CMPSC 311 - Introduction to Systems Programming
Module: Systems Programming

Professor Patrick McDaniel
Fall 2013
Patrick McDaniel

- Professor of Computer Science and Engineering
- Co-head of Security Group @ PSU
Things I take seriously

• Ethics
  ‣ Don’t cheat
  ‣ Don’t lie (even if the truth is embarrassing …)

• Show me the respect not to ...
  ‣ Surf, text, email, or other activity in my lecture.
    • Pay attention: No laptops, phones, ipads, etc.
  ‣ Show up late.
  ‣ Lie to me, con me …
MOSS

• Moss (for a Measure Of Software Similarity) is an automatic system for determining the similarity of programs. To date, the main application of Moss has been in detecting plagiarism in programming classes. Since its development in 1994, Moss has been very effective in this role. The algorithm behind moss is a significant improvement over other cheating detection algorithms (at least, over those known to CS folks).

Try it: http://theory.stanford.edu/~aiken/moss/
Software Systems

• A platform, application, or other structure that:
  ‣ is composed of multiple modules
    • the system’s **architecture** defines the *interfaces of* and *relationships between* the modules
  ‣ usually is complex
    • in terms of its implementation, performance, management
  ‣ hopefully meets some requirements
    • performance, security, fault tolerance, data consistency
100,000 Foot View of Systems

hardware

CPU  memory  storage  network
GPU  clock  audio  radio  peripherals

OS / app interface (system calls)

HW/SW interface (x86 + devices)

C application

C standard library (glibc)

C++ application

C++ STL / boost / standard library

Java application

JRE

operating system
A layered view

provides service to layers above

understands and relies on layers below

client

client

client

your system

layer below

layer below

• • •
A layered view

more useful, portable, reliable abstractions

constrained by performance, footprint, behavior of the layers below

- client
- client
- client

your system

- layer below
- layer below

-
Example system

- Operating system
  - a software layer that abstracts away the messy details of hardware into a useful, portable, powerful interface
  - modules:
    - file system, virtual memory system, network stack, protection system, scheduling subsystem, ...
    - each of these is a major system of its own!
  - design and implementation has many engineering tradeoffs
    - e.g., speed vs. (portability, maintainability, simplicity)
Another example system

• Web server framework
  ‣ a software layer that abstracts away the messy details of OSs, HTTP protocols, database and storage systems to simplify building powerful, scalable Web services
  ‣ modules:
    • HTTP server, HTML template system, database storage, user authentication system, ...
  ‣ also has many, many tradeoffs
    • programmer convenience vs. performance
    • simplicity vs. extensibility

Note: we will focus on the OS system this semester.
Systems and Layers

• Layers are collections of system functions that support some *abstraction* to service/app above
  ‣ Hides the specifics of the implementation of the layer
  ‣ Hides the specifics of the layers below
  ‣ Abstraction may be provided by software or hardware
  ‣ Examples from the OS layer
    • processes
    • files
    • virtual memory
A real world abstraction ...

- What does this thing do?

What about this?
Processes

- Processes are independent programs running concurrently within the operating systems
  - The execution abstraction provides is that it has sole control of the entire computer (a single stack and execution context)

Tip: if you want to see what processes are running on your UNIX system, use the “ps” command, e.g., “ps -ax”.

Files

• A file is an abstraction of a read only, write only, or ready/write data object.
  ‣ A data file is a collection of data on some media
    • often on secondary storage (hard disk)
  ‣ Files can be much more: in UNIX nearly everything is a file
    • Devices like printers, USB buses, disks, etc.
    • System services like sources of randomness (RNG)
    • Terminal (user input/out devices)

Tip: /dev directory of UNIX contains real and virtual devices, e.g., “ls /dev”.
Virtual Memory

• The *virtual memory* abstraction provides control over an imaginary address space
  ‣ Has a virtual address space which is unique to the process
  ‣ The OS/hardware work together to map the address on to ...
    • Physical memory addresses
    • Addresses on disk (*swap space*)

• Advantages
  • Avoids interference from other processes
  • *swap* allows more memory use than physically available
Byte-Oriented Memory Organization

