CS 16: Assembly Language Programming for the IBM PC and Compatibles
OBJECTIVES

- Start with Win32 console programming
- Talk about writing a graphical windows application
- Look at dynamic memory allocation
- Manage the x86 memory management
USEFUL QUESTIONS

- How do 32-bit programs handle text input-output?
- How are colors handled in 32-bit console mode?
- How does the Irvine32 link library work?
- How are times and dates handled in MS-Windows?
USEFUL QUESTIONS (2)

- How can I use MS-Windows functions to read and write data files?
- Is it possible to write a graphical Windows application in assembly language?
- How do Protected mode programs translate segments and offsets to physical addresses?
- I’ve heard that virtual memory is good. But why is that so?
Win32 Console Programming

- Background Information
  - Win32 Console Programs
  - API and SDK
  - Windows Data Types
  - Standard Console Handles
- Console Input
- Console Output
- Reading and Writing Files
WIN32 CONSOLE PROGRAMMING (2)

- Console Window Manipulation
- Controlling the Cursor
- Controlling the Text Color
- Time and Date Functions
Run in Protected mode
Emulate MS-DOS
Standard text-based input and output
Linker option: /SUBSYSTEM:CONSOLE
The console input buffer contains a queue of input records, each containing data about an input event
A console screen buffer is a two-dimensional array of character and color data that affects the appearance of text in the console window
Text-oriented (high-level) console functions
- Read character streams from input buffer
- Write character streams to screen buffer
- Redirect input and output

Event-oriented (low-level) console functions
- Retrieve keyboard and mouse events
- Detect user interactions with the console window
- Control window size & position, text colors
API and SDK

- **Microsoft Win32 Application Programming Interface**
  - API: a collection of types, constants, and functions that provide a way to directly manipulate objects through programming

- **Microsoft Platform Software Development Kit**
  - SDK: a collection of tools, libraries, sample code, and documentation that helps programmers create applications
  - Platform: an operating system or a group of closely related operating systems
# Translating Window Data Types

<table>
<thead>
<tr>
<th>Window Type(s)</th>
<th>MASM Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>DWORD</td>
</tr>
<tr>
<td>LONG</td>
<td>SDWORD</td>
</tr>
<tr>
<td>COLORREF, HANDLE, LPARAM, LPCTSTR, LPTSTR, LPVOID, LRESULT, UINT, WNDPROC, WPARAM</td>
<td>DWORD</td>
</tr>
<tr>
<td>BSTR, LPCSTR, LPSTR</td>
<td>PTR BYTE</td>
</tr>
<tr>
<td>WORD</td>
<td>WORD</td>
</tr>
<tr>
<td>LPCRECT</td>
<td>PTR RECT</td>
</tr>
</tbody>
</table>
A handle is an unsigned 32-bit integer

The following MS-Windows constants are predefined to specify the type of handle requested:

- STD_INPUT_HANDLE: standard input
- STD_OUTPUT_HANDLE: standard output
- STD_ERROR_HANDLE: standard error output
GetStdHandle returns a handle to a console stream
Specify the type of handle (see previous slide)
The handle is returned in EAX
Prototype:

```
GetStdHandle PROTO,
    nStdHandle:DWORD           ; handle type
```

Sample call:

```
INVOKE GetStdHandle, STD_OUTPUT_HANDLE
mov myHandle, eax
```
The ReadConsole function provides a convenient way to read text input and put it in a buffer

Prototype:

```plaintext
ReadConsole PROTO,
    handle:DWORD,    ; input handle
    pBuffer:PTR BYTE, ; pointer to buffer
    maxBytes:DWORD,  ; number of chars to read
    pBytesRead:PTR DWORD, ; ptr to num bytes read
    notUsed:DWORD    ; (not used)
```
Here’s how to input single characters:

- Get a copy of the current console flags by calling `GetConsoleMode`
- Save the flags in a variable
- Change the console flags by calling `SetConsoleMode`
- Input a character by calling `ReadConsole`
- Restore the previous values of the console flags by calling `SetConsoleMode`
From the ReadChar procedure in the Irvine32 library

```
.data
consoleInHandle  DWORD ?,
saveFlags    DWORD ?, ; backup copy of flags

.code
; Get & save the current console input mode flags
INVOKE GetConsoleMode, consoleInHandle, ADDR saveFlags

; Clear all console flags
INVOKE SetConsoleMode, consoleInHandle, 0
```
More from the ReadChar procedure in the Irvine32 library

; Read a single character from input
INVOKE ReadConsole,
   consoleInHandle, ; console input handle
   ADDR buffer, ; pointer to buffer
   1, ; max characters to read
   ADDR bytesRead, ; return num bytes read
   0 ; not used

; Restore the previous flags state
INVOKE SetConsoleMode, consoleInHandle, saveFlags
The COORD structure specifies X and Y screen coordinates in character measurements, which default to 0-79 and 0-24.

