HARVARD
School of Engineering
and Applied Sciences

Topics in Systems

CS61, Lecture 24

Hanspeter Pfister, Daniel Margo, Stephen Chong

December 2, 2010

Announcements

- Lab 5 due today
- Final exam Thursday Dec 16, 2pm-5pm
  - Regular office hours will continue until Dec 15, unless otherwise posted
- Q now open
  - Please give feedback!
  - You will receive email regarding course evaluations, or go to my.harvard.

Today and Thursday

- How does what we've learnt so far relate to:
  - The world around us, including technologies we use every day?
  - Current trends in computer systems technology and research?
- Next two lectures will look at a variety of topics
  - Moore's Law
  - Internet-scale
  - Mobile and embedded systems
  - GPUs
  - Operating systems
  - Programming languages

Operating Systems
(CS161)

What is an Operating System?

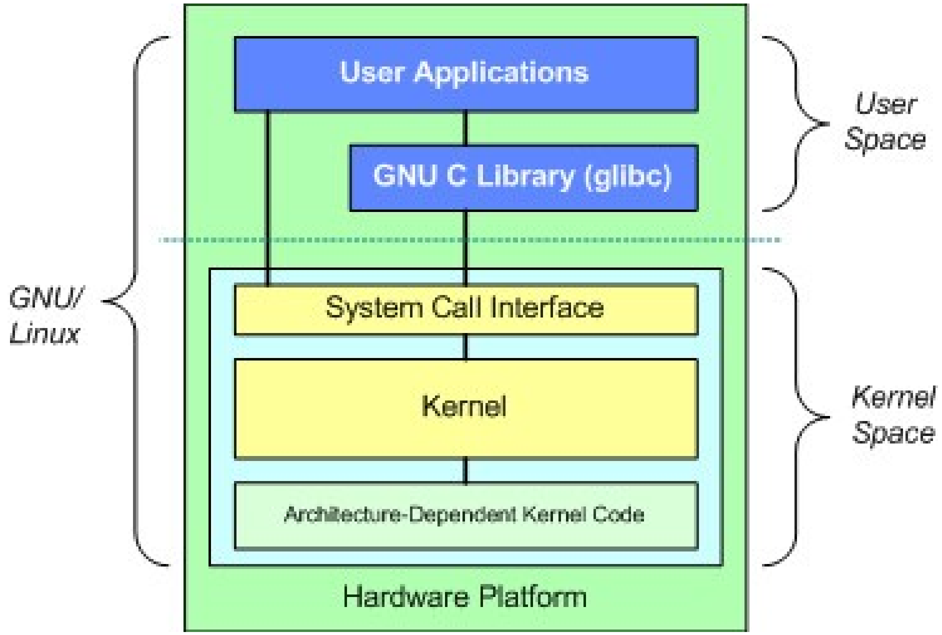
- The code between processes and hardware.
- It provides abstractions and allocates resources
  - CPU (multi-threading)
  - Memory (virtual memory)
  - Hard Disk (filesystem)
- The OS is an illusionist.

What is an Operating System?

- The code between processes and hardware.
- It provides abstractions and allocates resources
  - CPU (multi-threading)
  - Memory (virtual memory)
  - Hard Disk (filesystem)
- The OS is an illusionist.
  - Which is to say, it is a stupendous liar.

