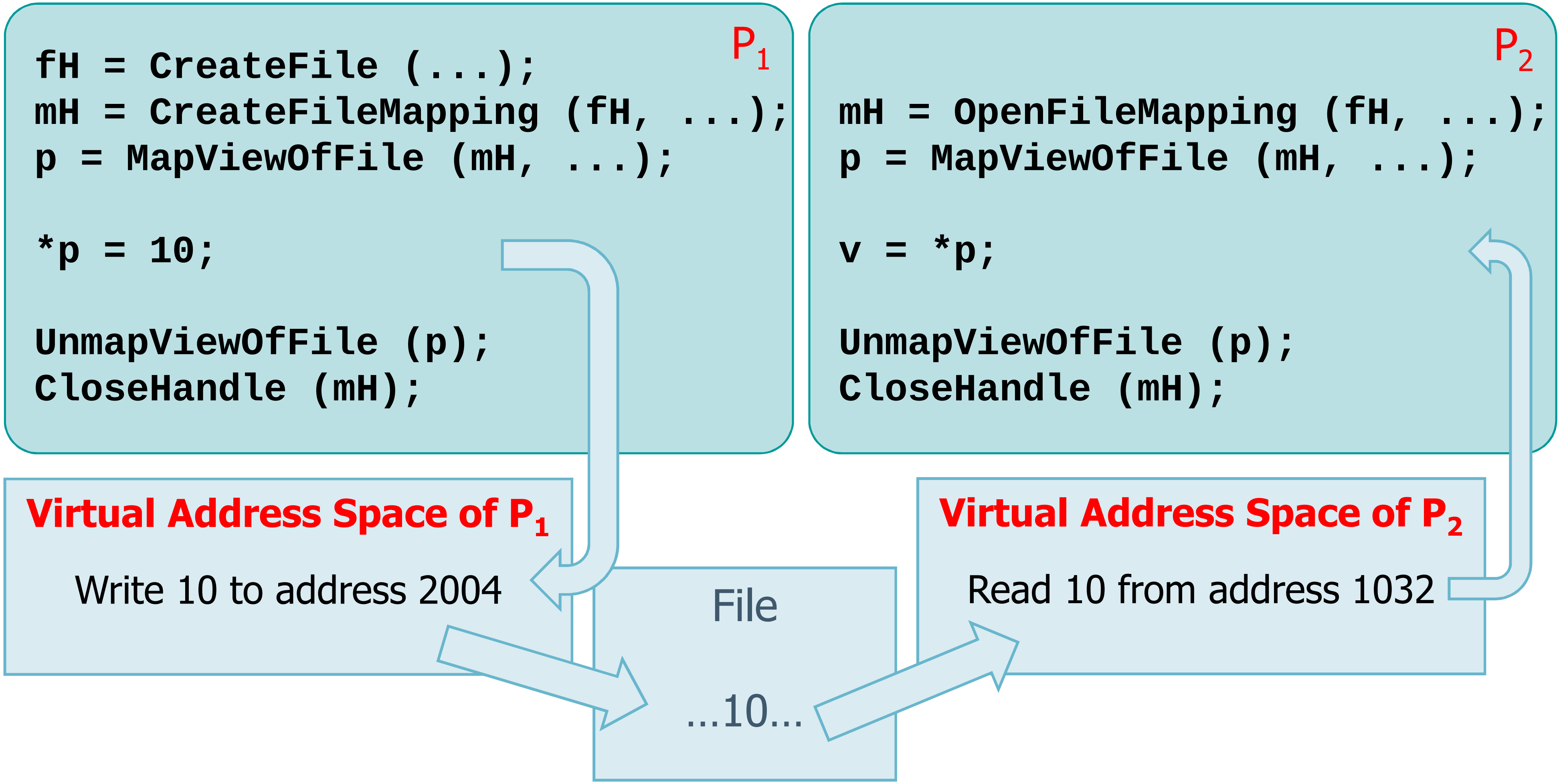
- Memory-mapped file used inside a **single** process

```
fH = CreateFile (...);
mH = CreateFileMapping (fH, ...);
pA = MapViewOfFile (mH, ...);
pB = MapViewOfFile (mH, ...);
while ( ) {
  pB->Data = pA->Data;
  pA++; pB++; ...
}
UnmapViewOfFile (pA);
UnmapViewOfFile (pB);
CloseHandle (mH);
CloseHandle (fH);
```