The SMALL_RECT structure specifies a window’s location in character measurements.

COORD STRUCT
  X WORD ?
  Y WORD ?
COORD ENDS

SMALL_RECT STRUCT
  Left  WORD ?
  Top   WORD ?
  Right WORD ?
  Bottom WORD ?
SMALL_RECT ENDS
The WriteConsole function writes a string to the screen, using the console output handle.

It acts upon standard ASCII control characters such as tab, carriage return, and line feed.

Prototype:

```plaintext
WriteConsole PROTO,
    handle:DWORD,          ; output handle
    pBuffer:PTR BYTE,      ; pointer to buffer
    bufsize:DWORD,         ; size of buffer
    pCount:PTR DWORD,      ; output count
    lpReserved:DWORD       ; (not used)
```
The `WriteConsoleOutputCharacter` function copies an array of characters to consecutive cells of the console screen buffer, beginning at a specified location.

**Prototype:**

```
WriteConsoleOutputCharacter PROTO,
    handleScreenBuf:DWORD, ; console output handle
    pBuffer:PTR BYTE,    ; pointer to buffer
    bufsize:DWORD,       ; size of buffer
    xyPos:COORD,         ; first cell coordinates
    pCount:PTR DWORD     ; output count
```
Win32 API Functions that create, read, and write to files:
- CreateFile
- ReadFile
- WriteFile
- SetFilePointer
CreateFile either creates a new file or opens an existing file
If successful, it returns a handle to the open file
Otherwise, it returns a special constant named INVALID_HANDLE_VALUE
Prototype:

```
CreateFile PROTO,
    pFilename:PTR BYTE,           ; ptr to filename
    desiredAccess:DWORD,          ; access mode
    shareMode:DWORD,             ; share mode
    lpSecurity:DWORD,            ; ptr to security attribs
    creationDisposition:DWORD,   ; file creation options
    flagsAndAttributes:DWORD,    ; file attributes
    htemplate:DWORD              ; handle to template file
```
Open an existing file for reading

```
INVOKE CreateFile,
    ADDR filename, ; ptr to filename
    GENERIC_READ, ; access mode
    DO_NOT_SHARE, ; share mode
    NULL, ; ptr to security attributes
    OPEN_EXISTING, ; file creation options
    FILE_ATTRIBUTE_NORMAL, ; file attributes
    0 ; handle to template file
```
Open an existing file for writing

```
INVOKE CreateFile,
    ADDR filename,
    GENERIC_WRITE,  ; access mode
    DO_NOT_SHARE,
    NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    0
```
Creates a new file with normal attributes, erasing any existing file by the same name

```
INVOKES CreateFile,
    ADDR filename,
    GENERIC_WRITE,
    DO_NOT_SHARE,
    NULL,
    CREATE_ALWAYS,  ; overwrite existing file
    FILE_ATTRIBUTE_NORMAL,
    0
```
ReadFile reads text from an input file

Prototype:

```
ReadFile PROTO,
    handle:DWORD, ; handle to file
    pBuffer:PTR BYTE, ; ptr to buffer
    nBufsize:DWORD, ; num bytes to read
    pBytesRead:PTR DWORD, ; bytes actually read
    pOverlapped:PTR DWORD ; ptr to async info
```
WriteFile writes data to a file, using an output handle.
The handle can be the screen buffer handle, or it can be one assigned to a text file.

