CSE 451: Operating Systems
Winter 2017

Module 1
Course Introduction

Gary Kimura
garyki@cs.washington.edu
Mark Zbikowski
mzbik@cs.washington.edu
Today’s agenda

• Administrivia
  – Course overview
    • course staff
    • general structure
    • the text(s)
    • policies
    • your to-do list

• OS overview
  – Trying to make sense of the topic
Course overview

• Operationally, everything you need to know will be on the course web page: http://www.cs.washington.edu/451/

• Or on the course email and email archive: TBD

• Or on the course discussion board: TBD
But to tide you over for the next hour …

• Course staff
  – Mark Zbikowski
  – Gary Kimura
  – Michael Johnson
  – Ryan McMahon

• General Course Structure
  – Read the text prior to class (really important)
  – Homework exercises to motivate reading by non-saints
  – Sections will focus on projects
  – You're paying for interaction. We lecture for 40+ minutes and we expect YOU to ask questions. If you don’t, we will ask YOU questions.
• The text
• The text
  – Really outstanding – written by current experts
    – Allows you to actually figure out how things work
    – *Way* better (and way less expensive) than any alternative
  — First Second edition – *still* has typos
    – Try not to resent this; help the authors debug it
  – Think of it as helping you to understand, and dig deeper than, the lecture, section, and project material

• Other resources
  – Many online; some of them are essential

• Policies
  – Collaboration vs. cheating
  – Projects: late policy

© 2013 Gribble, Lazowska, Levy, Zahorjan
• Start your projects early
• Projects
  – Start them early
  – Five of them
  – Start them early
  – Teams of two. You’re likely to be happier if you form a team on your own than if we form one for you!
  – Do not start your project late. You will regret it.
  – Start them early
• Projects
  – “JOS” – mock x86 operating system implementation
  – 5 Part project spanning the whole quarter
  – Projects done in groups of 2
    – Catalyst survey out for partner signups
    – BY THIS FRIDAY 11:59pm
  – Projects 1: Booting the JOS kernel and stack
    backtrace/console printing
    – Due next week Wednesday, Jan. 11th
  – Projects 2 – 5: Implementing virtual/physical memory, scheduling, file system … etc
• Your to-do list …
  – Please read the entire course web thoroughly, *today*
  – Be sure you’re on the cse451 email list, and check your email daily
    – You should have received email over the weekend!
    – Be sure your “@uw” email is being forwarded!
  – Please keep up with the reading
  – Homework 1 (reading) is posted on the web *now*
    – Due at the *start of class Friday*
  – Project 1 is posted on the web *now*
    – Will be discussed in section Thursday
    – Due at the end of the day *next Friday*
  – Fill out survey for projects by this Friday!
• Course registration
  – If you’re going to drop, please do it soon!
  – If you want to get into the class, be sure you’ve registered with the advisors
    – *They run the show*
    – *I have a registration sheet here!*
More about 451

• This is really two “linked” classes:
  – A classroom/textbook part (mainly run by Mark)
  – A project part (mainly the TAs and Gary)
• In a perfect world, we would do this as a two-quarter sequence
  – The world isn’t perfect … and CS majors have too many required courses as it is.
• By the end of the course, you’ll see how it all fits together!
  – There will be a lot of work. Do not start projects late. Have you heard that message?
  – You’ll learn a lot, and have a ton of fun
  – In the end, you’ll understand much more deeply how computer systems work
• “There is no magic”
• In this class you will learn:
  – what are the major components of most OS’s?
  – how are the components structured?
  – what are the most important (most common) interfaces?
  – what policies are typically used in an OS?
  – what algorithms are used to implement these policies?

• Philosophy
  – You may not ever build an OS
  – But as a computer scientist or computer engineer you need to understand the foundations
  – Most importantly, operating systems exemplify the sorts of engineering design tradeoffs that you’ll need to make throughout your careers – compromises among and within cost, performance, functionality, complexity, schedule …
  – We want you will love this course!
  – We want you to remember it in 5 years as one that paid off!
What is an Operating System?

• Answers:
  – I don't know.
  – Nobody knows.
  – The book claims to know – read Chapter 1.
  – They’re programs – big hairy programs
    • The Linux source you'll be compiling has over 1.7M lines of C
    • Windows has way, way more... NTFS for Windows 8 was over 800K itself.
What is an Operating System?