Logic

- Memory-mapped file used to share data between **two** processes



P_1

```
fH = CreateFile (...);
mH = CreateFileMapping (fH, ...);
p = MapViewOfFile (mH, ...);

*p = 10;

UnmapViewOfFile (p);
CloseHandle (mH);
```

P_2

```
mH = OpenFileMapping (fH, ...);
p = MapViewOfFile (mH, ...);

v = *p;

UnmapViewOfFile (p);
CloseHandle (mH);
```

Virtual Address Space of P_1

Write 10 to address 2004

File

...10...

Virtual Address Space of P_2

Read 10 from address 1032

CreateFileMapping

```
HANDLE CreateFileMapping (
  HANDLE hFile,
  LPSECURITY_ATTRIBUTES lpsa,
  DWORD dwProtect,
  DWORD dwMaximumSizeHigh,
  DWORD dwMaximumSizeLow,
  LPCTSTR lpMapName
);
```

It does not really perform the mapping

- Given a part of a file (eventually an entire file) CreateFileMapping returns a **mapping object**
- Return value
  - A **file mapping handle**, on success
  - NULL, on failure

CreateFileMapping

- Parameters
  - hFile
    - **Handle** of an already opened file
    - The protection flags must be compatible with dwProtect
  - lpsa
    - LPSECURITY_ATTRIBUTES
    - Often NULL

```
HANDLE CreateFileMapping (
  HANDLE hFile,
  LPSECURITY_ATTRIBUTES lpsa,
  DWORD dwProtect,
  DWORD dwMaximumSizeHigh,
  DWORD dwMaximumSizeLow,
  LPCTSTR lpMapName
);
```

CreateFileMapping

- dwProtect
  - How you can access the mapped file
    - PAGE_READONLY
      - Pages in the mapped region are read only
    - PAGE_READWRITE
      - Full access if hFile has both GENERIC_READ and GENERIC_WRITE access
    - PAGE_WRITECOPY
      - When you change mapped memory, a copy is written to the paging file **not** to the original file

```
HANDLE CreateFileMapping (
  HANDLE hFile,
  LPSECURITY_ATTRIBUTES lpsa,
  DWORD dwProtect,
  DWORD dwMaximumSizeHigh,
  DWORD dwMaximumSizeLow,
  LPCTSTR lpMapName
);
```

CreateFileMapping

Two 32bit fields
32 LSBs and 32 MSBs

- dwMaximumSizeHigh and dwMaximumSizeLow
  - Specify the size of the mapping object
  - The value 0 is used to specify the current file size
  - Use 0 (actual file size) if the file in going to be extended
- lpMapName
  - **Names the mapping object**, allowing other processes to share the object
  - Case sensitive
  - Often NULL, but not when used by openFileMapping

```
HANDLE CreateFileMapping (
  HANDLE hFile,
  LPSECURITY_ATTRIBUTES lpsa,
  DWORD dwProtect,
  DWORD dwMaximumSizeHigh,
  DWORD dwMaximumSizeLow,
  LPCTSTR lpMapName
);
```

OpenFileMapping

- It is possible to obtain a file mapping handle for an existing named mapping
- To do that, specify the mapping object's name
  - This name comes from a previous call to **CreateFileMapping**
- Two processes can share memory by sharing a file mapping
  - First, a process creates the named mapping uising **CreateFileMapping**
  - Subsequently, another processes open this mapping with the name using **OpenFileMapping**
    - The open will fail if the named object does not exist

OpenFileMapping

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

- Parameters
  - dwDesiredAccess
    - The access rights to the mapped region
    - See MapViewOfFile for the possible values
  - bInheritHandle
    - If TRUE, specifies whether the handle can be inherited by a sub-process (created with CreateProcess)
    - If FALSE, cannot be inherited

OpenFileMapping

- lpNameP
    - Is that name created by **CreateFileMapping**
- Return value
    - A file mapping handle, on success
    - NULL, on failure

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

MapViewOfFile

- Once a mapping object has been created
  - The next step is to map a file into the process's virtual address space
  - A pointer to the allocated block (or file-view) is returned
    - The main difference from a standard memory allocation operation lies in the fact that the allocated block is backed by a user-specified file rather than the paging file
  - Note
    - The mapping view does not expand if the file size increases
    - Growing files need to be re-mapped