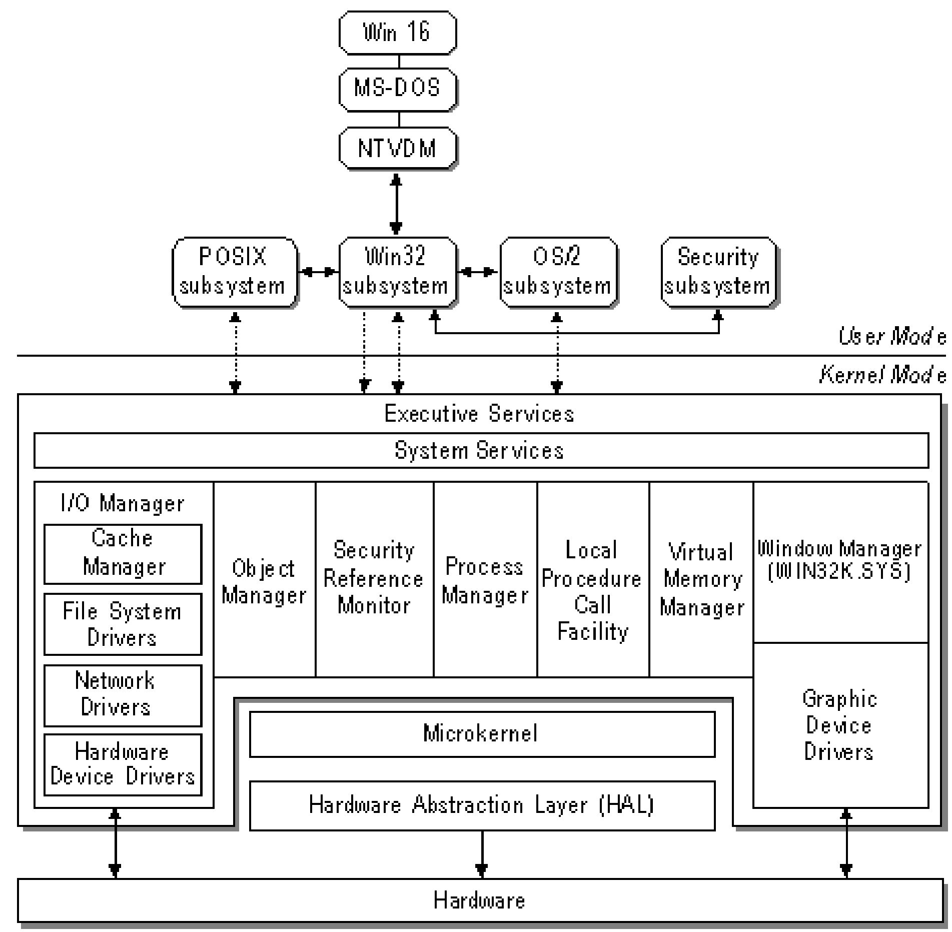
OS Architecture

- Applications make requests of the OS kernel through the system call interface and other traps (more on this shortly).
- There are many user applications, but only one OS; hence, the OS must handle concurrent requests.
- Most of the kernel is written in a portable high-level language (C).
- Small hardware-dependent layer written in machine code.



OS Architecture

- Kernel code is complex; there are many popular designs. NT is a “hybrid monolithic/microkernel”.
- Most designs strive to be modular; one component for each hardware abstraction.
- Lots of design concerns:
  - Good Abstractions!
  - Fairness
  - Speed
  - Security
  - Maintainability



Syscall Interface

- In CS61 we taught you:
  - To run a program: `fork` and `execve`!
  - ...so what does that mean?
- `fork` and `execve` are **syscalls:**
  - A request to the OS to do something. It may fail.
  - A lot like a function call into a shared library.
- But unlike a shared library, the OS is “trusted”:
  - User processes aren't allowed to mess with other processes or hardware, or change file permissions, etc.
  - But the OS has to; it uses special CPU instructions to do so.

Kernel Mode

- The `syscall` instruction causes the CPU to jump to the OS's “syscall handler” in **kernel mode**.
- In kernel mode, the CPU can do things like:
  - Access any process's memory.
  - Manipulate page tables.
  - Access hardware (e.g. the hard disk).
  - Transfer control between processes.
- The syscall handler inspects argument registers to figure out what the user wants, and then calls the appropriate OS component to handle the request.

Traps

- A **trap** is any control transfer to the kernel mode OS.
  - Syscalls are a voluntary trap.
- Involuntary traps happen constantly!
  - Segfault! OS takes control, kills your process.
  - Page fault! OS takes control, loads your page.
  - Device interrupts! Your file write is done/packet has arrived.
  - Timer interrupts! You've had the CPU for long enough.
- Setting traps is a kernel mode privilege.
  - Traps are how the OS maintains control over the machine.
  - The OS must never lose control! *No user code in kernel mode*.