• Answers:
  – I don't know.
  – Nobody knows.
  – The book claims to know – read Chapter 1.
  – They’re programs – big hairy programs
    – The Linux source you'll be compiling has over 1.7M lines of C

Okay. What are some goals of an OS?
The traditional picture

- “The OS is everything you don’t need to write in order to run your application”

- This depiction invites you to think of the OS as a library; we’ll see that
  - In some ways, it is:
    - all operations on I/O devices require OS calls (syscalls)
  - In other ways, it isn't:
    - you use the CPU/memory without OS calls
    - it intervenes without having been explicitly called
“Everything you don’t have to write”
What is Windows?
“Everything you don’t have to write”
What is Windows?

DOS
Application

Microsoft Word

Browser

TCP/IP

File system
COM

Installer

Printing

Windows

© John DeTreville, Microsoft Corp.
“Everything you don’t have to write”
What is .NET?
“Everything you don’t have to write”
What is .NET?

Application
Internet

Extensibility
Asynchrony

Device independence
Identity & security

XML

© John DeTreville, Microsoft Corp.
The OS and hardware

• An OS *mediates* programs’ access to hardware resources (*sharing* and *protection*)
  – computation (CPU)
  – volatile storage (memory) and persistent storage (disk, etc.)
  – network communications (TCP/IP stacks, Ethernet cards, etc.)
  – input/output devices (keyboard, display, sound card, etc.)

• The OS *abstracts* hardware into *logical resources* and well-defined *interfaces* to those resources (*ease of use*)
  – processes (CPU, memory)
  – files (disk)
  – programs (sequences of instructions)
  – sockets (network)
The text says an OS is …

- A Referee
  - Mediates resource sharing
- An Illusionist
  - Masks hardware limitations
- Glue
  - Provides common services
Why bother with an OS?

• Application benefits
  – programming simplicity
    • see high-level abstractions (files) instead of low-level hardware details (device registers)
    • abstractions are reusable across many programs
  – portability (across machine configurations or architectures)
    • device independence: 3com card or Intel card?

• User benefits
  – safety
    • program “sees” its own virtual machine, thinks it “owns” the computer
    • OS protects programs from each other
    • OS fairly multiplexes resources across programs
  – efficiency (cost and speed)
    • share one computer across many users
    • concurrent execution of multiple programs
The major OS issues

- **structure**: how is the OS organized?
- **sharing**: how are resources shared across users?
- **naming**: how are resources named (by users, by programs)?
- **protection**: how is one user/program protected from another?
- **security**: how is the integrity of the OS and its resources ensured?
- **performance**: how do we make it all go fast?
- **availability**: can you always access the services you need?
- **reliability**: what happens if something goes wrong (either with hardware or with a program)?
- **extensibility**: can we add new features?
- **communication**: how do programs exchange information, including across a network?
More OS issues...

- **concurrency**: how are parallel activities (computation and I/O) created and controlled?
- **scale**: what happens as demands or resources increase?
- **persistence**: how do you make data last longer than program executions?
- **distribution**: how do multiple computers interact with each other?
- **accounting**: how do we keep track of resource usage, and perhaps charge for it?
- **auditing**: can we reconstruct who did what to whom?

*There are tradeoffs – not right and wrong!*
Hardware/Software Changes with Time

• 1960s: mainframe computers (IBM)
• 1970s: minicomputers (DEC)
• 1980s: microprocessors and workstations (SUN), local-area networking, the Internet
• 1990s: PCs (rise of Microsoft, Intel, Dell), the Web
• 2000s:
  – Internet Services / Clusters (Amazon)
  – General Cloud Computing (Google, Amazon, Microsoft)
  – Mobile/ubiquitous/embedded computing (iPod, iPhone, iPad, Android)
• 2010s: sensor networks, “data-intensive computing,” computers and the physical world (“pervasive computing”)
• 2020: it’s up to you!!
Progression of concepts and form factors
Has it all been discovered?