MapViewOfFile

```
LPVOID MapViewOfFile (
  HANDLE hMapObject,
  DWORD dwAccess,
  DWORD dwOffsetHigh,
  DWORD dwOffsetLow,
  SIZE_T dwNumberOfByteToMap
);
```

SIZE_T is either a DWORD (on _WIN32) or a DWORDLONG (on _WIN64) depending on the compiler flag

- Return value
  - The starting address of the block (file view), on success
  - NULL, on failure

MapViewOfFile

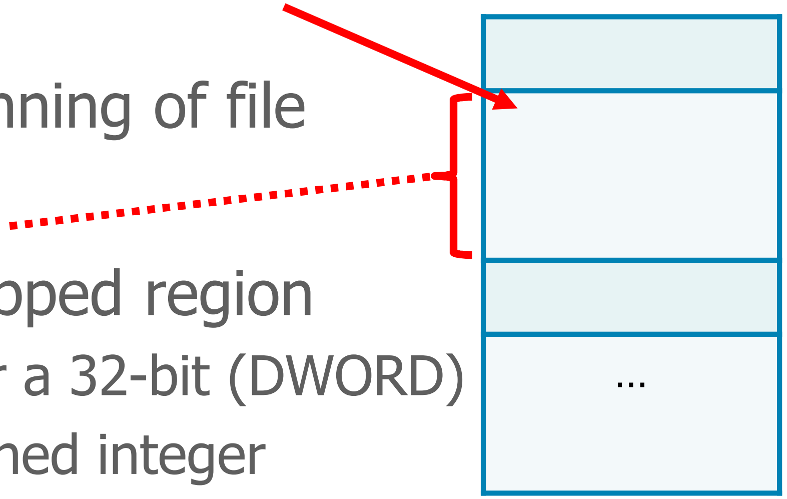
- Parameters
  - hMapObject
    - Identifies a file-mapping object (from **CreateFileMapping** or **OpenFileMapping**)
  - dwAccess
    - Is the file acces rights and must be compatible with the mapping object’s access
      - FILE_MAP_WRITE
      - FILE_MAP_READ
      - FILE_MAP_ALL_ACCESS

        (or of the previous flags)

```
LPVOID MapViewOfFile(
  HANDLE hMapObject,
  DWORD dwAccess,
  DWORD dwOffsetHigh,
  DWORD dwOffsetLow,
  SIZE_T dwNumberOfByteToMap
);
```

MapViewOfFile

- dwOffsetHigh and dwOffsetLow
  - Is the **starting** location of the mapped file region
  - Must be a **multiple of 64K**
  - **Zero** offset to map from beginning of file
- dwNumbrOfByteToMap
  - Is the **size** in bytes of the mapped region
    - SIZE_T is is defined as either a 32-bit (DWORD) 64-bit (DWORDLONG) unsigned integer
      - It is helps to enable source code portability
  - Zero indicates the entire file
    - The map size is limited by the 32-bit address (DWORD) in a 32-bit build



```
LPVOID MapViewOfFile(
  HANDLE hMapObject,
  DWORD dwAccess,
  DWORD dwOffsetHigh,
  DWORD dwOffsetLow,
  SIZE_T dwNumberOfByteToMap
);
```

UnmapViewOfFile

```
BOOL UnmapViewOfFile (
  LPVOID lpBaseAdress
);
```

- Just as it is necessary to release the memory allocated, it is necessary to release file views
  - Use **UnampViewOfFile** to release a file view
- Use **CloseHandle** to finally destroy mapping handles
  - For both OpenFileMapping and CreateFileMapping

File-Mapping Limitations

- Coherency
  - Processes that share a file through shared memory will have a coherent view of the file
    - If one process changes a mapped memory location, the other process will obtain that new value when it accesses the corresponding area of the file in its mapped memory
  - On the other hand, a process accessing a file through mapping and another process accessing it through conventional file I/O will not have coherent views of the file
    - It is not a good idea to access a mapped file with **ReadFile** and **WriteFile** at the same time