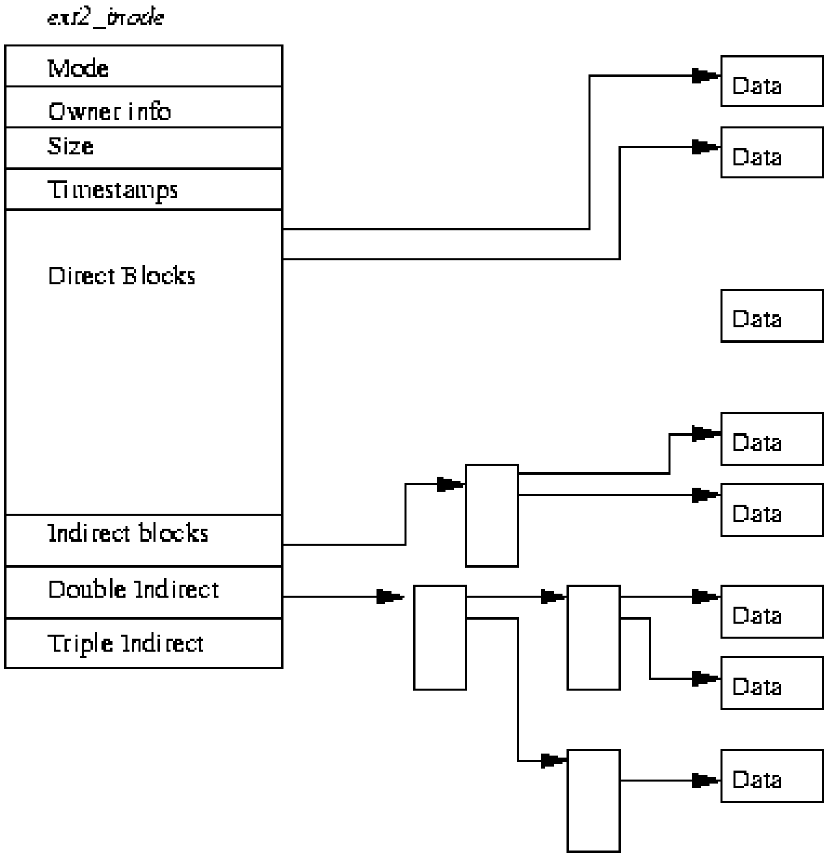
Filesystems

In CS61, we told you...

- Files are:
  - “Infinite-length” byte-addressable character streams.
  - You can open, close, read, and write them.
  - They live in hierarchical directories.
- Hard discs are:
  - Platters, tracks, sectors, and blocks of data. “Logical blocks” are addressable by their ID number.
  - Blocks are fixed-length, and sometimes they just go bad.
- There is a **lot of code** between files and hard disks!

Files (ext2)

- Every file has a corresponding disk structure called an **inode**.
- The inode contains file metadata, and pointers to data blocks.
- Only so much room for pointers...so we also have pointers to blocks full of pointers!
- ext2 has a max file size (2TB).



Directories (ext2)

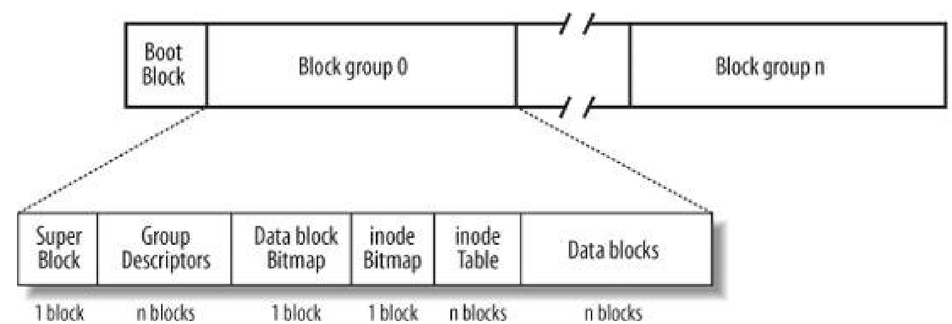
- A directory is like a file that contains filenames and inode #s.
- Note that filenames are a property of the containing directory, not the file!
- To find the file “/a/b/c”:
  - Open unique “/” directory.
  - Find “a/” in “/”.
  - Find “b/” in “a/”.
  - Find “c” in “b/”.
- A cache speeds this up a lot.

	inode	rec_len	name_len	file_type	name							
0	21	12	1	2	.	\0	\0	\0				
12	22	12	2	2	.	.	\0	\0				
24	53	16	5	2	h	o	m	e	1	\0	\0	\0
40	67	28	3	2	u	s	r	\0				
52	0	16	7	1	o	l	d	f	i	l	e	\0
68	34	12	4	2	s	b	i	n				

Filesystem (ext2)

- The filesystem consists of logical blocks organized in **block groups.**

- Each group contains:
  - A copy of the **superblock** (filesystem metadata).
  - A **group descriptor** (block group metadata).
  - Bitmaps that tell which blocks are allocated/free.
  - inodes and data blocks.



