What is this course about?

- Programs
- Platforms
- Performance
- ...

“The system is all the code your program uses that you didn’t have to write.”
What is this course about?

- Programs
- Platforms
- Sharing
- Concurrency
- Storage
- Protection and trust
- Resource management
- Virtualization
- Scale and performance
- Abstractions
Resources to know about

• Course Web
  – Or other links through CS department
  – All powerpoints, policies, reading, schedule, lab (project) instructions are posted there.

• Piazza
  – Announcements
  – Questions from you: anonymous postings OK

• Sakai
  – Grades, quizzes
Meetings

• “Lectures”:  
  – WF 1:25 – 2:40  
  – ~25 lectures total

• “Recitations”:  
  – M 3:05 (same room)  
  – TA: Max Demoulin  
  – (But I will do some)

• Two midterms

• Final exam (1.5x)
Course goals

• Learn to think about systems holistically.
• Sound mastery of structures and principles.
• Reinforce with practical, concrete examples.
• Reduce “unknown unknowns”.

“[T]here are known knowns; there are things we know that we know. But there are also unknown unknowns – there are things we do not know we don't know.”

—United States Secretary of Defense, Donald Rumsfeld

2003
Managing Complexity

Systems are built from components.

Operating systems define styles of software components and how they interact.

OS maps components onto the underlying machine.

…and makes it all work together.
Large, long-lived software systems are like buildings. They are built by workers using standard design patterns/grammar. They depend on some underlying infrastructure. But they can evolve and are not limited by the laws of physics.
A simple module (or “component”)

- A set of procedures/functions/methods.
- An interface (API) that defines a template for how to call/invoke the procedures.
- **State** is just data maintained and accessed by the procedures.
- A module may be a **class** that defines a template (type) for a data structure. A class may have multiple instances (objects) constructed from the template. The class procedures (**methods**) operate on a specific object.

**Abstract Data Type (ADT):** the module’s **state** is manipulated only through its **API** (Application Programming Interface).
OS Platform: A Model

Applications/services: May interact and serve one another.

Libraries/frameworks: Packaged code used by multiple applications

OS platform: same for all applications on a system, e.g., classical OS kernel

OS mediates access to shared resources. That requires protection and isolation.
Isolation and Sharing

Android Security Architecture

“A central design point of the Android security architecture is that no application, by default, has permission to perform any operations that would adversely impact other applications, the operating system, or the user. This includes reading or writing the user's private data (such as contacts or emails), reading or writing another application's files, performing network access, keeping the device awake, and so on.

Because each Android application operates in a **process sandbox**, applications must explicitly share resources and data. They do this by declaring the permissions they need for additional capabilities not provided by the basic sandbox. Applications statically declare the permissions they require, and the Android system prompts the user for consent at the time the application is installed. Android has no mechanism for granting permissions dynamically (at run-time) because it complicates the user experience to the detriment of security.”
Platform abstractions

• Platforms provide “building blocks”…
  • Instantiate/create/allocate
  • Manipulate/configure
  • Attach/detach
  • Combine in uniform ways
  • Release/destroy

• Abstractions are layered.
  • Expose the power through APIs
  • Hide the details behind APIs

The choice of abstractions reflects a philosophy of how to build and organize software systems.
Course prerequisites

• Basic data structures and programming
  – Lists, stacks, queues, graphs, DAGs, trees
  – Abstract data types (ADTs), classes, objects
  – Dynamic data structures

• Basic architecture (CPS 250)
  – CPU context: registers, control
  – Execution: runtime stack and frames
  – Memory and L1/L2/L3 caches, DMA I/O
  – Virtual addressing and memory layout
Code: instructions in memory

_p1:
pushq %rbp
movq %rsp, %rbp
movl $1, %eax
movq %rdi, -8(%rbp)
popq %rbp
ret

These are examples of sequences of machine instructions. Details of instruction set architecture (ISA) vary by machine.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>load</code></td>
<td>_x, R2; load global variable x</td>
</tr>
<tr>
<td><code>add</code></td>
<td>R2, 1, R2; increment: x = x + 1</td>
</tr>
<tr>
<td><code>store</code></td>
<td>R2, _x; store global variable x</td>
</tr>
</tbody>
</table>
Multicore
[Image: ExtremeTech: Moore’s Law is dead, long live Moore’s Law]
Multithreading

Single Threading

Task 1
Task 2
Task 3
Task 4

Thread 1

Core 1

ExecutionTime \approx \sum_{i=1}^{4} Task_i

Thread-level Parallelism

Task 1
Task 2
Task 3
Task 4

Thread 1
Thread 2
Thread 3
Thread 4

Core 1
Core 2
Core 3
Core 4

ExecutionTime \approx \max_{i} Task_i

http://www.cse.wustl.edu/~jain/cse567-11/ftp/multcore
Operating Systems

This is just a plug, from me, for you to know about processes, threads and concurrency issues. A lot of interviewers ask about that stuff, and it's pretty fundamental, so you should know it. Know about locks and mutexes and semaphores and monitors and how they work. Know about deadlock and livelock and how to avoid them. Know what resources a processes needs, and a thread needs, and how context switching works, and how it's initiated by the operating system and underlying hardware. Know a little about scheduling. The world is rapidly moving towards multi-core, and you'll be a dinosaur in a real hurry if you don't understand the fundamentals of "modern" (which is to say, "kinda broken") concurrency constructs.