Prototype:

```
WriteFile PROTO,
    fileHandle:DWORD,                ; output handle
    pBuffer:PTR BYTE,                ; pointer to buffer
    nBufsize:DWORD,                  ; size of buffer
    pBytesWritten:PTR DWORD,         ; num bytes written
    pOverlapped:PTR DWORD             ; ptr to asynch info
```
SetFilePointer moves the position pointer of an open file
You can use it to append data to a file, and to perform random-access record processing

```
SetFilePointer PROTO,
    handle:DWORD,       ; file handle
    nDistanceLo:SDWORD, ; bytes to move pointer
    pDistanceHi:PTR SDWORD, ; ptr to bytes to move
    moveMethod:DWORD    ; starting point
```

Example

```
; Move to end of file:

INVOKE SetFilePointer,
    fileHandle,0,0,FILE_END
```
Input and output handles are 64 bits
Before calling a system function, reserve at least 32 bytes of shadow space by subtracting from the stack pointer (RSP).
Restore RSP after the system call
Pass integers in 64-bit registers
First four arguments should be placed in RCX, RDX, R8, and R9 registers
64-bit integer values are returned in RAX
.data
STD_OUTPUT_HANDLE EQU -11
consoleOutHandle QWORD ?

.code
sub    rsp,40 ; reserve shadow space & align RSP
mov    rcx,STD_OUTPUT_HANDLE
call   GetStdHandle
mov    consoleOutHandle,rax
add    rsp,40
EXAMPLE: CALLING WRITECONSOLE

sub rsp, 5 * 8 ; space for 5 parameters
movr cx,rdx
call Str_length ; returns length of string in EAX
mov rcx,consoleOutHandle
mov rdx,rdx ; string pointer
mov r8, rax ; length of string
lea r9,bytesWritten
mov qword ptr [rsp + 4 * SIZEOF QWORD],0 ; (always zero)
call WriteConsoleA
CONSOLE WINDOW MANIPULATION

- Screen buffer
- Console window
- Controlling the cursor
- Controlling the text color
The active screen buffer includes data displayed by the console window
SetConsoleTitle changes the console window's title
Pass it a null-terminated string

```assembly
.data
titleStr BYTE "Console title",0
.code
INVOKED SetConsoleTitle, ADDR titleStr
```
GetConsoleScreenBufferInfo returns information about the current state of the console window.

It has two parameters:
- A handle to the console screen
- A pointer to a structure that is filled in by the function

```assembly
.data
outHandle DWORD ?
consoleInfo CONSOLE_SCREEN_BUFFER_INFO <>
.code
    INVOKE GetConsoleScreenBufferInfo, outHandle, ADDR consoleInfo
```
- `dwSize`: size of the screen buffer (char columns and rows)
- `dwCursorPos`: cursor location
- `wAttributes`: colors of characters in console buffer
- `srWindow`: coords of console window relative to screen buffer
- `maxWinSize`: maximum size of the console window
SetConsoleWindowInfo lets you set the size and position of the console window relative to its screen buffer.

Prototype:

```c
SetConsoleWindowInfo PROTO,
    nStdHandle:DWORD,          ; screen buffer handle
    bAbsolute:DWORD,           ; coordinate type
    pConsoleRect:PTR SMALL_RECT ; window rectangle
```
SetConsoleScreenBufferSize lets you set the screen buffer size to X columns by Y rows

Prototype:

```
SetConsoleScreenBufferSize PROTO,
    outHandle:DWORD, ; handle to screen buffer
dwSize:COORD    ; new screen buffer size
```
CONTROLLING THE CURSOR

- GetConsoleCursorInfo
  - Returns the size and visibility of the console cursor
- SetConsoleCursorInfo
  - Sets the size and visibility of the cursor
- SetConsoleCursorPosition
  - Sets the X, Y position of the cursor
CONSOLE_CURSOR_INFO

- Structure containing information about the console’s cursor size and visibility

```assembly
CONSOLE_CURSOR_INFO STRUCT
    dwSize DWORD ?
    bVisible DWORD ?
CONSOLE_CURSOR_INFO ENDS
```
Sets the foreground and background colors of all subsequent text written to the console

Prototype:

```c
SetConsoleTextAttribute PROTO,
    outHandle:DWORD, ; console output handle
    nColor:DWORD    ; color attribute
```
Copies an array of attribute values to consecutive cells of the console screen buffer, beginning at a specified location