A Modern Filesystem (ZFS)

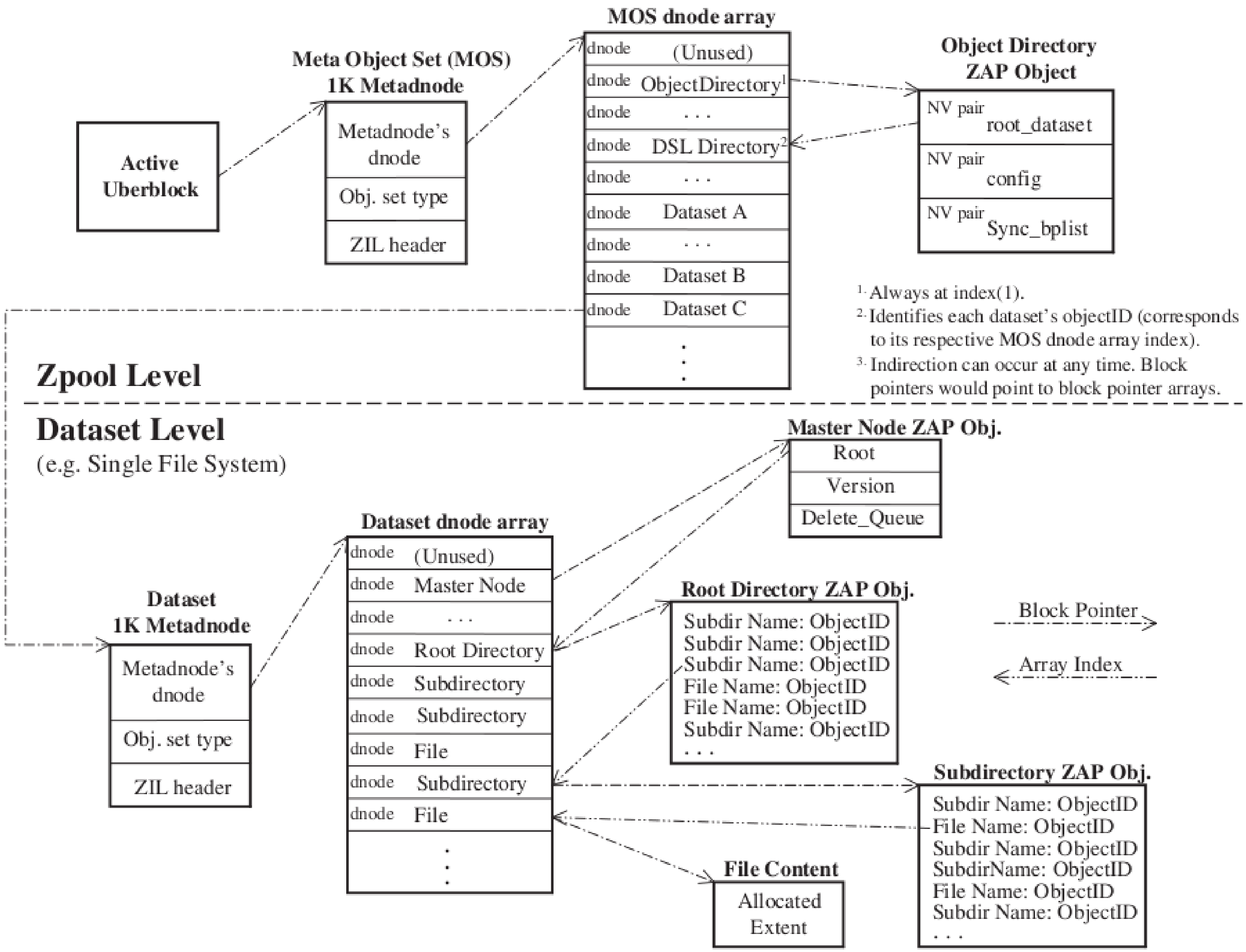


Fig. 2 – On-Disk Data Walk (Figure adapted from Bruning, 2008a).

CS161

- CS161 is a project oriented-course.
  - We give you a half-finished OS...
  - 202 .c files; 84 .h files; 30,514 lines of code.
  - ...and then you have to finish it.
- The OS we give you, OS/161, is “real”:
  - Written at Harvard by David Holland.
  - Inspired by NetBSD, and very similar.
  - Runs on a machine simulator (for sane debugging).

