A systems approach to teaching computer systems

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

Massachusetts Institute of Technology
Who are the professor’s customer?

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

*Students!*
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
- Web 2.0
- Multicore

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
- Only a few students will actually program
  - An operating system
  - A parallel computer
  - A database manager
  - A cryptographic protocol
- Many students will use those systems so they need to understand them
  - Plan 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 (Linux, Core 2 Duo, IPv4, SQL, TLS)
  - 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
MIT 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, transactions, transactional memory, logs, rename
• Concurrency control
  – read/write locks, two-phase locking, optimistic
• Performance
  – Caching, scheduling
• Virtualization
  – virtual machine, virtual circuit, 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 fragments
- 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 comm. links
• Naming: glue to connect abstractions
• Client/server: strong modularity
• Operating systems: isolate clients and servers
• Performance: bottlenecks in a pipeline
• Network systems: connect client and servers
• Fault tolerance: modularity to cope with failures
• Transactions: modularity to cope with concurrency and failures
• Consistency: invariants across computers
• Security: modularity to cope with adversaries
Content themes

- Design principles
- The pervasive importance of modularity
  - Abstractions, naming, virtual memory, transactions, secure connections
- Stronger and stronger modularity
  - Accidental, hardware, and malicious failures
- Network centered
  - Client/service, RPC, communication links
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 astonishment
- 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
- Fail-safe defaults
- 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
  - copy an existing interface
- Examples:
  - Virtual memory: address spaces
  - Virtual processors: threads
  - Virtual communication link: pipe, IPC
- Artifact: operating system
Tease ideas apart

Assume unlimited processors and memory

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.
Concurrent problem

```plaintext
shared structure {
    message message[N];   // A shared bounded buffer
    integer in initially 0; // with a maximum of N messages
    integer out initially 0; // Counts number of messages put in the buffer
} buffer;

procedure SEND (buffer reference p, message msg) {
    while p.in - p.out = N do { ; // do nothing // } // If buffer is full, wait for room
    p.message[p.in mod N] ← msg;    // Put message in the buffer
    p.in ← p.in + 1;                // Increment in
}

procedure RECEIVE (buffer reference p) {
    while p.in = p.out do { ; // do nothing // } // If buffer is empty, wait for message
    msg ← p.message[p.out mod N];     // Copy item out of buffer
    p.out ← p.out + 1;                // Increment out
    return msg;                       // Return message to receiver
}
```

Figure 5-5: An implementation of a SEND and RECEIVE using bounded buffers.

• Correctness relies on assumptions, but illustrates one-writer principle
Enforce assumptions

```
shared structure {
    message message[N]; // A shared bounded buffer
    long integer in initially 0; // with a maximum of N messages
    long integer out initially 0; // Counts number of messages put in the buffer
    lock buffer_lock initially UNLOCKED; // Lock to order sender and receiver
}

procedure SEND (buffer reference p, message msg) {
    ACQUIRE (p.buffer_lock);
    while p.in - p.out = N do {
        RELEASE (p.buffer_lock); // While waiting release lock so that RECEIVE
        ACQUIRE (p.buffer_lock); // can remove a message
    }
    p.message[p.in mod N] ← msg; // Put message in the buffer
    p.in ← p.in + 1; // Increment in
    RELEASE (p.buffer_lock);
}
```

• Several senders (and receivers)
Reduce to the key problem

```python
procedure FAULTY_ACQUIRE (L) {
    while L = LOCKED do {
        ;
    } // empty iterator, keep retrying the while
    L ← LOCKED;
    // the while test failed, got the lock
}
```

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.

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

```plaintext
procedure SEND (buffer reference p, message msg) {
    ACQUIRE (p.buffer_lock);
    while p.in – p.out = N do {
        RELEASE (p.buffer_lock);
        YIELD();
        ACQUIRE (p.buffer_lock);
        // Processor came back, maybe there is room
        // Wait loop end, go back to test
        p.message[p.in mod N] ← msg;
        // Put message in the buffer
        p.in ← p.in + 1;
        // Increment in
        RELEASE (p.buffer_lock);
    }
}
```

- Number of threads may be larger than number of processors
Go deep: remove mysteries

- Pseudocode removes thread switching mystery
- Designed on modern assumptions: multiple processors
Other usages of virtualization