Prototype:

```c
WriteConsoleOutputAttribute PROTO,
    outHandle:DWORD,          ; output handle
    pAttribute:PTR WORD,      ; write attributes
    nLength:DWORD,            ; number of cells
    xyCoord:COORD,            ; first cell coordinates
    lpCount:PTR DWORD         ; number of cells written
```
WriteColors Program

- Creates an array of characters and an array of attributes, one for each character
- Copies the attributes to the screen buffer
- Copies the characters to the same screen buffer cells as the attributes
- Sample output (starts in row 2, column 10):
; Writing Text Colors (WriteColors.asm)
; Demonstration of WriteConsoleOutputCharacter, 
; and WriteConsoleOutputAttribute functions.
INCLUDE Irvine32.inc
.data
outHandle HANDLE ?
cellsWritten DWORD ?
xyPos COORD <10,2>
; Array of character codes:
buffer BYTE 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
BYTE 16,17,18,19,20
BufSize DWORD ($ - buffer)
; Array of attributes:
attributes WORD 0Fh,0Eh,0Dh,0Ch,0Bh,0Ah,9,8,7,6
WORD 5,4,3,2,1,0F0h,0E0h,0D0h,0C0h,0B0h
WriteColors Assembly Code (2)

.code
main PROC
; Get the Console standard output handle:
    INVOKE GetStdHandle,STD_OUTPUT_HANDLE
    mov outHandle,eax
; Set the colors from (10,2) to (30,2):
    INVOKE WriteConsoleOutputAttribute, outHandle, ADDR attributes, BufSize, xyPos, ADDR cellsWritten
; Write character codes 1 to 20:
    INVOKE WriteConsoleOutputCharacter, outHandle, ADDR buffer, BufSize, xyPos, ADDR cellsWritten
    call ReadChar
    exit
main ENDP
END main
TIME AND DATE FUNCTIONS

- GetLocalTime, SetLocalTime
- GetTickCount, Sleep
- GetDateTime
- SYSTEMTIME Structure
- Creating a Stopwatch Timer
GETLOCALTIME, SETLOCALTIME

- GetLocalTime returns the date and current time of day, according to the system clock

  ```
  GetLocalTime PROTO,
  pSystemTime:PTR SYSTEMTIME
  ```

- SetLocalTime sets the system’s local date and time

  ```
  SetLocalTime PROTO,
  pSystemTime:PTR SYSTEMTIME
  ```
GetTickCount function returns the number of milliseconds that have elapsed since the system was started.

```
GetTickCount PROTO ; return value in EAX
```

Sleep pauses the current program for a specified number of milliseconds.

```
Sleep PROTO,
    dwMilliseconds:DWORD
```
The GetDateTime procedure in the Irvine32 library calculates the number of 100-nanosecond time intervals that have elapsed since January 1, 1601.

```
GetDateTime PROC,
    pStartTime:PTR QWORD
```

Pass it a pointer to an empty 64-bit FILETIME structure, which is then filled in by the procedure.

```
FILETIME STRUCT
    loDateTime DWORD ?
    hiDateTime DWORD ?
FILETIME ENDS
```
SYSTEMTIME STRUCT

wYear WORD ? ; year (4 digits)
wMonth WORD ? ; month (1-12)
wDayOfWeek WORD ? ; day of week (0-6)
wDay WORD ? ; day (1-31)
wHour WORD ? ; hours (0-23)
wMinute WORD ? ; minutes (0-59)
wSecond WORD ? ; seconds (0-59)
wMilliseconds WORD ? ; milliseconds (0-999)

SYSTEMTIME ENDS
The Timer.asm program demonstrates a simple stopwatch timer

It has two important functions:
- TimerStart: receives a pointer to a doubleword, into which it saves the current time
- TimerStop: receives a pointer to the same doubleword, and returns the difference (in milliseconds) between the current time and the previously recorded time

