Operating Systems COT 4600 – Fall 2009

Dan C. Marinescu

Office: HEC 439 B

Office hours: Tu, Th 3:00-4:00 PM

Class organization

- Class webpage:
- http://www.cs.ucf.edu/~dcm/Teaching/OperatingSystemsCOT4600/ClassIndex.html
- Textbook:
  - ``Principles of Computer Systems Design; An Introduction'' by Jerome Saltzer and Frans Kaasohoek. Publisher: Morgan Kaufmann, ISBN 978-0-12-374957-4.

The textbook has 6 chapters

- Systems
- Elements of Computer System Organization
- The Design of Naming Schemes
- Enforcing Modularity with Clients and Services
- Enforcing Modularity with Virtualization
- Performance

Class revision

- We have revised the COT 4600 class for the Fall 2009 semester to give the students a fresh look at basic principles guiding the design and implementation of operating systems.
- The focus of the class is switched from the discussion on ``how'' operating systems are implemented to the identification of the most important questions the designer of an operating system has to address and ``why'' a solution is better than others.

Class revision (cont'd)

- Another major departure from the more traditional approach in covering operating systems is the emphasize on performance; several lectures cover computer system performance analysis.
- We also emphasize the ``big picture'' the relationship of operating systems with other subjects from undergraduate curriculum including:
  - computer architecture,
  - programming languages,
  - algorithms,
  - networking,
  - databases,
  - modeling and performance analysis.

Assignments

- There are 6 homework assignments and a class project.
- A <u>homework</u> consists of 3-5 problems at the end of each chapter in the textbook
- The <u>class project:</u> simulate the operation of a simple kernel for a computer system. It involves multiple phases:
  - Simulate a processor with a minimal instruction set operating in kernel and user mode. Due week 4.
  - Virtualize the memory. Design and implement a paging system and a virtual memory manager. Due week 8.
  - Virtualize the processor. Add a thread management system. Due week 10.
  - Add a virtual communication channel allowing threads to communicate using a bounded buffer and *send* and *receive* primitives. Due week 14.

Grading

- Homework: 15%
- Project: 35%
- Midterm: 20%
- Final: 30%

Lecture 1

- Today:
  - Systems and Complexity
  - Sources of Complexity
  - Modularity, Abstractions, Layering, Hierarchy
- Next time
  - Names
  - Complexity of Computer Systems

Man-made systems

- Basic requirements for man-made systems:
  - Functionality
  - Performance
  - Cost
- All systems are physical ➔ the laws of physics governing the functioning of any system must be well understood.
- Physical resources are limitted.

Complex systems

- Large number of components
- Large number of interconnections
- Many irregularities
- Long description
- For man-made systems: a team of designers, implementers, and maintainers.

Issues faced by the designer of a complex system

- Emerging Properties
- Propagation of effects
- Incommensurate scaling
- Tradeoffs

Emerging properties

- A characteristic of complex systems → properties that are not evident in the individual components but show up when the components interact with one another.
- Example: you have several electronic components which radiate electromagnetic energy; if they are too close to one another their function are affected.

How the nature deals with complexity

- For biological systems: symmetry, construction of complex biological structures from building blocks.
- Self-organization ➔though difficult to define, its intuitive meaning is reflected in the observation made by Alan Turing that ``global order can arise from local interactions''
- Scale-free systems. Each component interacts directly only with a small number of other components.
- Man-made systems to imitate nature!!

Scale-free systems

- The scale-free organization can be best explained in terms of the network model of the system, a random graph with vertices representing the entities and the links representing the relationships among them.
- In a scale-free organization, the probability *P(m)* that a vertex interacts with m other vertices decays as a power law:

$$P(m) \approx m^{-d}$$

with *d* a positive real number, regardless of the type and function of the system, the identity of its constituents, and the relationships between them.

Examples of self-organization

- The collaborative graph of movie actors where links are present if two actors were ever cast in the same movie; in this case d=2.
- The power grid of the Western US has some 5,000 vertices representing power generating stations; in this case d=4.
- The World Wide Web, d=2.1. This means that the probability that m pages point to one page is

$$P(m) = m^{-2.1}$$

- The citation of scientific papers d=3.

Propagation of effects

- In a complex system:
  - Changes of one component affect many other components. Example, changing the size of the tire of a car.
  - A problem affecting one component propagates to others. For example, the collapse of the housing industry in the Us affected the economy of virtually all countries in the world.

Incommensurate scaling

- Not all components of a complex system follow the same scaling rules. Examples:
  - The pyramids
  - The tankers
- The power dissipation increases as $(\text{clock rate})^3$. If you double the clock rate, then the power dissipation increases by a factor of 8 so you need a heat removal system 8 times more powerful.