• New challenges constantly arise
  – embedded computing (e.g., iPod)
  – sensor networks (very low power, memory, etc.)
  – peer-to-peer systems
  – ad hoc networking
  – scalable server farm design and management (e.g., Google)
  – software for utilizing huge clusters (e.g., MapReduce, Bigtable)
  – overlay networks (e.g., PlanetLab)
  – worm fingerprinting
  – finding bugs in system code (e.g., model checking)

• Old problems constantly re-define themselves
  – the evolution of smart phones recapitulated the evolution of PCs, which had recapitulated the evolution of minicomputers, which had recapitulated the evolution of mainframes
  – but the ubiquity of PCs re-defined the issues in protection and security, as phones are doing once again
Protection and security as an example

- none
- OS from my program
- your program from my program
- my program from my program
- access by intruding individuals
- access by intruding programs
- denial of service
- distributed denial of service
- spoofing
- spam
- worms
- viruses
- stuff you download and run knowingly (bugs, trojan horses)
- stuff you download and run obliviously (cookies, spyware)
New Agawi Study Says Apple’s iPhone 5 Has Fastest Response Time

By David Murphy  September 21, 2013 00:23pm EST  14 Comments

Agawi TouchMarks I: Minimum App Response Times (MART**) for smartphones
(Lower numbers are better)

<table>
<thead>
<tr>
<th>Device</th>
<th>Time in Milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>iPhone 5 (iOS)</td>
<td>55</td>
</tr>
<tr>
<td>iPhone 4 (iOS)</td>
<td>85</td>
</tr>
<tr>
<td>Galaxy S4 (Android)</td>
<td>114</td>
</tr>
<tr>
<td>Lumia 928 (Windows)</td>
<td>117</td>
</tr>
<tr>
<td>HTC One (Android)</td>
<td>121</td>
</tr>
<tr>
<td>Moto X (Android)</td>
<td>123</td>
</tr>
</tbody>
</table>

© 2013 Gribble, Lazowska, Levy, Zahorjan
An OS history lesson

- Operating systems are the result of a 60 year long evolutionary process.
- We'll follow a bit of their evolution
- That should help make clear what some of their functions are, and why
In the Beginning...

• 1943
  – T.J. Watson (created IBM):
    “I think there is a world market for maybe five computers.”

• Fast forward … 1950
  – There are maybe 20 computers in the world
    • They were unbelievably expensive
    • Imagine this: machine time is more valuable than person time!
    • Ergo: efficient use of the hardware is paramount
  – Operating systems are born
    • They carry with them the vestiges of these ancient forces
The Primordial Computer
The OS as a linked library

• In the very beginning…
  – OS was just a library of code that you linked into your program; programs were loaded in their entirety into memory, and executed
    • “OS” had an “API” that let you control the disk, control the printer, etc.
  – Interfaces were literally switches and blinking lights
  – When you were done running your program, you’d leave and turn the computer over to the next person

• Recapitulation: Paul Allen writing a bootstrap loader for the Altair as the plane was landing in New Mexico
Asynchronous I/O

• The disk was really slow
• Add hardware so that the disk could operate without tying up the CPU
  – Disk controller
• Hotshot programmers could now write code that:
  – Starts an I/O
  – Goes off and does some computing
  – Checks if the I/O is done at some later time
• Upside
  – Helps increase (expensive) CPU utilization
• Downsides
  – It's hard to get right
  – The benefits are job specific
The OS as a “resident monitor”

• Everyone was using the same library of code
• Why not keep it in memory?

• While we’re at it, make it capable of loading Program 4 while running Program 3 and printing the output of Program 2
  – SPOOLing – Simultaneous Peripheral Operations On-Line

• What new requirements does this impose?
Multiprogramming

• To further increase system utilization, multiprogramming OSs were invented
  – keeps multiple runnable jobs loaded in memory at once
  – overlaps I/O of one job with computing of another
    • while one job waits for I/O completion, another job uses the CPU
  – Can get rid of asynchronous I/O within individual jobs
    • Life of application programmer becomes simpler; only the OS programmer needs to deal with asynchronous events
  – How do we tell when devices are done?
    • Interrupts
    • Polling

  – What new requirements does this impose?
(An aside on protection)

• Applications/programs/jobs execute directly on the CPU, but cannot touch anything except “their own memory” without OS intervention
(An aside on concurrency)

- Transistor density continues to increase (Moore’s Law), but individual cores aren’t getting faster – instead, we’re getting more of them (the number doubles on roughly the old 18-month cycle)
• The burden is on the programmer to use an ever increasing number of cores

