CSCI-GA.2250-001

Operating Systems

Lecture 1: Introduction

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Who Am I?

• Mohamed Zahran (aka Z)
• Computer architecture/OS/Compilers Interaction
• http://www.mzahran.com
• Office hours: Tuesdays 5:00-7:00 pm
• Room: WWH 328
• Course web page: http://cs.nyu.edu/courses/fall11/CSCI-GA.2250-001/
Formal Goals of This Course

• What exactly is an operating systems?
• How does the OS interact with the hardware and other software applications?
• Main concepts of an OS
• OS in many contexts
Informal Goals of This Course

• To get more than an A
• To learn OS and enjoy it
• To use what you have learned in MANY different contexts
• To be able to develop your own OS if you want to
• To start your research project in OS
The Course Web Page

• Lecture slides
• Info about mailing list, labs, ...
• FAQ
• Useful links (manuals, tools, book errata, ... )
The Textbook

Modern Operating Systems 3/E
Andrew S. Tanenbaum
ISBN-10: 0136006639
Grading

- Homeworks : 10%
- Project : 30%
- Midterm (Oct 4th) : 25%
- Final : 35%
Media Player
emails
Games
Word Processing
Does a programmer need to understand all this hardware in order to write these software programs?
Operating System
The Two Main Tasks of OS

• Provide programmers (and programs) a clean abstract set of resources

• Managing the hardware resources
Application programs

Operating system

Hardware

Beautiful interface

Ugly interface
A Glimpse on Hardware
A Glimpse on Hardware
Booting Sequence

• BIOS starts
  – checks how much RAM
  – keyboard
  – other basic devices
• BIOS determines boot Device
• The first sector in boot device is read into memory and executed to determine active partition
• Secondary boot loader is loaded from that partition.
• This loaders loads the OS from the active partition and starts it.

POST (Power On Self Test)
OS

Types

• Mainframe OS
  • batch
  • transaction processing
  • timesharing
  • e.g. OS/390
• Server OS
• Multiprocessor OS
• PC OS
• Embedded OS
• Sensor node OS
• RTOS
• Smart card OS

Concepts

Different Structures
**Types**
- Mainframe OS
  - batch
  - transaction processing
  - timesharing
  - e.g. OS/390
- Server OS
- Multiprocessor OS
- PC OS
- Embedded OS
- Sensor node OS
- RTOS
- Smart card OS

**Concepts**
- Processes
  - Its address space
  - Its resources
  - Process table
- Address space
- File system
- I/O
- Protection

**Different Structures**
OS

Types
- Mainframe OS
  - batch
  - transaction processing
  - timesharing
  - e.g. OS/390
- Server OS
- Multiprocessor OS
- PC OS
- Embedded OS
- Sensor node OS
- RTOS
- Smart card OS

Concepts
- Processes
  - Its address space
  - Its resources
  - Process table
- Address space
- File system
- I/O
- Protection

Different Structures
- Monolithic
- Layered systems
- Microkernels
- Client-server
- Virtual machines
Source Code to Execution

1. **C source**
   - Compiler
   - Assembly

2. Object File
   - Assembler
   - Library
   - Linker

3. Executable
   - Loader
   - DLL
What happens to your program after it is compiled but before it can be executed?
The OS Expectation

• The OS expects executable files to have a specific format
  – Header info
    • Code locations and size
    • Data locations and size
  – Code & data
  – Symbol Table
    • List of names of things defined in your program and where they are defined
    • List of names of things defined elsewhere that are used by your program, and where they are used.
Example of Things

#include <stdio.h>
extern int errno;

int main () {
    printf ("hello, world\n")
    <check errno for errors>
}

• Symbol defined in your program and used elsewhere
  • main

• Symbol defined elsewhere and used by your program
  • printf
  • errno
Two Steps Operation: Parts of OS

**Linking**
- Stitches independently created object files into a single executable file (i.e., `a.out`)
- Resolves cross-file references to labels
- Listing symbols needing to be resolved by loader

**Loading**
- copying a program image from hard disk to the main memory in order to put the program in a ready-to-run state
- Maps addresses within file to physical memory addresses
- Resolves names of dynamic library items
- schedule program as a new process
Libraries (I)

• Programmers are expensive.
• Applications are more sophisticated.
  – Pop-down menus, streaming video, etc
• Application programmers rely more on library code to make high quality apps while reducing development time.
  – This means that most of the executable is library code
Libraries (II)

- A collection of subprograms
- Libraries are distinguished from executables in that they are not independent programs
- Libraries are "helper" code that provides services to some other programs
- Main advantages: reusability and modularity
Static Libraries

• These libraries are stored on disk.
• Linker links only the libraries referenced by the program
• Main disadvantage: needs a lot of memory (for example, consider standard functions such as printf and scanf. They are used almost by every application. Now, if a system is running 50-100 processes, each process has its own copy of executable code for printf and scanf. This takes up significant space in the memory.)
Dynamic Link Libraries (Shared Libraries)

• Why not keep those shared library routines in memory and link at object file when needed? (DLLs)

• A shared library is an object module that can be loaded at run time at an arbitrary memory address, and it can be linked to by a program in memory.

• An application can request a dynamic library during execution

• Main advantage: saving memory

• Main disadvantage: ~10% performance hit
A Bit About Relocation

• modifies the object program so that it can be loaded at an address different from the location originally specified
• The compiler and assembler (mistakenly) treat each module as if it will be loaded at location zero

(e.g. \textit{jump 120} is used to indicate a jump to location 120 of the current module)
A Bit About Relocation

• To convert this relative address to an absolute address, the linker adds the base address of the module to the relative address.

• The base address is the address at which this module will be loaded.

Example: Module A is to be loaded starting at location 2300 and contains the instruction

  jump 120

The linker changes this instruction to

  jump 2420
A Bit About Relocation

• How does the linker know that Module A is to be loaded starting at location 2300?
  – It processes the modules one at a time. The first module is to be loaded at location zero. So relocating the first module is trivial (adding zero). We say that the relocation constant is zero.
  – After processing the first module, the linker knows its length (say that length is L1).
  – Hence the next module is to be loaded starting at L1, i.e., the relocation constant is L1.
  – In general the linker keeps the sum of the lengths of all the modules it has already processed; this sum is the relocation constant for the next module.
A Bit About Relocation

Module M5 will go here. Its relocation constant is L1+L2+L3+L4.
A Bit About Relocation

Base M4 = L1 + L2 + L3

Value of f = Base + rel
Enough for Today

• We are done with Chapter 1
• Read the whole thing. It is a bit long but easy to read and is entertaining
• Skim (not skip!): 1.2, 1.3, 1.6, and 1.8