Calls the Win32 GetTickCount function
STOPWATCH TIMER ASSEMBLY CODE

; Calculate Elapsed Time               (Timer.asm)
; Demonstrate a simple stopwatch timer, using the Win32 GetTickCount function.
INCLUDE Irvine32.inc
INCLUDE macros.inc
.data
startTime DWORD ?
.code
main PROC
    INVOKE GetTickCount           ; get starting tick count
    mov startTime,eax            ; save it
; Create a useless calculation loop.
    mov    ecx,10000100h
    L1:    imul    ebx
    imul    ebx
    imul    ebx
    imul    ebx
    loop    L1
INVOKE GetTickCount ; get new tick count
cmp eax,startTime ; lower than starting one?
jb error ; it wrapped around
sub eax,startTime ; get elapsed milliseconds
call WriteDec ; display it
mWrite "milliseconds have elapsed",0dh,0ah>
jmp quit

error:
mWrite "Error: GetTickCount invalid--system has "
mWrite "been active for more than 49.7 days",0dh,0ah>

quit:
exit
main ENDP
END main
WRITING A GRAPHICAL WINDOWS APPLICATION

- Required Files
- POINT, RECT Structures
- MSGStruct, WNDCLASS Structures
- MessageBox Function
- WinMain, WinProc Procedures
- ErrorHandler Procedure
- Message Loop & Processing Messages
- Program Listing
Required Files

- **make32.bat**: batch file specifically for building this program
- **WinApp.asm**: program source code
- **GraphWin.inc**: include file containing structures, constants, and function prototypes used by the program
- **kernel32.lib**: same MS-Windows API library used earlier in this chapter
- **user32.lib**: additional MS-Windows API functions
POINT AND RECT STRUCTURES

- **POINT**: X, Y screen coordinates
  ```plaintext
  POINT STRUCT
  ptX DWORD ?
  ptY DWORD ?
  POINT ENDS
  ```

- **RECT**: Holds the graphical coordinates of two opposing corners of a rectangle
  ```plaintext
  RECT STRUCT
  left DWORD ?
  top DWORD ?
  right DWORD ?
  bottom DWORD ?
  RECT ENDS
  ```
MSGStruct: holds data for MS-Windows messages (usually passed by the system and received by your application)

```
MSGStruct STRUCT
    msgWnd DWORD ?
    msgMessage DWORD ?
    msgWparam DWORD ?
    msgLparam DWORD ?
    msgTime DWORD ?
    msgPt POINT <>
MSGStruct ENDS
```
Each window in a program belongs to a class, and each program defines a window class for its main window.

**WNDCLASS STRUCT**

- `style` DWORD ? ; window style options
- `lpfnWndProc` DWORD ? ; WinProc function pointer
- `cbClsExtra` DWORD ? ; shared memory
- `cbWndExtra` DWORD ? ; number of extra bytes
- `hInstance` DWORD ? ; handle to current program
- `hIcon` DWORD ? ; handle to icon
- `hCursor` DWORD ? ; handle to cursor
- `hbrBackground` DWORD ? ; handle to background brush
- `lpszMenuName` DWORD ? ; pointer to menu name
- `lpszClassName` DWORD ? ; pointer to WinClass name

WNDCLASS ENDS
WNDCLASS STRUCTURE (2)

- style is a conglomerate of different style options, such as WS_CAPTION and WS_BORDER, that control the window’s appearance and behavior.
- lpfnWndProc is a pointer to a function (in our program) that receives and processes event messages triggered by the user.
- cbClsExtra refers to shared memory used by all windows belonging to the class.
  - Can be null.
- cbWndExtra specifies the number of extra bytes to allocate following the window instance.
**WNDCLASS STRUCTURE (3)**

- `hInstance` holds a handle to the current program instance
- `hIcon` and `hCursor` hold handles to icon and cursor resources for the current program
- `hbrBackground` holds a background (color) brush
- `lpszMenuName` points to a menu string
- `lpszClassName` points to a null-terminated string containing the window’s class name
Displays text in a box that pops up and waits for the user to click on a button

```
MessageBox PROTO,
    hWnd:DWORD,
    pText:PTR BYTE,
    pCaption:PTR BYTE,
    style:DWORD
```

- hWnd is a handle to the current window
- pText points to a null-terminated string that will appear inside the box
- pCaption points to a null-terminated string that will appear in the box’s caption bar
- style is an integer that describes both the dialog box’s icon (optional) and the buttons (required)
**MESSAGEBOX EXAMPLE**

- Displays a message box that shows a question, including an OK button and a question-mark icon

```
.data
hMainWnd DWORD ?
QuestionText BYTE "Register this program now?"
QuestionTitle BYTE "Trial Period Has Expired"

.code
INVOKE MessageBox,
    hMainWnd,
    ADDR QuestionText,
    ADDR QuestionTitle,
    MB_OK + MB_ICONQUESTION
```
Every Windows application needs a startup procedure, usually named `WinMain`, which is responsible for the following tasks:

- Get a handle to the current program
- Load the program’s icon and mouse cursor
- Register the program’s main window class and identify the procedure that will process event messages for the window
- Create the main window
- Show and update the main window
- Begin a loop that receives and dispatches messages
WinProc receives and processes all event messages relating to a window
- Some events are initiated by clicking and dragging the mouse, pressing keyboard keys, and so on

WinProc decodes each message, carries out application-oriented tasks related to the message

WinProc PROC,
    hWnd:DWORD, ; handle to the window
    localMsg:DWORD, ; message ID
    wParam:DWORD, ; parameter 1 (varies)
    lParam:DWORD ; parameter 2 (varies)

- Contents of wParam and lParam vary, depending on the message
In the example program from this chapter, the WinProc procedure handles three specific messages:

- WM_LBUTTONDOWN, generated when the user presses the left mouse button
- WM_CREATE, indicates that the main window was just created
- WM_CLOSE, indicates that the application’s main window is about to close
The ErrorHandler procedure has several important tasks to perform:

- Call GetLastError to retrieve the system error number
- Call FormatMessage to retrieve the appropriate system-formatted error message string
- Call MessageBox to display a popup message box containing the error message string
- Call LocalFree to free the memory used by the error message string
INVOKE GetLastError ; Returns message ID in EAX
mov messageID,eax

; Get the corresponding message string.
INVOKE FormatMessage, FORMAT_MESSAGE_ALLOCATE_BUFFER + \
  FORMAT_MESSAGE_FROM_SYSTEM, NULL, messageID, NULL,
  ADDR pErrorMsg, NULL, NULL

; Display the error message.
INVOKE MessageBox, NULL, pErrorMsg, ADDR ErrorTitle,
  MB_ICONERROR + MB_OK

; Free the error message string.
INVOKE LocalFree, pErrorMsg
In WinMain, the message loop receives and dispatches (relays) messages

Message_Loop:
    ; Get next message from the queue.
    INVOKE GetMessage, ADDR msg, NULL,NULL,NULL

    ; Quit if no more messages.
    .IF eax == 0
        jmp Exit_Program
    .ENDIF

    ; Relay the message to the program's WinProc.
    INVOKE DispatchMessage, ADDR msg

    jmp Message_Loop
WinProc receives each message and decides what to do with it.

```asm
WinProc PROC, hWnd:DWORD, localMsg:DWORD,
    wParam:DWORD, lParam:DWORD

    mov eax, localMsg
    .IF eax == WM_LBUTTONDOWN ; mouse button?
        INVOKE MessageBox, hWnd, ADDR PopupText,
            ADDR PopupTitle, MB_OK
            jmp WinProcExit
    .ELSEIF eax == WM_CREATE ; create window?
        INVOKE MessageBox, hWnd, ADDR AppLoadMsgText,
            ADDR AppLoadMsgTitle, MB_OK
            jmp WinProcExit
    (etc.)
```

DYNAMIC MEMORY ALLOCATION

- Reserving memory at runtime for objects
  - aka heap allocation
  - Standard in high-level languages (C++, Java)

