# A systems approach to teaching computer systems

Jerry Saltzer and Frans Kaashoek
{Saltzer, kaashoek}@mit.edu

Massachusetts Institute of Technology

# Who is a professor's customer?

- Students?
- Industry?
- Universities?
- Parents?

*Student!*

# Student's career is ~40 years

- Identify long lasting ideas
- Mechanics will change
- Few students program 40 years
- But many are involved in system design
  - Even if they are not in the IT industry
- A few students need to become experts

*In systems we serve our students poorly*

# Too many system classes

- Operating systems
- Databases
- Computer networks
- Computer architecture
- Computer security
- Distributed systems
- Fault tolerant systems
- Concurrency

*Students don't have time to take all of them, so they leave with gaps in basic systems concepts*

# Classes have the wrong focus

- Most classes require substantial programming
- Few students will need to implement
  - An operating system
  - A parallel computer
  - A database
  - A cryptographic protocol
- Many students need to understand systems
  - Design a Web site
  - Roll out a financial application
  - Advise management on IT strategies

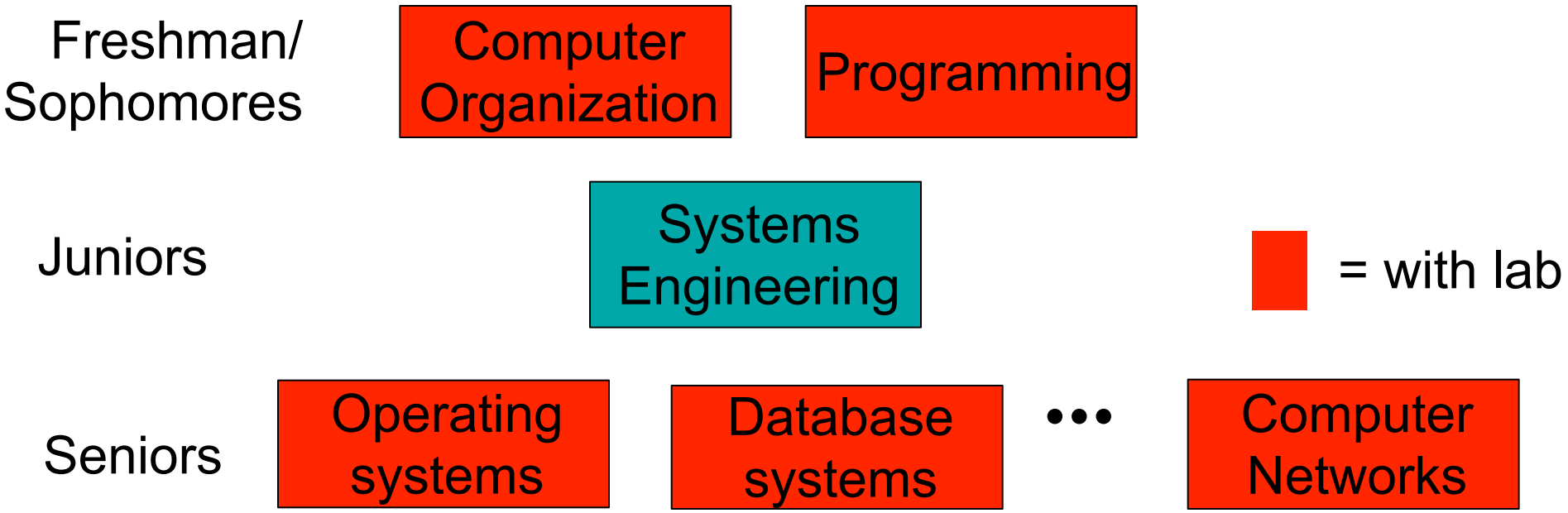
# Poor identification of long-lasting ideas

- Each class takes a semester (or more)
  - No reason to pull out big ideas
- Pressure to focus on details
  - Each class has a lab
  - Must learn some artifact (NachOS, Minix)
  - Details matter (e.g., how to disable interrupts on the x86)

# Lack broad appeal

- Students without “street” knowledge feel at a disadvantage
- Programming creates macho culture
- Little interest from other majors
  - Even though other majors rely more and more on computer systems

# Our approach: a different systems curriculum




- An intro class without programming but with long-lasting ideas
- Follow on classes can go in real depth, including labs
- In-depth often requires general system knowledge