The best, most practical book I've ever personally read on the subject is Doug Lea's Concurrent Programming in Java. It got me the most bang per page. There are obviously lots of other books on concurrency. I'd avoid the academic ones and focus on the practical stuff, since it's most likely to get asked in interviews.

Coding

You should know at least one programming language really well, and it should preferably be C++ or Java. C# is OK too, since it's pretty similar to Java. You will be expected to write some code in at least some of your interviews. You will be expected to know a fair amount of detail about your favorite programming language.

Other Stuff
Welcome to the Machine
What every computer systems student should know about computers

Jeff Chase
Department of Computer Science
Duke University
January 11, 2013

Any systems course presumes a basic knowledge of computer architecture. This note gives an overview of the key concepts and vocabulary. For the supporting detail, the gold-standard source is *Computer Systems: A Programmer’s Perspective*, popularly known as CS:APP, which gives an in-depth, nuts-and-bolts treatment of system programming with C/Unix on x86 computers.

This note reviews the basic concepts and terms for any conversation about operating systems and applications and how they run on a machine. The title should not be taken to suggest that this note contains everything you need to know: any serious systems student should take some time to get smart with sources like CS:APP. Developing a working understanding of the material in that book will place you with a fairly elite group of system-builders who truly understand how “real code runs on real metal”.

But if the material here is familiar to you then you should be OK for an OS class. If it is *my* class, then these are the terms and meanings that I use to talk about machines. Be aware that others may use their terms slightly differently, and also that terms may have additional meanings in other conversations.
Read it on the course web

Essential C

By Nick Parlante

This Stanford CS Education document tries to summarize all the basic features of the C language. The coverage is pretty quick, so it is most appropriate as review or for someone with some programming background in another language. Topics include variables, int types, floating point types, promotion, truncation, operators, control structures (if, while, for), functions, value parameters, reference parameters, structs, pointers, arrays, the pre-processor, and the standard C library functions.

The most recent version is always maintained at its Stanford CS Education Library URL http://cslibrary.stanford.edu/101/. Please send your comments to nick.parlante@cs.stanford.edu.

I hope you can share and enjoy this document in the spirit of goodwill in which it is given away -- Nick Parlante, 4/2003, Stanford California.
Performance

Queues likely
Grading:
50% exams (2M+F)
50% projects/labs
Normalize to 100

Probability that a random student’s score is X or below
80% of the requests have response time $R$ with $x_1 < R < x_2$.

50% (median)

10% quantile

"Tail" of 10% of requests with response time $R > x_2$.

A few requests have very long response times.

What's the mean $R$?

Understand how the mean (average) can be misleading, e.g. if tail is heavy.
Reading

• Course notes and slides
• External sources on every topic
  – OS in Three Easy Pieces
  – A few academic papers and web readings
  – Yes, even a “comic book”

We’ll look at these with varying levels of scrutiny.
No text, but these may be useful.

$70

A good intro to web-scale systems: Web/SaaS/cloud
http://saasbook.info
Workload: projects

1. Dynamic heap memory (malloc/free)
2. Threads and concurrent programming
3. Black hat lab ("buggyserver")
4. Distributed consensus ("Raft")
Collaboration

• OK among groups:
  – General discussion of course concepts, programming environment, problems, pitfalls.
  – “What does this part of the handout mean?”

• Not OK among groups
  – “Substantive” code sharing
  – “How do I do this part of the handout?”

• Definitely not OK:
  – Using code from a previous semester.
  – (Or from any unattributed source)

• Read the syllabus. If in doubt, ask me.
Thoughts on cheating

Cheating is a form of laziness.

Cheating happens at those other schools.

Duke students work hard and don’t cut corners.

Your work is your own: if in doubt, ask. Listen to what shame tells you.

Cheating is a response to fear, but it just creates more fear.
We will use the **git** version control system.

Repositories are hosted on gitlab.oit service.

Your group’s repository is private to your group and to us (and to the auto-grader).

You will use the **git** command to clone your repository in a local directory/folder.

Use **git** to commit/push/checkin and pull/fetch updates.

The **git** command runs as a local program/process that talks to gitlab.oit over the network.

Q: how does gitlab.oit know who you are?
   - A1: Login to gitlab.oit **with your NetID using Shibboleth**.
   - A2: register a **public key** with gitlab.oit.
SSH

SSH keys

An SSH key allows you to establish a secure connection between your computer and GitLab.

Before generating an SSH key, check if your system already has one by running `cat ~/.ssh/id_rsa.pub`. If you see a long string starting with `ssh-rsa` or `ssh-dsa`, you can skip the ssh-keygen step.

