CS262: Advanced Topics in Computer Systems

Fall 2026

Lecture 1: Introduction

Rishabh Iyer
rishabh.iyer@berkeley.edu

A Quick Intro

Instructor: Rishabh Iyer

- New-ish Assistant Professor in CS (started July'25).
- Research focus: Cloud Computing and AI Systems

TA: Shubham Mishra

- PhD student with Natacha Crooks and John Kubiatowicz
- Research focus: Distributed Systems and Security

Agenda for Today

- What is this course about?
- Logistics
- Paper discussion: Unix

What is This Course About?

Goal: Learn about systems research by

- Reading several seminal papers
- Doing it: work on an exciting project

Hopefully sow the seeds for the next generation of impactful systems.

- Apache Mesos, Ray started out as a course project in this class.

But what is systems research about?

What is Systems Research About?

If we think of systems as everything between applications and HW, systems have two roles:

- Manage (and multiplex) physical resources (e.g., compute, mem, storage, network).
- Provide useful abstractions to applications (files, VMs, shared memory, etc).

But is this a good definition of systems research?

- Any and all work that changes an abstraction or how resources are managed isn't deemed systems research ...

The truth is that it is hard to define what good systems research is

- The goal of the course is to help you identify such research by going through examples

One Possible Answer

Provided by Codex, with some prodding from me

Systems research studies the computing substrate beneath applications: OSes, networks, storage, databases, and runtimes.

It produces new knowledge about which capabilities and guarantees can be provided, at what cost, and under what assumptions.

Its contributions are typically generalizable mechanisms, principles, or limits; with implementations and evaluations providing evidence for those claims.

So What Examples Will We Look At

CS262A is, in my opinion, quite a unique systems course

- Berkeley has historically taken a very broad view of systems
- Culture of avoiding typical silos of operating systems, databases, distributed systems, etc

So this course is almost a whirlwind tour many different categories of systems

- Each lecture is a different category.
- More of a breadth-first approach, as opposed to a depth-first approach.

Schedule Part 1: Classics

Lecture Number	Topic
1	Introduction
2	Databases
3	File systems
4	Transactions
5	OS classics
6	OS (II)
7	Consensus
8	Distributed storage
9	DHTs, KV stores
10	Distributed coordination

Schedule Part 2: The Cloud Era

Lecture Number	Topic
11	Virtualization
12	Cluster mgmt
13	Scheduling
14	Datacenter applications
15	Distributed data processing
16	Serverless
17	Disaggregated/CXL
18	Security & Privacy
19	Verification

Schedule Part 3: The AI Era

Lecture Number	Topic
20	Parallel programming
21	ML programming frameworks
22	Pre-training
23	Post-training
24	Inference (I)
25	Inference (II)
26	Thanksgiving
27	Thanksgiving
28	AI for systems
29	Systems support for agents?
30	OSDI deadline
31	Poster day

Agenda for Today

- What is this course about?
- **Logistics**
- Paper discussion: Unix

Some Important Details

Course website is live: https://cs262a.github.io
- Contains schedule, grading rubric, link to paper review form.

All course-related communication will be done via Slack.
- Those enrolled should have got an invite. Please email me if you have not.

Instructor office hours:
- 11-12pm on Wednesday. Room 1110B in the Gateway

Undergrads who haven't been sent an enrollment code please find me after class.
- **Disclaimer: The class is close to the limit, so I cannot enroll all of you.**

Lecture Format (starting from 8 Sep)

Three parts:

- I will provide an introduction to the area (~15-20 mins)
- Discussions for each paper (20 mins) x2
- Possible in-class design exercise (15-20 mins)

For the first three lectures (including today), I will lead the discussions.

What Do You Have To Do?

- Read and review papers
- Lead and participate in discussions
- Course project

- Grading rubric:
  - 40%: class participation (includes paper reviews).
  - 20%: paper presentations.
  - 40%: course project.