# Opportunity: identify common ideas

- System abstractions fall in one of three categories:
  - interpreters, memory, and communication links
- Atomicity
  - atomic instructions, locks, rename, logs
- Concurrency control
  - semaphores, two-phase locking, two-phase commit
- Virtualization
  - virtual memory, RAID

# Outline

- Content overview
  - Example: virtualization
- Assignments
- Quizzes
- Results
- Adopting

# Computer system engineering (6.033)

- Started in late sixties
- Principles capture long-lasting wisdom
- Key ideas in depth using pseudocode
- Design oriented, instead of programming
  - Students “solve” a design problem and write a report
- Hands-on assignments
  - Reality exposure in lieu of a full-blown lab
- Case studies of successful systems
  - Papers from the research literature
- Large staff for about 200 students per semester

# The mechanics

- 2 large lectures a week: covers key ideas
- 2 small-group discussing meetings per week
  - Discuss research papers w. successful design
- 4 one-pagers
  - Answer question about one of the papers assigned
- 7 Hands-on assignments
  - Poke at several systems from the outside
- 2 design projects per term
  - One individual, one team
- 3 quizzes

*"the EECS humanities class"*

# Content overview

- Introduction: system complexity
- Abstractions: interpreters, memory, and communication links
- Naming: glue to connect abstractions
- Client/server: strong modularity
- Operating systems: isolate client and servers
- Performance: bottlenecks in a pipeline
- Network systems: connect client and servers
- Fault tolerance: modularity in the face of failures
- Transactions:  modularity in the face of concurrency and failures
- Security: modularity in the face of attackers

# Content themes

- The pervasive importance of modularity
  - Abstractions, Naming, Client/service, Layering, etc.
- Robustness and resilience
  - Stronger and stronger modularity
- Design principles

# Principles

- Adopt sweeping simplifications
- Avoid excessive generality
- Be explicit
- Decouple modules with indirection
- Design for iteration
- End-to-end argument
- Keep digging principle
- Law of diminishing returns
- Open design principle
- Principle of least surprise
- Robustness principle
- Unyielding foundation rule
- Safety margin principle
- Avoid rarely used components
- Never modify the only copy!
- One writer rule
- The durability mantra
- Minimize secrets
- Complete mediation
- Least privilege principle
- Separation of privilege
- Economy of mechanism
- Minimize common mechanism

# Example:virtualization

- Key problem: enforcing modularity between applications on same computer
- Key idea: virtualization
  - Virtual memory: address spaces
  - Virtual processors: threads
  - Virtual communication link: pipe, IPC
- Artifact: operating system

# Tease ideas apart



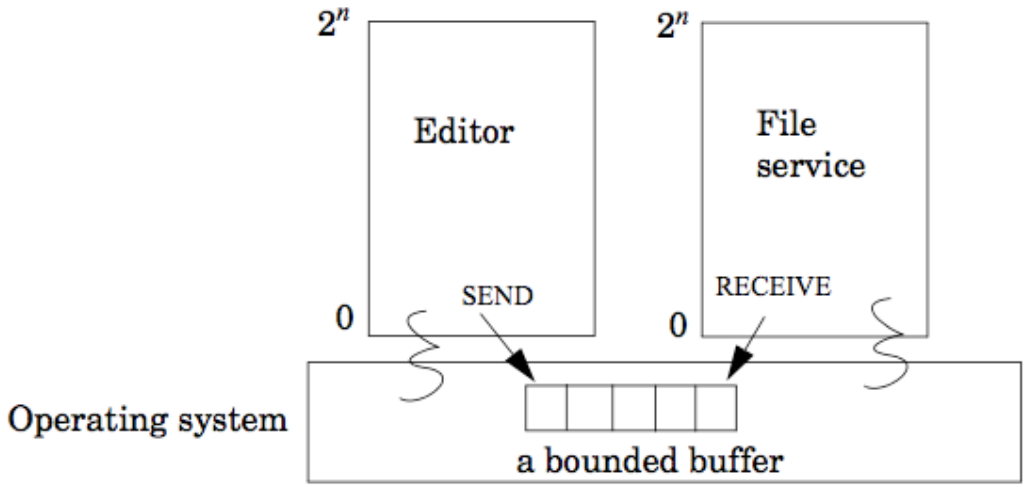



**Figure 5–2:** An operating system providing the editor and file service module each their own virtual computer. Each virtual computer has a thread that virtualizes the processor. Each virtual computer has a virtual address space that provides each module with the illusion that it has its own memory. To allow communication between virtual computers, the operating system provides SEND, RECEIVE, and a bounded buffer of messages.