<table>
<thead>
<tr>
<th>Virtualization method</th>
<th>Physical resource</th>
<th>Virtual resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiplexing</td>
<td>server</td>
<td>Web site</td>
</tr>
<tr>
<td>multiplexing</td>
<td>processor</td>
<td>thread</td>
</tr>
<tr>
<td>multiplexing</td>
<td>real memory</td>
<td>virtual memory</td>
</tr>
<tr>
<td>multiplexing and emulation</td>
<td>real memory and disk</td>
<td>virtual memory with paging</td>
</tr>
<tr>
<td>multiplexing</td>
<td>wire or communication channel</td>
<td>virtual circuit</td>
</tr>
<tr>
<td>aggregation</td>
<td>communication channel</td>
<td>channel bonding</td>
</tr>
<tr>
<td>aggregation</td>
<td>disk</td>
<td>RAID</td>
</tr>
<tr>
<td>emulation</td>
<td>disk</td>
<td>RAM disk</td>
</tr>
<tr>
<td>emulation</td>
<td>Macintosh</td>
<td>virtual PC</td>
</tr>
</tbody>
</table>

**Figure 5–1:** Examples of virtualization
Papers discussed this spring

- Worse is better
- Therac 25
- Unix time sharing
- X windows
- Eraser
- Map reduce
- Unison
- Hints for computer design

- Ethernet
- End-to-end argument
- NATs
- Wide-area routing
- RAID
- LFS
- 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: Widadgo 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 little or no experience with systems
• Unix shell, X windows, ping&trace, DNS, LFS benchmarks, log-based recovery, and X509 certificates
• Doable in less than 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

```
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
```

Wal-sys should print out the contents of the "DB" and "LOG" files, and then exit.

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.
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.

<table>
<thead>
<tr>
<th>File size (kilobytes)</th>
<th>% of Total Files Stored</th>
<th>% of Total Disk Space Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>25%</td>
<td>1%</td>
</tr>
<tr>
<td>1-10</td>
<td>45%</td>
<td>2%</td>
</tr>
<tr>
<td>10-100</td>
<td>25%</td>
<td>7%</td>
</tr>
<tr>
<td>100-1000</td>
<td>3%</td>
<td>20%</td>
</tr>
<tr>
<td>&gt;1000</td>
<td>2%</td>
<td>70%</td>
</tr>
</tbody>
</table>

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 (2008)

In this project you will design a flash storage system. The system that you design will serve as the core of flash-based storage devices like USB memory sticks and flash memory cards. The following diagram shows the overall structure of a computer connected to a USB memory stick; a connection to an SD card, for example, would look exactly the same except for a different physical interconnect.

The goal of this project is to design the flash controller (the red software component in the first diagram above), which is a non-trivial piece of software. The controller will provide a disk-like API that the USB device driver will call. The controller will access the NAND chip through the API of the flash device driver. The complexity of the flash controller results from the fact that its role is to make the flash look like a magnetic disk drive, even though the physical characteristics of flash chips are very different from those of disk drives.
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:

```c
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.)
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):

```c
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];
```

```c
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.)
Quiz 3

Alice wants to communicate with Bob over an insecure network. She learned about one-time pads in 6.033, and decides to use a one-time pad to secure her communications. Since Alice wants to send a k-bit message to Bob in the future, she generates a random k-bit key \( r \) and hands it to Bob in person.

When Alice comes to send Bob her message, she XORs the message \( m \) with the key \( r \) to produce a cyphertext \( c \), and sends this on the network. Bob XORs \( c \) with \( r \) to retrieve \( m \).

12. [6 points]: Assume that Alice’s message \( m \) is a concatenation of a header followed by some data. Consider an eavesdropper Eve who snoops on Alice’s conversation. If Eve can correctly guess the value of the header in Alice’s message, which of the following are correct?

(Circle ALL that apply)

A. Eve’s ability to decrypt the data bits in \( m \) is not improved by her knowledge of the header bits.
   
   Yes. The one-time pad encrypts each bit independently, hence there is no correlation between the encryption of the data bits and the header bits.

B. The data bits in Alice’s message are confidential.
   
   Yes. Since each bit in \( m \) is XORed with the corresponding bit in the random key, it is impossible to determine the correct decryption without knowledge of the key (or \( m \)).

C. The data bits in Alice’s message are securely authenticated.
   
   No. An active attacker could change bits in \( c \) and the one-time scheme would not alert Bob that he was not reading the information Alice sent.
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 textbook exists
  – Includes extensive problems and solutions chapter
  – Iterated for 40 years in 6.033
  – 5 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