Operating Systems & Multitasking (Finishing §2.1)

—

Executables, DLLs, and Loading

—

§2.2 x86 Architecture Details (Part 1)

Operating Systems & Multitasking

(finishing §2.1)

Operating Systems (Review from Last Time)

- An *operating system (OS)*
  - Manages hardware resources
  - Provides services to application programs, e.g.,
    - Input/output – remember what *device drivers* do? (assigned reading, §1.1)
    - Read/write/delete files
    - Allocate memory
    - Run another program
  - The operating system's *application programming interface (API)* provides functions that you can call from C/C++ or assembly language programs to perform these tasks. Example from the Win32 API: `WriteConsole` (link)
- The part of the OS the user interacts with (e.g., to start programs) is the *shell*

Multitasking

- A running program is called a *process*
  - You can run several copies of the same program at the same time; each is a separate process
  - Processes *cannot* access each other's memory
- Each process can have multiple *threads,* i.e., it can execute several procedures simultaneously
  - *Example: in your Web browser, the rendering engine (which determines how to display a document) runs in one thread, while network communication is performed in separate threads, so the document can be displayed at the same time more data is being loaded
  - A process's memory is *shared* among all its threads; each thread does *not* have its own memory
- A *task* is either a process or a thread
- A *multitasking* operating system can run multiple tasks at the same time

*Source: http://taligarsiel.com/Projects/howbrowserswork1.htm#The_rendering_engines_threads

Multitasking

- **Q:** Suppose a processor has only one CPU. How can an operating system run multiple tasks simultaneously?
- **A:** Run one program for a few milliseconds, then run another for a few milliseconds, then another, etc.
- A part of the OS called the *scheduler* allocates a *time slice* to each task
  - E.g., as of 2007, Windows used time slices of 3 to 180 milliseconds*
  - Time slices are small enough to give the illusion that tasks are running simultaneously
  - An OS that does this is called a *preemptive multitasking* OS (next slide)

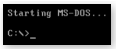
*Source: http://support.microsoft.com/kb/259025

Multitasking

- *Preemptive multitasking:* the OS switches to the next task whenever the time slice expires; higher-priority tasks can interrupt lower-priority ones
  - Used in Windows 95 and later, Linux/Android, iOS
- *Cooperative multitasking:* tasks must voluntarily give up control before a task switch occurs
  - Used in Windows 3.1 and earlier
  - One application could cause the entire system to hang
- MS-DOS (Microsoft Disk Operating System) was *not* multitasking: only one application ran at a time (and had full control of the hardware)

Microsoft Operating Systems

- **MS-DOS** - Microsoft Disk Operating System
  - Main OS for PCs during the 1980s and early 1990s
  - Command line interface



- **Microsoft Windows**
  - First commercial success was Windows 3.0 (1990)
  - Windows 95 (1995) switched from cooperative to preemptive multitasking

Image Sources: http://en.wikipedia.org/wiki/File:StartingMsdos.svg and http://en.wikipedia.org/wiki/File:Windows_3.0_logo.svg