File-Mapping Limitations

- Large files
  - With 32-bit operating systems
    - Large files (greater than 4GB) cannot be mapped entirely into virtual memory space
    - When dealing with large files, you must create code that carefully **maps** and **unmaps** file regions as you need them
  - With 64-bit build very large files can be mapped
- An existing file mapping cannot be expanded
  - The maximum size must be known when the mapping is created

Example

- There are several problems in which two or more synchronization primitives have to be used together
- Example
  - Two processes with several threads
  - They want to work on a shared memory
    - They may use a memory mapped file
  - They need to protect their own R/W activity
    - They may use a mutex for the critical section
  - The writer (producer) need a strategy to let the reader (consumer) know when he has done
    - They may use an event

Example

P_i

```
eH=CreateEvent(...);
mH=CreateMutex(...);
fmH=CreateFileMapping(...);
WaitForSingleObject(mH);
ptr=MapViewOfFile(fmH...);
... write to shared memory
SetEvent(eH);
UnmapViewOfFile(fmH);
ReleaseMutex(mH);
CloseHandle(...);
```

Local Mutex (for CS)

Create a new file mapping

Open an existing file mapping

Local Mutex (for CS)

P_j

```
eH=CreateEvent(...);
mH=CreateMutex(...);
fmH=OpenFileMapping(...);
WaitForMultipleObjects(
   [eH,mH],WAIT_ALL,...);
ptr=MapViewOfFile(fmH...);
... Read shared memory
UnmapViewOfFile(fmH);
ReleaseMutex(mH);
CloseHandle(...);
```

Exercise

- Preliminaries
  - An advantage of memory mapping is the ability to use convenient memory-based algorithms to process files
  - **Sorting** data in memory, for instance, is much easier than sorting records in a file
- Specification
  - Write a program to sort a file with fixed-length records
    - Assumes an 8-byte sort key at the start of each record
    - Restrict the progam to deal with fix-size records

Exercise

- Use the C library function **qsort** to sort the file
  - This requires a programmer-defined record comparison function (keyCompare)

- Logic
  - Create the file mapping on a temporary copy of the input file
  - Create a single view of the file
  - Sort the file
  - Print the results to standard output

Solution

```
#include ...
#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 pKey1, LPCTSTR pKey2) {
  return _tcsncmp (pKey1, pKey2, KEY_SIZE);
}
```

Definitions of the record structure in the sort file

Compare two records of generic characters.
The key position and length are global variables

See tchar.h: #define _tcsncpy strncpy

Solution

The file name is the first argument

```
int _tmain (int argc, LPTSTR argv[]) {
  ... Definitions ...

  _stprintf_s (tempFile, MAX_PATH, _T ("%s.tmp"), argv[1]);
  CopyFile (argv[1], tempFile, TRUE);


  hFile = CreateFile (tempFile, GENERIC_READ
    | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);
  if (hFile ==INVALID_HANDLE_VALUE)
    ...

  GetFileSizeEx (hFile, &fileSize);
  fileSize.QuadPart += sizeof(TCHAR);
  if (fileSize.HighPart > 0)
    ... This file is too large to map on a Win32 system ...
```

Copy the **input** file to a **temp** output file that will be sorted. Do not alter the input file.

Open the file (use the temporary copy)

Add space for '\0'

If the file is too large, catch that when it is mapped

Solution

Map file

dwProtect parameter

```
hMap = CreateFileMapping (hFile, NULL, PAGE_READWRITE,
  fileSize.HighPart, fileSize.LowPart, NULL);

if (hMap == NULL) ... error ...
```

dwProtect parameter

```
pFile = MapViewOfFile (hMap, FILE_MAP_ALL_ACCESS, 0, 0, 0);
if (pFile == NULL) ... error ...
```

Map from 0

Entire file

Sort

```
qsort (pFile, (SIZE_T)fileSize.QuadPart / RECSIZE,
  RECSIZE, KeyCompare);
```

#records

Record size

Comparison function

Solution

```
  pTFile = (LPTSTR) pFile;
  pTFile[fileSize.QuadPart/sizeof(TCHAR)] = _T('\0');

  _tprintf (_T("%s"), pFile);

  UnmapViewOfFile (pFile);
  CloseHandle (hMap);
  CloseHandle (hFile);
  DeleteFile (tempFile);

  return (1);
}
```

Add ‘\0’
(if no other ‘\0’ exist)

Add string
termination

Print output file
(as a unique string)

Clean and close