- Assume unlimited processors and memory

```
procedure SEND (port, m) {
        p ← ports[port];
        if state of p = FREE then return error;        // If port is not in use, then return error
        success ← FALSE;
        while not success do {
                if (in of p) – (out of p) < N then {          // Is there room in the buffer?
1                       buffer[(in of p) mod N] of p ← m;     // Put message in the buffer
2                       in of p ← in of p + 1;                // Increment in
                        success ← TRUE;
                }
        }
}

procedure RECEIVE (port) {
        p ← ports[port];
        if state of p = FREE then return error;        // If port is not in use, then return error
        success ← FALSE;
        while not success do {
                if out of p < in of p then {                  // Is there anything to receive?
                        m ← buffer[out of p mod N] of p;      // Yes, copy item out of buffer
                        out of p ← out of p + 1;              // Increment out
                        success ← TRUE;
                }
        }
        return m;                                     // Return message to receiver
}
```

**Figure 5–4:** An implementation of a virtual communication link

shared variable must be coordinated. This assumption exemplifies a principle that coordination is simplest when all shared variables follow the rule:

**One-writer rule**

If each variable has only one writer, coordination becomes easier.

# Reduce to the key problem

```
1       procedure FAULTY_ACQUIRE (L) {
2               while L = LOCKED do {          // spin until L is FREE
3                       ;                      // empty iterator, keep retrying the while
4               }
5               L ← LOCKED;                    // the while test failed, got the lock
6       }
```

We must guarantee that ACQUIRE itself is isolated. Once ACQUIRE is an isolated operation, we can isolate arbitrary sequences of operations. This reduction is an example of a technique called *bootstrapping*, which resembles an inductive proof. Bootstrapping means that we look for a systematic way to reduce a general problem (e.g., isolating updates to an arbitrary set of shared variables) to some much-narrower particular version of the same problem (e.g., isolating updates to a single shared lock). We then solve the narrow problem using some specialized method that might work for only that case, because it takes advantage of the specific situation. The general solution then consists of two parts: a method for solving the special case and a method for reducing the general problem to the special case. In the case of ACQUIRE, the solution for the specific problem is either building a special hardware instruction that is itself isolated, or by some extremely careful programming.

```
procedure ACQUIRE (L) {
        R1 ← RSM (L)                   // read and set lock L
        while R1 = LOCKED do {         // was it already locked?
                R1 ← RSM (L)           // yes, do it again, till we see it wasn't
        }
}
```

# Remove assumptions: yield

```
1     procedure SEND (port, m) {
2         p ← ports[port];
3         if state of p = FREE then return error;        // If port is not in use, then return error
4         success ← FALSE;
5         while not success do {
6             ACQUIRE (buffer_lock of p);
7             if (in of p) – (out of p) < N then {        // Is there room in the buffer?
8                 buffer[(in of p) mod N] of p ← m;       // Put message in the buffer
9                 in of p ← in of p + 1;                  // Increment in
10                success ← TRUE;
11            }
12            RELEASE (buffer_lock of p);
13            if not success then YIELD ();
14        }
15    }
```

- Number of threads may be larger than number of processors

# Go deep: remove mysteries

```
shared struct thread {
        integer topstack;                    // value of the stack pointer
        integer state;                       // RUNNABLE or RUNNING
} threadtable[7];

1       procedure YIELD () {
2               ACQUIRE (threadtable_lock);
3               state of threadtable[ID] ← RUNNABLE;// switch state to RUNNABLE
4               topstack of threadtable[ID] ← SP;// save state: store yielding's thread SP
5               do {    // schedule a RUNNABLE thread
6                       ID ← (ID + 1) mod 7;
7               } while state of threadtable[ID].state ≠ RUNNABLE;// skip running threads
8               state of threadtable[ID] ← RUNNING;        // set state to RUNNING
9               SP ← topstack of threadtable[ID];          // dispatch: load SP of next thread;
10              RELEASE (threadtable_lock);
11              return;
12      }
```

- Pseudocode removes thread switching mystery
- Designed on modern assumptions: multiple processors

# Other usages of virtualization

- Virtual storage device: RAID
- Virtual display: window
- Virtual circuit: TCP connection
- Virtual computer: virtual machine

# Papers discussed this spring

- Worse is better
- Therac 25
- Unix time sharing
- X windows
- Eraser
- Map reduce
- Google
- Hints for computer design
- Ethernet
- End-to-end argument
- NATs
- LFS
- ARIES
- Reflections on trust
- Beyond stack smashing
- Witty worm

# One-pager assignment

For background, please read sections 1 and 2 of the Eraser paper. Then read **just sections 1 and 4** of the RaceTrack paper.