• Programs Refer to Virtual Addresses
  ‣ Conceptually very large array of bytes
  ‣ Actually implemented with hierarchy of different memory types
  ‣ System provides address space private to particular “process”
    • Program being executed
    • Program can clobber its own data, but not that of others

• Compiler + Run-Time System Control Allocation
  ‣ Where different program objects should be stored
  ‣ All allocation within single virtual address space
Machine Words

- Machine Has “Word Size”
  - Nominal size of integer-valued data including addresses
  - Many traditional machines use 32 bits (4 bytes) words
    - Limits addresses to 4GB
    - Becoming too small for memory-intensive applications
  - Recent systems use 64 bits (8 bytes) words
    - Potential address space $\approx 1.8 \times 10^{19}$ bytes
    - x86-64 machines support 48-bit addresses: 256 Terabytes

- Machines support multiple data formats
  - Fractions or multiples of word size
  - Always integral number of bytes
Word-Oriented Memory Organization

- **Addresses Specify Byte Locations**
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)

<table>
<thead>
<tr>
<th>32-bit Words</th>
<th>64-bit Words</th>
<th>Bytes</th>
<th>Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr = 0000</td>
<td>Addr = 0000</td>
<td></td>
<td>0000</td>
</tr>
<tr>
<td>Addr = 0004</td>
<td></td>
<td></td>
<td>0001</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td></td>
<td></td>
<td>0002</td>
</tr>
<tr>
<td>Addr = 0012</td>
<td></td>
<td></td>
<td>0003</td>
</tr>
<tr>
<td></td>
<td>Addr = 0008</td>
<td></td>
<td>0004</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0005</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0006</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0007</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0008</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0009</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0011</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0012</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0013</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0014</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0015</td>
</tr>
</tbody>
</table>
Systems programming

• The programming skills, engineering discipline, and knowledge you need to build a system using these abstractions:
  ‣ **programming**: C (the abstraction for ISA)
  ‣ **discipline**: testing, debugging, performance analysis
  ‣ **knowledge**: long list of interesting topics
    • concurrency, OS interfaces and semantics, techniques for consistent data management, algorithms, distributed systems, ...
    • most important: deep understanding of the “layer below”
Programming languages

- **Assembly language / machine code**
  - *(approximately)* directly executed by hardware
  - tied to a specific machine architecture, not portable
  - no notion of structure, few programmer conveniences
  - possible to write really, really fast code

- **Compilation** of a programming language results in executable code to be run by hardware.
  - gcc (C compiler) produces target machine executable code (ISA)
  - javac (Java compiler) produces Java Virtual Machine executable code
Programming languages

• Structured but low-level languages (C, C++)
  ‣ hides some architectural details, is kind of portable, has a few useful abstractions, like types, arrays, procedures, objects
  ‣ permits (forces?) programmer to handle low-level details like memory management, locks, threads
  ‣ low-level enough to be fast and to give the programmer control over resources
    • double-edged sword: low-level enough to be complex, error-prone
    • shield: engineering discipline
Programming languages

• High-level languages (Python, Ruby, JavaScript, ...)
  ‣ focus on productivity and usability over performance
  ‣ powerful abstractions shield you from low-level gritty details (bounded arrays, garbage collection, rich libraries, ...)
  ‣ usually interpreted, translated, or compiled via an intermediate representation
  ‣ slower (by 1.2x-10x), less control
Discipline

• Cultivate good habits, encourage clean code
  ‣ coding style conventions
  ‣ unit testing, code coverage testing, regression testing
  ‣ documentation (code comments!, design docs)
  ‣ code reviews

• Will take you a lifetime to learn
  ‣ but oh-so-important, especially for systems code
    • avoid write-once, read-never code
Knowledge

• Tools
  ‣ gcc, gdb, g++, objdump, nm, gcov/lcov, valgrind, IDEs, race
detectors, model checkers, ...

• Lower-level systems
  ‣ UNIX system call API, relational databases, map/reduce,
Django, ...

• Systems foundations
  ‣ transactions, two-phase commit, consensus, handles,
virtualization, cache coherence, applied crypto, ...