- Heap manager
  - Allocates large blocks of memory
  - Maintains free list of pointers to smaller blocks
  - Manages requests by programs for storage
# Windows Heap-Related Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetProcessHeap</td>
<td>Returns a 32-bit integer handle to the program’s existing heap area in EAX. If the function succeeds, it returns a handle to the heap in EAX. If it fails, the return value in EAX is NULL.</td>
</tr>
<tr>
<td>HeapAlloc</td>
<td>Allocates a block of memory from a heap. If it succeeds, the return value in EAX contains the address of the memory block. If it fails, the returned value in EAX is NULL.</td>
</tr>
<tr>
<td>HeapCreate</td>
<td>Creates a new heap and makes it available to the calling program. If the function succeeds, it returns a handle to the newly created heap in EAX. If it fails, the return value in EAX is NULL.</td>
</tr>
<tr>
<td>HeapDestroy</td>
<td>Destroys the specified heap object and invalidates its handle. If the function succeeds, the return value in EAX is nonzero.</td>
</tr>
<tr>
<td>HeapFree</td>
<td>Frees a block of memory previously allocated from a heap, identified by its address and heap handle. If the block is freed successfully, the return value is nonzero.</td>
</tr>
<tr>
<td>HeapReAlloc</td>
<td>Reallocates and resize a block of memory from a heap. If the function succeeds, the return value is a pointer to the reallocated memory block. If the function fails and you have not specified HEAP_GENERATE_EXCEPTIONS, the return value is NULL.</td>
</tr>
<tr>
<td>HeapSize</td>
<td>Returns the size of a memory block previously allocated by a call to HeapAlloc or HeapReAlloc. If the function succeeds, EAX contains the size of the allocated memory block, in bytes. If the function fails, the return value is SIZE_T – 1. (SIZE_T equals the maximum number of bytes to which a pointer can point.)</td>
</tr>
</tbody>
</table>
Get a handle to the program’s existing heap

```
.data
hHeap HANDLE ?

.code
INVOKED GetProcessHeap
.INVOKE GetProcessHeap
.IF eax == NULL ; cannot get handle
       jmp quit
.ELSE
       mov hHeap,eax ; handle is OK
.ENDIF
```
sample code (2)

- Allocate a block of memory from the existing heap

.data
hHeap HANDLE ? ; heap handle
pArray DWORD ? ; pointer to array

.code
INVOKE HeapAlloc, hHeap, HEAP_ZERO_MEMORY, 1000
 IF eax == NULL
   mWrite "HeapAlloc failed"
   jmp quit
 ELSE
   mov pArray, eax
 .ENDIF
Free a block of memory previously allocated by calling HeapAlloc

.data
hHeap HANDLE ? ; heap handle
pArray DWORD ? ; pointer to array

.code
INVoke HeapFree,
    hHeap, ; handle to heap
    0, ; flags
    pArray ; pointer to array
X86 Memory Management

- Reviewing Some Terms
- New Terms
- Translating Addresses
- Converting Logical to Linear Address
- Page Translation
Multitasking permits multiple programs (or tasks) to run at the same time
- The processor divides up its time between all of the running programs
- Segments are variable-sized areas of memory used by a program containing either code or data
Segmentation provides a way to isolate memory segments from each other
  - This permits multiple programs to run simultaneously without interfering with each other

A segment descriptor is a 64-bit value that identifies and describes a single memory segment
  - Contains information about the segment’s base address, access rights, size limit, type, and usage
A segment selector is a 16-bit value stored in a segment register (CS, DS, SS, ES, FS, or GS)
- Provides an indirect reference to a memory segment

A logical address is a combination of a segment selector and a 32-bit offset
The x86 processor uses a one- or two-step process to convert a variable's logical address into a unique memory location.

- The first step combines a segment value with a variable’s offset to create a linear address.
- The second optional step, called page translation, converts a linear address to a physical address.
The segment selector points to a segment descriptor, which contains the base address of a memory segment.

The 32-bit offset from the logical address is added to the segment’s base address, generating a 32-bit linear address.
INDEXING INTO A DESCRIPTOR TABLE

- Each segment descriptor indexes into the program's local descriptor table (LDT)
- Each table entry is mapped to a linear address
Paging makes it possible for a computer to run a combination of programs that would not otherwise fit into memory.

Only part of a program must be kept in memory, while the remaining parts are kept on disk.

The memory used by the program is divided into small units called pages.

As the program runs, the processor selectively unloads inactive pages from memory and loads other pages that are immediately required.
OS maintains page directory and page tables

PAGE TRANSLATION: CPU converts the linear address into a physical address

PAGE FAULT: occurs when a needed page is not in memory, and the CPU interrupts the program

OS copies the page into memory, program resumes execution
A linear address is divided into a page directory field, page table field, and page frame offset.

The CPU uses all three to calculate the physical address.
32-bit console programs
- Read from the keyboard and write plain text to the console window using Win32 API functions

Important functions
- ReadConsole, WriteConsole, GetStdHandle, ReadFile, WriteFile, CreateFile, CloseHandle, SetFilePointer

Dynamic memory allocation
- HeapAlloc, HeapFree

x86 Memory management
- Segment selectors, linear address, physical address
- Segment descriptor tables
- Paging, page directory, page tables, page translation