For this week's assignment, you are the manager of several hundred programmers building a large software product for your company. The examples on page 11 of the RaceTrack paper cross your desk, and you are disturbed to learn that these race-condition bugs slipped through several years of Microsoft's code reviews, undetected. Despite this disappointing case study (or maybe because of it), you decide to try to prevent your company's program from shipping with concurrency bugs.

In your role as manager, *write a one-page memo to your employees* outlining the rules or steps you will put in place to try to prevent similar bugs from appearing in your company's product. Since the programmers may not jump at your every command, you will also need to *persuade* them that this is the right course of action. You only have one page, so it's best to focus on the most important points and justifications.

# Example one pager

From: Widagdo Setiawan
Sam Madden 11AM 6.033 recitation
Date: March 2, 2006
Subject: Plans for reducing race condition bugs in multithreaded modules

Problems

Multithreaded modules have been used intensively in our software products. Until now, we never had any well-defined rules regarding multithreaded algorithms. However, after reading the RaceTrack paper [1] from Microsoft Research, it is evident that even after years of extensive code review, subtle race condition bugs still reside in both their Visual Studio Library and Common Language Runtime modules. Therefore, to avoid or reduce similar problems in our software products, I have decided to enforce the following rules whenever a thread safe module is constructed.

Rules

1. Use standard lock techniques

   It is apparent that all three bugs described in the RaceTrack paper arose because the developers did not use any standard locking mechanisms to prevent race conditions. However, the standard locking mechanisms in the .NET Framework are known to be very

# Hands-on assignments

- Reality exposure in lieu of full-blown lab
  - Mostly intended for students with no or little experience with systems
- Unix shell, X windows, ping&trace, DNS, LFS benchmarks,  log-based recovery, and X509 certificates
- Each require under an hour of work

This hands-on assignment will give you some experience using a Write Ahead Log (WAL) system. This system corresponds to the WAL scheme described in Chapter 9.C of the course notes. You should carefully read that section before attempting this

```
begin 1
create_account 1 studentA 1000
commit 1
end 1
begin 2
create_account 2 studentB 2000
begin 3
create_account 3 studentC 3000
credit_account 3 studentC 100
debit_account 3 studentA 100
commit 3
show_state
crash
```

Use a text editor to examine the "DB" and "LOG" files and answer the following questions (do not run *wal-sys* again until you have answered these questions):

1) *Wal-sys* displays the current state of the database contents after you type **`show_state`**. Why doesn't the database show *studentB*?

2) When the database recovers, which accounts should be active, and what values should they contain?

3) Can you explain why the "DB" file does not contain a record for *studentC* and contains the pre-debit balance for *studentA*?

# 6.033 Design Project 1: A Fast but Potentially Unreliable File System

## I. Assignment

There are two deliverables for Design Project 1:

1. Two copies of a ***design proposal*** not exceeding 800 words, due on Tuesday, **March 7, 2006**.

2. Two copies of a ***design report*** not exceeding 2,500 words, due on Thursday, **March 23, 2006**.

Your goal is to build a fast file system for a machine that will be used to store files for clients temporarily. One could use such a machine, for example, to cache requested web pages at the edges of the Internet, or to hold files containing data collected from sensors. Clients can use a web server running on the machine to upload, download, and delete files. You can anticipate a range of file sizes from small (e.g., a file with temperature readings) to large (e.g., a file containing a video clip). The web server is the only application running on the machine and the only application your design will need to support.