Microsoft Operating Systems

- **Microsoft Windows 1.0**

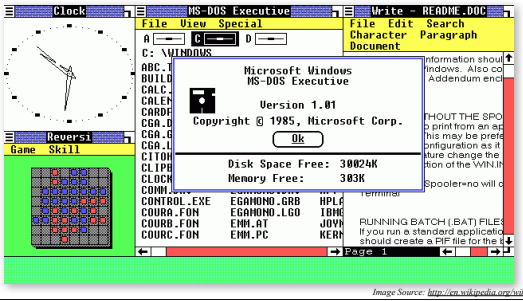


Image Source: http://en.wikipedia.org/wiki/File:Windows1.0.png

Microsoft Operating Systems

- **Microsoft Windows 3.0**

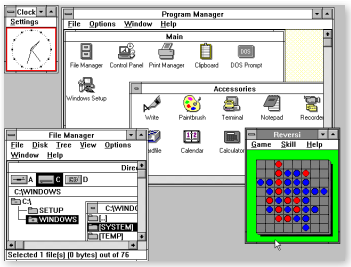


Image Source: http://en.wikipedia.org/wiki/File:Windows1.0.png

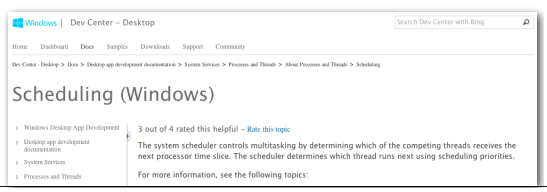
Section 2.1.5, Question 11

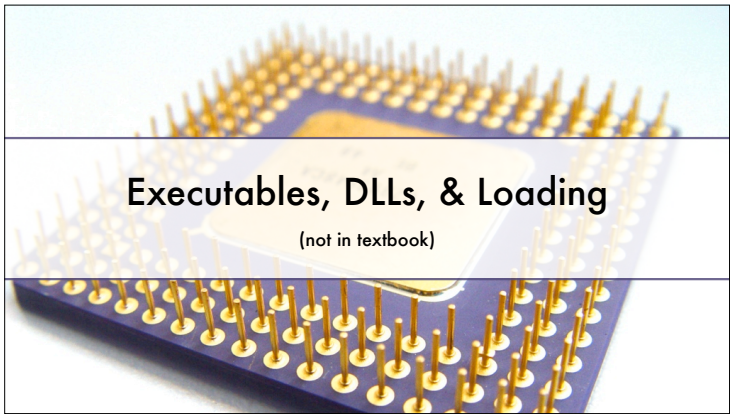
Multitasking

- Before switching to another task, the OS saves the *state* of the current task
- A task's *state* consists of:
  - Values stored in registers
  - Instruction pointer (program counter)
  - Status flags (indicate whether carry, overflow, etc. occurred)
  - Memory segments (what parts of memory are used to store code, data, etc.)
- This is *everything* needed to continue executing the task where it left off
  - The OS copies a tasks's state to memory (*saves* it)
  - Then it runs another task for a while
  - Then it *restore* the original tasks's state
    (sets the register values, instruction pointer, etc. from the values saved in memory)

Windows' Scheduling

- Extensive information on how Windows schedules tasks is available from the Windows Dev Center
  - http://msdn.microsoft.com/en-us/library/windows/desktop/ms685096(v=vs.85).aspx





Executables, DLLs, & Loading

(not in textbook)

Executables and DLLs

- EXE Files - Windows executable programs
  - Executables are runnable programs
    (e.g., type C:\WINDOWS\NOTEPAD.EXE in the Run dialog)
  - Large programs contain lots of procedures (functions);
    one procedure (often called `main`) is the *program entrypoint*
  - Procedures are assembled/compiled to machine language
  - This machine language code can be stored in the .exe file
  - It can also be stored in a *dynamic link library* (DLL) that is loaded *when the program runs*
    - Different from a *static library* or *statically-linked library*

Executables and DLLs

- DLLs - Dynamic link libraries
  - Contain machine language code for many procedures
  - But DLLs are not runnable like executables
  - If a DLL is used by many different processes, the OS can load a single copy, and it can be shared by the processes
    - E.g., the WriteConsole function (provided by Windows) is stored in C:\WINDOWS\SYSTEM32\KERNEL32.DLL
    - E.g., COMDLG32.DLL provides common dialog boxes (Open, Save, Print, ...)
    - Windows only keeps one copy of these DLLs in memory

What's In an Executable?

- Windows' .exe files are stored in the *Portable Executable (PE)* file format
- PE files contain, among other things:
  - A **text** section (machine language code)
  - A **data** section (strings, global variables, etc.)
  - An **idata** section
    - Describes what DLLs need to be loaded at runtime
    - Describes what procedures in that DLL are called by this program

Excerpt of bytes from EXEVIEW.EXE:
(idata section)

This program will call GetOpenFileNameA...

```
...A8 0000 1AA8 0000  ................
...A8 0000 0000 0000  (...<...L.......
...46 696C 654E 616D  ..GetOpenFileNam
...67 3332 2E64 6C6C  eA..comdlg32.dll
...5 466F 6E74 496E  ..%.CreateFontIn
...744 4933 322E 646C  directA.GDI32.dl
...576 6963 6543 6170  l...GetDeviceCap
...46F 636B 4F62 6A65  s...GetStockObje
...454 6578 744D 6574  ct....GetTextMet
...365 6C65 6374 4F62  ricsA...SelectOb
...365 7442 6B43 6F6C  ject....SetBkCol
...454 6578 7443 6F6C  or..5.SetTextCol
...874 4F75 7441 0000  or..E.TextOutA..
```

...which is a function provided by comdlg32.dll

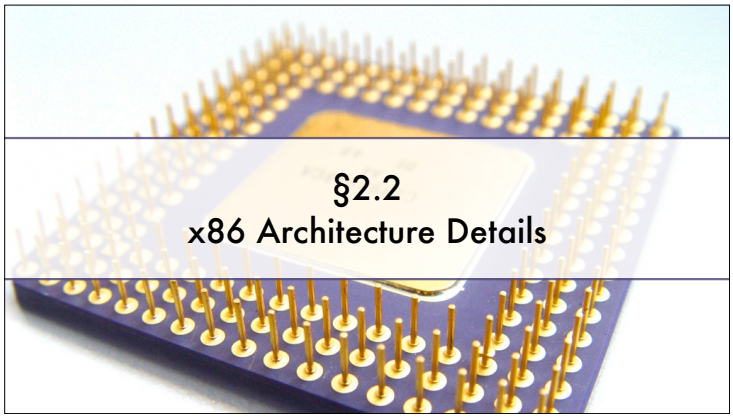
Image Source: http://www.csn.ul.ie/~caolan/publink/winresdump/winresdump/doc/pefile2.html