To generate a new SSH key, just open your terminal and use code below. The ssh-keygen command prompts you for a location and filename to store the key pair and for a password. When prompted for the location and filename, you can press enter to use the default.

It is a best practice to use a password for an SSH key, but it is not required and you can skip creating a password by pressing enter. Note that the password you choose here can’t be altered or retrieved.

```
ssh-keygen -t rsa -C "chase@duke.edu"
```

Use the code below to show your public key.

```
cat ~/.ssh/id_rsa.pub
```

Copy-paste the key to the 'My SSH Keys' section under the 'SSH' tab in your user profile. Please copy the complete key starting with `ssh-` and ending with your username and host.
AuthZ 101

subject action
Authenticated by PKI / SSL / MAC "subject says action"

identity

Authenticated by PKI / SSL / MAC "subject says action"

Alice

requester (subject)

credentials

guard

request

authorized

object

service

reference monitor applies guard policy

subj/obj attributes

policy rules

checker

APPROVED
Cryptography for Busy People

- Standard crypto functions parameterized by keys.
  - Key is a fixed-width “random” value (width matters!)
  - Symmetric (DES): fast, requires shared key: $K = K_1 = K_2$
  - Asymmetric (RSA): slow, uses two keys: a keypair $\{K_1, K_2\}$
- “Believed to be computationally infeasible” to break
Asymmetric (public key) crypto

• Each subject/principal possesses a **keypair**.
  - \( \text{Decrypt}(K, \text{Encrypt}(K^{-1}, M)) = M \)

• Keep one key **private**; the other is **public**.

• Either key can be used to encrypt/decrypt.

Anyone can mint a keypair.

If we know one another’s **public** keys then we can communicate securely.
Asymmetric crypto works both ways

A’s private key
or
A’s public key

A’s public key
or
A’s private key
How to use asymmetric crypto?

• **A** can send a message to **B**, encrypted with **A**’s private key.

• **B** can send a message to **A**, encrypted with **A**’s public key.

• Benefits? Other possibilities?
Spelling it out

• **Do** encrypt message $M$ with your **private** key to authenticate it, i.e., to convince the recipient that $M$ really came from you.
  – Better yet, digitally sign $M$: that’s faster (next).

• **Do** encrypt $M$ with the **recipient’s public** key to keep it secret: only the intended recipient can decrypt it.

• **Don’t** encrypt $M$ with your public key: it’s just weird and pointless, since nobody else can read the encrypted message. Bob probably blew his chances with Alice.

• **Don’t** encrypt $M$ with the recipient’s private key: if you know someone’s private key then you should not use it! Forget it and don’t tell anyone.
The point of the remaining slides is:

• We take a broad view of “operating systems” encompassing a variety of application platforms.

• We start with Unix, a canonical/classical OS.

• Unix has continuing relevance: it continues to thrive deep inside the rich platforms we use today: knowing about the Unix kernel helps to understand how they work.

• Hardware and application demands change rapidly. Operating system kernels evolve slowly, but we often add more code around them to adapt to change.

• You’ll see these slides again.
What is this course about?

• “Greater Unix”
  – Classical OS abstractions and structure
  – Systems programming with C and Unix

• Networked systems
  – Sockets and servers, smartphones to clouds
  – Elementary cryptosystems
  – Distributed systems topics

• Managing concurrency
  – Threads, multi-threaded Java

• Managing storage and data
  – Files, caches, disks, recovery, content delivery
Platforms are layered/nested
End-to-end application delivery

Cloud and Software-as-a-Service (SaaS)
Rapid evolution, no user upgrade, no user data management.
Agile/elastic deployment on virtual infrastructure.
SaaS platform elements

[wiki.eeng.dcu.ie]
OpenStack, the Cloud Operating System

Management Layer That Adds Automation & Control

- Connects to apps via APIs
- Self-service Portals for users

[Anthony Young @ Rackspace]
EC2
The canonical public cloud

Virtual Appliance Image

Template (AMI)

Launch different instance types

Large Instance

High-CPU Extra Large Instance

High-Memory Double Extra Large Instance

Amazon EC2

Region: us-east-1
Availability Zone
Availability Zone
Availability Zone

Region: eu-west-1
Availability Zone
Availability Zone
Availability Zone

Other Region...

Paid
Workload

Number of Nodes

Time
Drivers of Change

- Increasing diversity
- Exponential growth
- Aggregation
- Composition
- Orchestration
- Backward compatibility

Broad view: smartphones to servers, web, and cloud.
“Software Architecture”

Software architecture

Computer architecture

Physics stops here.

User Applications

Operating System(s)

Substrate / Architecture

Comparative architecture: what works
Reusable / recurring design patterns
- Used in OS
- Supported by OS
But Java programs are interpreted. They run on an “abstract machine” (e.g., JVM) implemented in software (e.g., a C program). "bytecode"

http://www.media-art-online.org/java/help/how-it-works.html
“Classical OS” Reloaded.

Virtual Machine (JVM)