# Example report

**1. Introduction**

Many file systems are designed to be fast, while maintaining reliability in the face of crashes. This paper presents an alternate design, MEMFiS: A Fast Memory-Backed File System, which optimizes only for speed and not for reliability. To maximize efficiency and simplicity, MEMFiS employs three unusual design decisions:

1. The File Table is stored in memory, which sacrifices a high maximum number of files for a reduced number of disk seeks per system call.
2. The disk is segmented into pre-sized blocks, which sacrifices optimal disk utilization for a non-fragmented disk with performance that degrades minimally over time.
3. The read-ahead caching scheme, which sacrifices throughput for an initial read() request for data availability in the cache for future requests.

MEMFiS is optimized for a file distribution similar to that of a typical desktop computer, as presented in Table 1.

| File size (kilobytes) | % of Total Files Stored | % of Total Disk Space Used |
|---|---|---|
| 0-1 | 25% | 1% |
| 1-10 | 45% | 2% |
| 10-100 | 25% | 7% |
| 100-1000 | 3% | 20% |
| >1000 | 2% | 70% |

Table 1: The table demonstrates the file distribution of my laptop. For example, 25% of files in a typical file distribution are 0-1 kilobyte in size. However, these files only consume 1% of the total disk space. According to this data, the average file size is 200 kilobytes.

# Design project 1: single (2005)

In 6.004, you designed a microprocessor, the brain of a modern computer. But just as your brain would be useless without its nervous system, the Beta would be useless without a way to connect to external devices. Processors need to communicate with the outside world -- through video displays, network cards, and keyboards. It's these devices and others that turn a processor into a useful computer system.

In this project, you will design a *bus*: a way to attach lots of external devices to the Beta at the same time. We call these devices "Beta External Devices," or BEDs. They can be keyboards, hard drives, iPods, video cards, DDR pads, digital cameras, Ethernet cards, mice -- you name it.

Your design will need to specify how the bus works at four levels:

- **(A) Physical Connection.** We specify the one possible wiring configuration between the BEDs and the Beta, but you will need to design the protocol that the Beta uses to exchange data with BEDs and its memory.
- **(B) Identification.** How does the Beta know what BEDs are connected? How does the Beta speak to an individual BED?
- **(C) Initialization.** What does the Beta do when BEDs are added or removed?
- **(D) Application Interface.** How does the operating system (OS) allow software applications to interact with BEDs?

# Design project 2: team

## Design Project 2: Delay-Tolerant Wireless Networking with Cheap Laptops (v2)

*(Note: this is version 2 of the design project. If you printed out a version that does not have this note, please print the current version.)*

### 0. Assignment

You will do design project 2 in teams of three students **from your recitation section.**

There are four deliverables for Design Project 2:

1. A list of team members emailed to your recitation instructor by **April 11, 2006**.
2. One copy of a ***design proposal*** not exceeding 800 words, due on Thursday, **April 27, 2006**.
3. One copy of a ***design report*** not exceeding 5,000 words, due on Thursday, **May 11, 2006**.
4. An in-class presentation on **May 16, 2006**. Details of the presentations will be posted closer to the due date.

6.033 design reports are different from quizzes and problem sets. These projects, like those in real life, are under-specified, and it is your job to complete the specification sensibly, given the project requirements. As with real-world designs, those requirements often need some adjustment as you flesh out your design. We strongly recommend that you start early so that you can iterate your design. A good design will likely take more than a few days.

# Design project 2: requirements

Your protocol should provide the following API:

```
result send_msg(msg, len, dest, app_port);

result register_handler(app_port, handler_function);
```

Applications call `send_msg` to deliver a message, `msg`, of arbitrary length, `len`, to a remote laptop, `dest`. Note that the laptop cannot necessarily communicate directly with `dest` when the application calls `send_msg`. However, your network protocol should provide the following *best-effort* delivery guarantee: it will eventually deliver `msg` to the laptop named `dest` unless a failure occurs during that transmission. (Note that the two applications we have asked you to design have specific reliability requirements; you will need to decide to what extent those requirements are satisfied by your network protocol versus separately by each application.)