How To Read A Paper?

Key questions:

- What is the problem?
- Why is/was the problem important?
- What is the key idea/insight underlying the solution?
- What assumptions does the solution make?
- Does the paper convincingly demonstrate that it solves the problem?

For older papers:

- What lessons can I learn from the paper that apply to systems today?
- What can I learn from the success or failure of the proposed system?

A Note On The Use of AI

I strongly encourage you to use AI to help you understand the paper and answer questions that would otherwise be very difficult to answer (e.g., historical context of classical papers).

BUT

- Please use it to *augment* your learning not *automate* it.
- Focus on learning. Getting an answer from an agent can create the *illusion* of learning.
- Put yourself first. Remember that the only meaningful outcome of this course (and your education in general) is *you and what you learn*. It is not about a grade or sounding intelligent in class.

Paper Reviews (in the agentic AI era)

Review form has only a single question:

- *List any questions you would like discussed in class for the above paper*
- In my experience, is often the best rubric for how much time someone has spent reading the paper.

Review is due 24 hrs before class.

Important note:

- There are no stupid questions!
- A subset of questions will be anonymized & discussed during the class.

Paper Discussions/Presentations

Assumes that everyone has read the paper

- We will spend minimal time on recapping the details of the paper.
- Can make exceptions if parts are hard to understand, but this won't be the norm.

Outline of a good presentation

- Recaps the paper (i.e., problem, key insights, and key results) in 6-8 mins.
- Presents *opinions* (both positive and negative) for 3-4 mins.
- Poses open-ended questions that leads to discussion for 8-10 mins.

Presentation Logistics

Each student must present or co-present at least one paper.

- Will send out sign-up sheet on Slack.
- No more than 2 presenters per paper.

Presentations are due for feedback via email 1 week in advance.

- First two presentations (i.e., 9/8 and 9/10) can submit 5 days in advance.
- Please use Google slides.

Course Project

Investigate new ideas and solutions in a class research project

- Define the problem
- Execute the research
- Write up and present your research

Ideally, best projects will become conference papers (e.g., OSDI/SOSP, VLDB, NSDI)

Course Project Logistics

Can be done either individually or in groups of up to 3 students.

I will send out a list of potential topics soon.

- You can either choose one or come up with your own.

Proposals due Sept 22. One-page long. Must answer:

- What is the problem?
- How is it solved/addressed today, and what are the limits of current practice?
- What is the final deliverable and what are some milestones along the way?
- Any special resources you will need.

Agenda for Today

- What is this course about?
- Logistics
- **Paper discussion: Unix**

Unix's Main Contribution

Ritchie and Thompson make a striking claim about Unix's greatest contribution.

What do they mean?

> *The success of UNIX lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas, and especially in showing that they can be keys to the implementation of a small yet powerful operating system.*

Before There Was Unix, There Was Multics

Designed to support 9 goals (listed here) that can be grouped into 5 broad categories:

- Interactive computing for users, not batch processing.
- A reliable, persistent store that is a natural part of the programming environment.
- Allow selective sharing of data and invocation of protected operations among mutually distrustful users.
- Let programs use reusable, independently maintained shared procedures.
- Let application I/O be redirected to different entities (e.g., files, devices, or other processing modules) without modifying the application.

Easily Accessible, Reliable, Persistent Store

How did Multics implement this?

Easily Accessible, Reliable, Persistent Store

How did Multics implement this?

- Persistent segments were integrated into virtual memory.
- Programs accessed persistent data through ordinary pointers; the OS transparently paged it between memory and disk. This coupled storage, paging, sharing & protection.
- The OS could persist pages, but could not infer application-level transactions or invariants.

What did Unix do?

Easily Accessible, Reliable, Persistent Store

How did Multics implement this?

- Persistent segments were integrated into virtual memory.
- Programs accessed persistent data through ordinary pointers; the OS transparently paged it between memory and disk. This coupled storage, paging, sharing & protection.
- The OS could persist pages, but could not infer application-level transactions or invariants.