Trade-offs

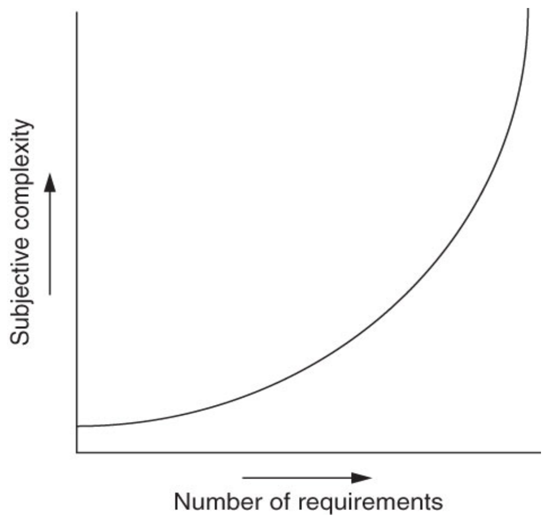
- Many tradeoffs are involved in the design of any system
- Examples:
  - a network switch ➔ what should be done in hardware and what should be done in software
  - a hybrid car with a gas and an electric engine ➔ how powerful should the gas engine be
  - a spam filter ➔ where to set the threshold

Systems and the environment

- System ➔ a set of interconnected components that has a an expected behavior observed at the interface with its environment
- The environment ➔ a critical component to be considered in the design of any system

Two sources of complexity

1. Cascading and interacting requirements
   - 1.1 When the number of requirements grows then the number of exception grows.
   - 1.2 The principle of escalating complexity:

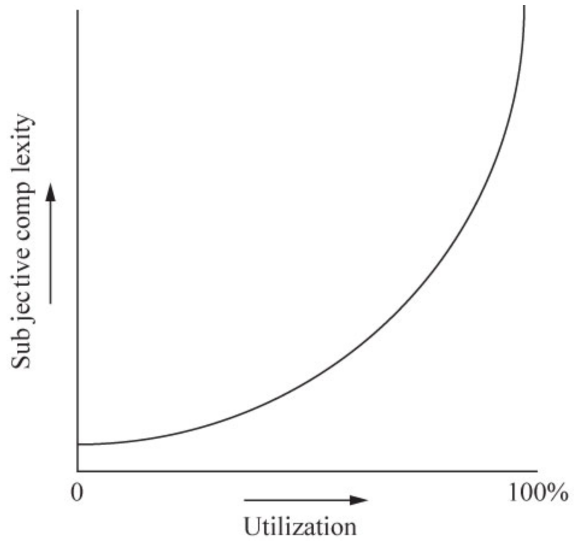


Two sources of complexity (cont'd)

- 1.3 Meeting many requirements with a single design ➔ the need for generality. Advice: avoid excessive generality.
- 1.4 Requirement changes:
  - Example: the electric car produced by Tesla.

Two sources of complexity (cont'd)

- 2. High performance
  - 2.1 Every system must satisfy performance standards.
  - 2.2 The law of diminishing return ➔the more one improves one performance metrics the more effort the next improvement will require



Modularity for Coping with Complexity

- Why does modularity reduce complexity ➔we can focus on the interaction within one module/component.
- Example: assume that:
  - B - the # of bugs in a program is proportional with N , the number of statements
  - T- the time to debug a program is proportional with N x B thus it is proportional with N^2
  - Now we divide the program in K modules each with N/K statements each:
    - The time to debug a module is proportional with $(N/K)^2$
    - The time to debug the K modules is K x $(N/K)^2 = N^2/K$
    - We have reduced the time by a factor of K. Is that so?

Abstractions for Coping with Complexit

- Abstraction➔ separation of the
  - interface from the internals or
  - specification from implementation

  Example: you do not need to know how the engine of your car works in order to drive the car
- Why abstractions reduce complexity ➔ because they minimize the interconnections between components.
- Observations:
  - Best division usually follows natural boundaries.
  - The goal is defeated by unintentional or accidental interconnections among components.

More about abstractions

- Abstractions are critical for understanding critical phenomena. Think about the abstract model of computation provided by the Turing Machine.
- Do not be carried away by abstractions. For example, often software designers think about an abstract computer and are not concerned about the physical resources available to them. E.g., the small display of a wireless phone.

More about abstractions (cont'd)

- The robustness principle ➔ be tolerant of inputs and strict on outputs.

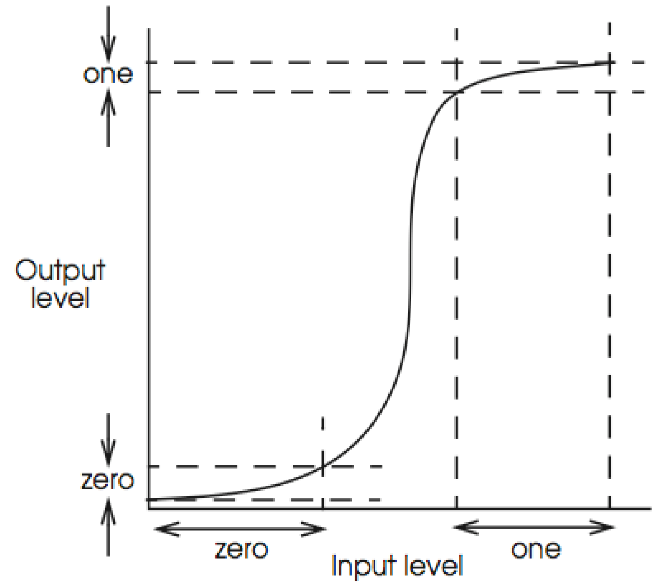


Figure 1–3: How gain and non-linearity of a digital component restore levels. The range of accepted inputs is much wider than the range of generated outputs.

More about abstractions (cont'd)

- The safety margin principle ➔ Keep track of the safty margin of the cliff or you may fall over edge!!

Layering for Coping with Complexit

- Layering → building a set of successive functional entities with restricted communication patterns, a layer may only communicate with the layer below it and with the one above it.
- Examples: networking

Hierarchy for Coping with Complexity

- Hierarchical structures → construct a large system from a small collection of relatively large subsystems
- Examples:
  - Corporations
  - An army
  - A computer is a collection of subsystems