• A lot of this course is about concurrency
  – It used to be a bit esoteric
  – It has now become one of the most important things you'll learn (in any of our courses)
Timesharing

• To support interactive use, create a timesharing OS:
  – multiple terminals into one machine
  – each user has illusion of entire machine to him/herself
  – optimize response time, perhaps at the cost of throughput

• Timeslicing
  – divide CPU equally among the users
  – if job is truly interactive (e.g., editor), then can jump between programs and users faster than users can generate load
  – permits users to interactively view, edit, debug running programs
• MIT CTSS system (operational 1961) was among the first timesharing systems
  – only one user memory-resident at a time (32KB memory!)
• MIT Multics system (operational 1968) was the first large timeshared system
  – nearly all OS concepts can be traced back to Multics!
  – “second system syndrome”
• CTSS as an illustration of architectural and OS functionality requirements
• In early 1980s, a single timeshared VAX-11/780 (like the one in the Allen Center atrium) ran computing for all of CSE.

• A typical VAX-11/780 was 1 MIPS (1 MHz) and had 1MB of RAM and 100MB of disk.
  – An Apple iPhone 5s (A7 processor) is 1.3GHz dual-core (x2600), has 2GB of RAM (x2000), 64GB of flash (x640), a quad-core GPU (unheard of).
Parallel systems

• Some applications can be written as multiple parallel threads or processes
  – can speed up the execution by running multiple threads/processes simultaneously on multiple CPUs [Burroughs D825, 1962]
  – need OS and language primitives for dividing program into multiple parallel activities
  – need OS primitives for fast communication among activities
    • degree of speedup dictated by communication/computation ratio
  – many flavors of parallel computers today
    • SMPs (symmetric multi-processors)
    • MPPs (massively parallel processors)
    • NOWs (networks of workstations)
    • Massive clusters (Google, Amazon.com, Microsoft)
    • Computational grid (SETI @home)
Personal computing

• Primary goal was to enable new kinds of applications
  • Bit mapped display [Xerox Alto,1973]
    – new classes of applications
    – new input device (the mouse)
• Move computing near the display
  – why?
• Window systems
  – the display as a managed resource
• Local area networks [Ethernet]
  – why?
• Effect on OS?
Distributed OS

- Distributed systems to facilitate use of geographically distributed resources
  - workstations on a LAN
  - servers across the Internet
- Supports communications between programs
  - interprocess communication
    - message passing, shared memory
  - networking stacks
- Sharing of distributed resources (hardware, software)
  - load balancing, authentication and access control, …
- Speedup isn’t the issue
  - access to diversity of resources is goal
Client/server computing

- Mail server/service
- File server/service
- Print server/service
- Compute server/service
- Game server/service
- Music server/service
- Web server/service
- etc.
Peer-to-peer (p2p) systems

• Napster
• Gnutella
  – example technical challenge: self-organizing overlay network
  – technical advantage of Gnutella?
  – er … legal advantage of Gnutella?
Embedded/mobile/pervasive computing

- Pervasive computing
  - cheap processors embedded everywhere
  - how many are on your body now? in your car?
  - cell phones, PDAs, network computers, …
- Often constrained hardware resources
  - slow processors
  - small amount of memory
  - no disk
  - often only one dedicated application
  - limited power
- But this is changing rapidly!
  - cf. specs of iPhone 5S earlier!
Ad hoc networking
The major OS issues

- **structure**: how is the OS organized?
- **sharing**: how are resources shared across users?
- **naming**: how are resources named (by users, by programs)?
- **protection**: how is one user/program protected from another?
- **security**: how is the integrity of the OS and its resources ensured?
- **performance**: how do we make it all go fast?
- **availability**: can you always access the services you need?
- **reliability**: what happens if something goes wrong (either with hardware or with a program)?
- **extensibility**: can we add new features?
- **communication**: how do programs exchange information, including across a network?
More OS issues…

• **concurrency**: how are parallel activities (computation and I/O) created and controlled?
• **scale**: what happens as demands or resources increase?
• **persistence**: how do you make data last longer than program executions?
• **distribution**: how do multiple computers interact with each other?
• **accounting**: how do we keep track of resource usage, and perhaps charge for it?
• **auditing**: can we reconstruct who did what to whom?

*There are tradeoffs – not right and wrong!*