What did Unix do?

- Chose not to implement this abstraction.
- Decoupled process memory from the file system. Represented persistent data as unstructured byte-stream files accessed explicitly with open, read, and write.
- Left representation, atomicity, and crash recovery to applications and databases that understood the data's semantics.

Selected Sharing & Protected Operations

How did Multics implement this?

Selected Sharing & Protected Operations

How did Multics implement this?

- ACLs specified who could access each segment.
- Protection rings and guarded entry points allowed unprivileged programs to invoke specific operations in privileged code without gaining direct access to its data.
- This was fine-grained, but required machinery to validate the entry point, switch protection domains and stacks, and safely pass arguments across the boundary.

What did Unix do?

Selected Sharing & Protected Operations

How did Multics implement this?

- ACLs specified who could access each segment.
- Protection rings and guarded entry points allowed unprivileged programs to invoke specific operations in privileged code without gaining direct access to its data.
- This was fine-grained, but required machinery to validate the entry point, switch protection domains and stacks, and safely pass arguments across the boundary.

What did Unix do?

- Used simple owner/group/other permissions to protect files.
- The setuid bit let a program run with its owner's identity—for example, allowing a payroll program to read a protected database.
- This was much simpler but coarser: the entire program became privileged and had to enforce which operations callers were allowed to perform.

Composing Programs

What Multics did:

Composing Programs

What Multics did:

- Made dynamic linking the default way to construct programs from reusable procedures.
- These procedures became part of the calling program: changing an interface could break its callers, and a faulty procedure could corrupt the entire process.

What Unix did:

Composing Programs

What Multics did:

- Made dynamic linking the default way to construct programs from reusable procedures.
- These procedures became part of the calling program: changing an interface could break its callers, and a faulty procedure could corrupt the entire process.

What Unix did:

- Made a separately executable program (i.e., a process) the unit of composition.
- Programs communicated through byte streams on standard input and output.
- The shell combined them using fork, exec, pipes, and redirection, without sharing memory or being linked together.

Standardized I/O

What Unix did:

- Reduced I/O to byte streams identified by small integer file descriptors.
- Files, terminals, devices, and pipes all used the same read and write interface.
- The shell redirected standard input and output or connected programs with pipes—without modifying the programs.

Revisiting Unix's Main Contribution

"The full exploitation of a carefully selected set of fertile ideas" produced a "small yet powerful operating system."

Persistent storage: careful selection

- Rejected persistent virtual memory

Protected operations: careful selection of simple ideas.

- Simple identities, permissions, and setuid.

Composing programs: Fertile idea

- Process as the unit of composition

Standardized I/O: Full exploitation of simple ideas for power

- The same file-descriptor interface worked for files, devices, and pipes.

UNIX's Technical Legacy

Uniform byte-stream I/O and file descriptors

- open, read, and write became the enduring interface for files, terminals, devices, pipes, and later sockets.

The fork–exec process model

- Process creation, descriptor inheritance, and program replacement still underpin Unix shells, daemons, and process supervisors.

Pipes, filters, and a programmable shell

- Established a lasting model of composing independent programs through simple stream interfaces.

A portable operating system written in C

- Made operating systems portable across hardware and established C and the Unix interface as foundations for BSD, Linux, macOS, and Android.

Broader Lessons for Systems Design

Design around a concrete use case, not maximal generality.

- Unix built an interactive programmer's workbench, not a universal computing utility.
- It helped that the developers were the intended users.

Prefer few, simple abstractions that compose over multiple, complex ones.

- Byte streams, file descriptors, processes, and the shell were modest individually but powerful together.

Mastering complexity is not the same as extracting simplicity.

- When you want to get a system to work, you need to first master complexity.
- When you want to make it easy to use and extend, you need to extract simplicity.