CS161 Projects

- Synchronization:
  - Implement primitives!
- Syscalls:
  - Implement all of them! Every one! Even `fork` and `execve`!
- Virtual Memory:
  - Write it from scratch. And make it thread safe.
  - And don't forget the OS has to page *itself*. Pretty weird stuff.
- Filesystems:
  - Make it thread safe (surprisingly challenging).
  - Performance is important; changes are encouraged.
- <u>You will learn a lot if you put in the effort.</u>

Questions?

HARVARD
School of Engineering and Applied Sciences

Topics in Systems: Programming Languages

CS61, Lecture 24

Stephen Chong

December 2, 2010

What is PL?

- We use **programming languages** to tell computers what to do
  - Describes computation
  - An interface between humans and computers
- PL is the study of programming languages
  - Principles and limitations of **computing** (programming) **models**
  - Design, implementation, use, and comparison of systems or languages built on these models

Benefits of PL

- Widely-applicable design and implementation techniques
  - Understanding (and efficiently implementing) many models of computation leads to insights and principles for system design
  - e.g., Google's map-reduce framework
    - Inspired by map and reduce constructs in LISP and other functional languages
    - Stateless, effect-free computation over streams of data–allows it to scale
  - e.g., Many systems take complex input and perform computation described by that input
    - Web browsers, printer drivers, PDF renderers, scripted robot control systems, spreadsheets, …

Benefits of PL

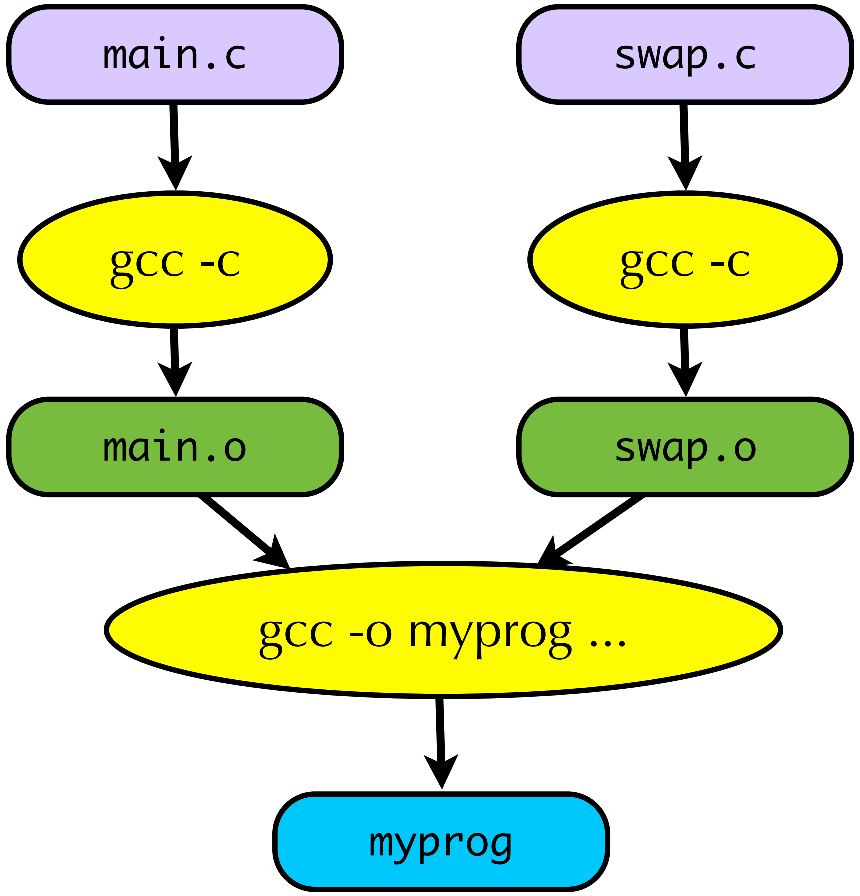
- Create new domain specific languages or Virtual Machines
  - Mathematica, MATLAB, LaTeX, Verilog, VHDL, …
  - Many programmers create new domain specific APIs, languages, or VMs.
    - How expressive should language be? How to execute programs?
    - Many common mistakes: dynamically scoped variables, non-symmetric functional call and return, ...
- Domain specific languages provide computational model for thinking about data and algorithms specific to problems in a specific domain
  - e.g., How to use map-reduce effectively?
    - Sawzall, Dryad, Pig: languages that give a higher level of abstraction than MapReduce
  - E.g. MLFi