# Quiz 1

Alyssa claims that semaphores can also be used to make operations atomic. She proposes the following modification to a *port_info* structure and RECEIVE_MESSAGE to allow threads to concurrently invoke RECEIVE_MESSAGE on the same port without race conditions (only the commented lines changed):

```
structure port_info {
 semaphore n ← 0;
 semaphore mutex ←????;     // see question below
 message buffer[NMSG];
 long integer in ← 0;
 long integer out ← 0;
} port_infos[NPORT];

procedure RECEIVE_MESSAGE(dest_port)
   structure port_info d;
   d ← port_infos[dest_port];
   DOWN(d.mutex);     // enter atomic section
   DOWN(d.n);
   m ← d.buffer[d.out mod NMSG];
   d.out ← d.out + 1;
   UP(d.mutex);     // leave atomic section
   return m;
```

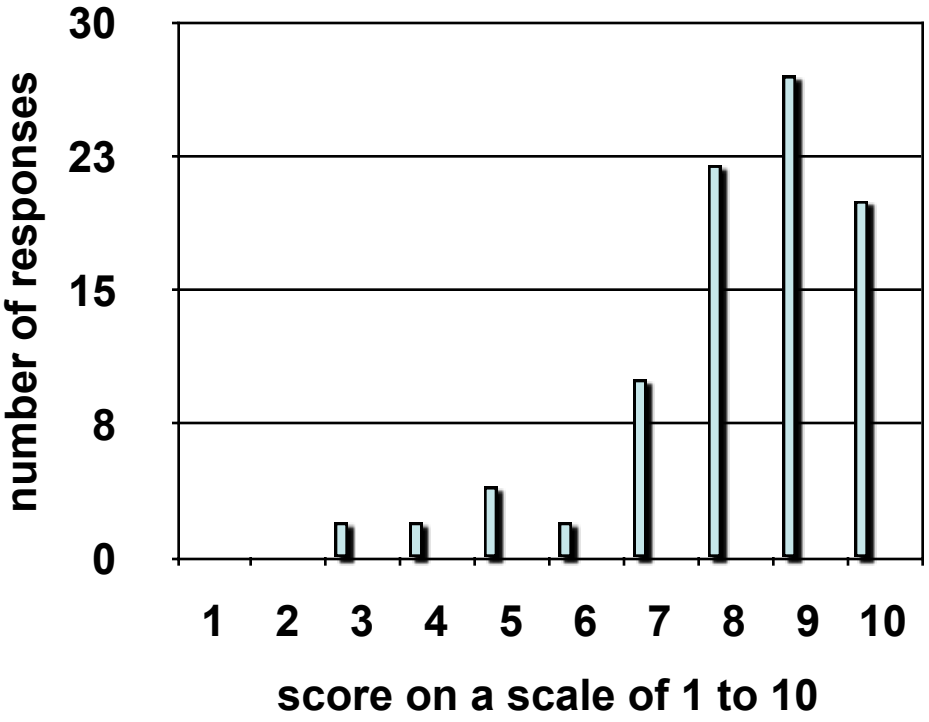
**12. [8 points]:** To what value can *mutex* be initialized to avoid race conditions and deadlocks when multiple threads call RECEIVE_MESSAGE on the same port?

**(Circle True or False for each choice.)**

# Does the approach work?

- Students think so:
  - All MIT EECS students take it, even though it is not required for EE majors
  - Results from a survey 5 years after graduation:
    - Most valuable EECS class
  - Women and minority students enjoy the class
  - A few students outside of EECS take the class
- Instructors think so:
  - They love to teach it
  - Instructors come from AI, Systems, and Theory

# Student feedback (spring 2006)




*"You don't need to know anything about systems before hand"*

*"I was able to answer every question the Google interviewer asked me!"*

# ACM/IEEE 2001 curriculum

- Curriculum has 2 layers:
  1. Modules that constitute appropriate CS education
  2. Suggested packaging

- 6.033: a different packaging of 226c:
  - Operating systems and networks (compressed)
  - Plus: naming, fault tolerance, and both system and cryptographic security

# Incremental adoption

- Use text several quarters/semesters
  - Intro OS course and keep lab
  - Intro networking course
- Use text as intro graduate course
  - Combines well with research papers

# Where do I get the material?

- All material for last 10 years is at:
  http://web.mit.edu/6.033
- A polished version on MIT's Open Course Ware
- Complete draft of text exists
  - Includes extensive problems and solutions chapter
  - Iterated for 30 years in 6.033
  - 3 years experience with current version
  - Externally reviewed
  - Send us email
- Interested in being a test class?

# Summary

- Too many systems classes, too little time, too few principles, too much mechanics
- Alternative: broad intro class, followed by in-depth classes
- Advantages:
  - Broad appeal
  - Focus on design principles and key ideas
  - No programming required, but can be hands-on
- Disadvantage:
  - Curriculum change, but introduction can be incremental