Loading

- The *loader* is the component of the OS that is responsible for running programs
- So, among other things, Windows' loader:
  1. Allocates memory for the process
     - Portions of this memory are designated as the *code segment, data segment, & stack segment*
       - Machine code instructions will be loaded into the code segment.
       - Data segment will be used to store strings, arrays, etc. used by the program
       - Stack segment is used for function calls (e.g., keep track of what functions called what functions). More in Chapter 8.
  2. Reads the .exe file from disk
     - Machine code from the **text** section loaded into code segment
     - Bytes from the **data** section loaded into the data segment
  3. Reads the **idata** section and loads DLLs if necessary
  4. "Connects" procedure calls in the program to the correct entrypoints in the DLL
  5. Sets the instruction pointer to the program entrypoint (to start running code in the code segment)

For More Information

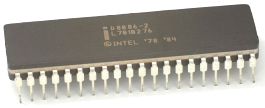
- J. Levine, *Linkers and Loaders,* Morgan Kaufmann (2000)
- M. Pietrek, "An In-Depth Look into the Win32 Portable Executable File Format," *MSDN Magazine* (February, 2002)
  http://msdn.microsoft.com/en-us/magazine/cc301805.aspx
- R. Kath, The Portable Executable File Format from Top to Bottom
  http://www.csn.ul.ie/~caolan/publink/winresdump/winresdump/doc/pefile2.html
- *Microsoft PE and COFF Specification*
  http://msdn.microsoft.com/en-us/library/windows/hardware/gg463119.aspx



§2.2
x86 Architecture Details

Intel x86 Microprocessors

- *(1978)* Intel **8086** 16-bit registers, 16-bit data bus
  Intel 8088 16-bit registers, 8-bit data bus
  All subsequent processors are backward-compatible



- *(1985)* Intel **80386** (AKA 386 or i386)
  32-bit registers, 32-bit data bus
  First IA-32 processor

Sources: 8086: http://en.wikipedia.org/wiki/File:KL_Intel_D8086.jpg 80386: http://en.wikipedia.org/wiki/File:KL_Intel_i386DX.jpg

x86 Modes of Operation

- **Real-Address Mode**
  - Provided for backward compatibility with the 8086
  - Processor boots in this mode (for backward compatibility)
  - 20-bit memory addresses ⇒ **1 MB address space**
- **Protected Mode** – OS's started using this with the 80386 processor
  - Modern Intel processors are intended to be run primarily in this mode
    - Windows, Linux run in protected mode
  - Designed for multitasking – prevents processes from overwriting each other's memory
  - 32-bit memory addresses ⇒ **4 GB address space**
- **Q.** When the processor ran your program from Lab 1, what mode was it in?
- *More modes available - see textbook*

x86 Registers

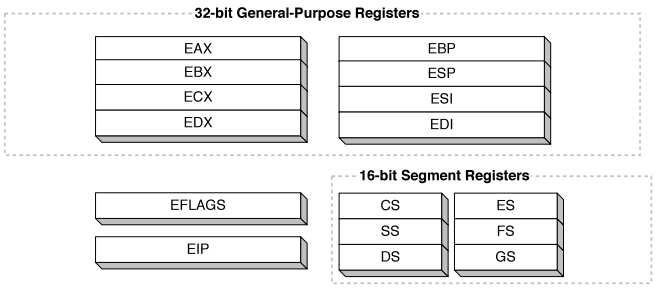


Image Source: Irvine 6/e

x86 Registers

- **32-bit general-purpose registers:**
  - EAX (Extended) Accumulator
  - EBX Base
  - ECX Count
  - EDX Data
  - ESI Source Index
  - EDI Destination Index
  - ESP Stack Pointer
  - EBP Base Pointer

Names are based on historical or special uses.

Do not use ESP or EBP for arithmetic/data transfer; they have special uses (Chapter 8)

x86 Registers

- General-purpose registers (except EBP, ESP) can be used more or less arbitrarily, but some instructions use them for special purposes, e.g.,
  - The `mul` (multiplication) instruction requires one operand to be in EAX (Chapter 7)
  - EBP is used to access function parameters and local variables in procedures (Chapter 8)
  - ESI and EDI are used by high-speed memory transfer instructions, e.g., `movsb` (Chapter 9)

Homework

- **Lab 1** due now (on paper)
- **Homework 1** will be posted tonight or early tomorrow
  - Due in Canvas next Friday (9/5) at 2:00 p.m.
  - *Must* submit a PDF file in Canvas – hardcopy (paper) not accepted