Abstraction

- Programming languages provide **abstraction**
- Abstractions of underlying machinery or mechanisms
  - e.g., assembly language instead of machine opcodes
  - e.g., high-level languages (C, Java, Pascal, ...) instead of assembly
  - e.g., monitors, semaphores, actors, join-patterns, …, provide abstraction of concurrent execution
- Abstractions of domain specific concepts
  - Swazall, Dryad, Pig, MLFi, …
- Sapir–Whorf hypothesis: language influences thought

Compilation vs Interpretation

- Compilation: translating higher level languages to lower level languages
  - e.g., C to assembly
- Interpretation: a program (interpreter) executes the source code directly
  - e.g., JavaScript interpreter in browser
- Pros and cons:
  - Efficiency, portability, obfuscation, ease of rewriting
- Blurring the boundaries:
  - Virtual machines
  - JIT compilation



Functional languages

- What values can a program manipulate at runtime?
  - Integers, strings, floats, …
- Functional languages allow *functions* to be runtime values
  - i.e., functions can be given as arguments, received as return values, etc.
  - e.g.,
    - `addFour ≡ λx:int. x+4` is a function that takes an argument `x` and adds 4 to it.
    - `foo ≡ λf:int→int. f( f(34) )` is a function that takes an argument `f`, applies `f` to 34, and applies `f` to the result
    - `foo(addFour)` evaluates to 42
  - Can be mind-bending the first time you see it!
    - `fact ≡ λf,n. if (n=0) then 1 else n*(f(f,n-1))`
    - `fact(fact, 5)`

Functional languages

- Functional languages are typically **pure**
  - Without side-effects, do not affect memory or perform I/O
- Simplifies concurrent programming
  - All computations just perform reads
  - Means concurrent computations do not affect each other so synchronization simpler

Declarative vs. Imperative

- Imperative languages: computation described as *commands*: statements that change program state
  - e.g., `x := 1; while (x < 10) x := x+1;`
- Declarative languages: computation described as *what* the computation should achieve, not *how* it should achieve it
  - e.g., `select avg(salary) from employees where dept = “Accounting”`
- Presents different abstractions to the programmer
  - Declarative programming higher level, and typically more compact
  - SQL one of the big success of declarative programming
- Joe Hellerstein (visiting Harvard this year): BOOM: Orders of Magnitude simpler code for the cloud

Language-based security

- Using programming language techniques and abstractions for security
- Using formal semantics of programming languages to **define** security
  - What does it mean for a system to be secure?
- Mechanisms to enforce language abstractions
  - E.g., buffer overruns are dangerous because they violate the programmer's abstraction. Mechanisms to enforce this abstraction (e.g., error when array accessed out of bounds) improve security
- Mechanisms to enforce/reason about program behavior
  - Program analysis
    - e.g., type system can prove that program never treats an `int` as if it were a pointer
    - e.g., reasoning about correctness of cryptographic protocols
  - Program rewriting
    - Does the program satisfy correct behavior? Who knows? Just rewrite it so that it does...
  - ...

Language-based security

- Greg Morrisett: using sophisticated logics to prove that programs are "correct", including programs of Unknown Provenance
- Me: application-specific information security: specifying, reasoning about, and enforcing

HARVARD
School of Engineering
and Applied Sciences

The end of CS 61

What we’ve covered

- An introduction to computer systems
  - Machine representation of data and programs
  - Compilation, optimization, linking, loading
  - Memory, storage, caching, virtual memory, dynamic memory management
  - Systems programming
    - I/O, sockets, processes
  - Concurrency
    - Threads, synchronization mechanisms, synchronization problems
- Answered (to some extent!) the questions
  - What happens when I run a program?
  - How do computers work?
  - What affects the performance and reliability of my programs?

Where to from here

- We've just scratched the surface!
- Some courses
  - CS 141: Computing hardware
  - CS 143: Computer networks
  - CS 152: Programming languages
  - CS 153: Compilers
  - CS 161: Operating systems
  - CS 171: Visualization
  - CS 175: Computer graphics
  - CS 189r: Autonomous multi-robot systems
  - Other 100 level courses and many 200 level courses...
- Harvard Computer Society http://www.hcs.harvard.edu
- Systems Research at Harvard http://www.eecs.harvard.edu/~syrah/

Final exam

- Final exam Thursday Dec 16, 2pm-5pm
- All material covered in lecture (except guest lecture), sections, and labs is examinable
- Similar format to midterm
- Open book, closed note

- Office hours will continue as normal
  - (unless otherwise posted)
- A practice final will be released soon