Operating Systems Principles

1. Introduction

We build increasingly large amounts of increasingly complex software. Size and complexity create numerous problems:

- increasingly sophisticated software means that the typical project involves multiple orders of magnitude more code than projects of 20-30 years ago.
- increasing complexity means that the work involved in creating a piece of software is much worse than linear with the number of lines of code involved.
- increasing complexity makes it very difficult to learn and fully understand our software, which leads to many more errors in its construction.
- systems are now assembled from numerous independently developed and delivered pieces, and small changes in one component often result in failures of other components.
- as the number of independent interacting components increases, we begin to see complex emergent behaviors that could not have been predicted from our experience with smaller systems.
- increasingly rich functionality and interactions with increasingly many other external systems make complete testing combinatorically impossible.
- our increasing dependency on this software means that failures will cause more severe problems, for more people, more often.

We need tools and techniques to tame this complexity. Without such tools, we cannot deliver today's software ... much less tomorrow's.

Operating systems have long been among the largest and complex pieces of software:

- They involve complex interactions among many sub-systems.
- They involve numerous asynchronous interactions and externally originated events.
- They involve the sharing of stateful resources among cooperating parallel processes.
- They involve coordinated actions among heterogenous components in large distributed systems.
- They must evolve to meet ever-changing requirements.
- They must be able to run, without error, for years, despite the regular failures of all components.
- They must be portable to virtually all computer architectures and able to function in heterogenous environments (with many different versions of many different implementations, on many different platforms).

Few of these problems are unique to operating systems, but operating systems encountered them sooner and more intensely than other software domains. As a result, these problems have come to be best understood and solutions developed in the context of Operating Systems. When evolving disciplines encounter these problems, they commonly go back to see how they have been characterized and solved within operating systems.

2. Complexity Management Principles

Software Engineering has developed numerous tools to help us manage the complexity of large software projects. Many of these are software development and project management techniques, but there are also several fundamental design principles.

2.1 Layered Structure and Hierarchical Decomposition

A complex system can easily have too many components and interactions to be held in a single mind, at a single time. To make the whole system (or even individual components) understandable, we must break the system down into smaller components *that can be understood individually*.

Hierarchical decomposition is the process of decomposing a system in a top-down fashion. If we wanted to describe the University of California, we might

- Start by dividing it into the Regents, the Office of the President, and the campuses.
- The campuses can be divided into ten university campuses (UCLA, UCSB, ...) and three laboratories (Livermore, Los Alamos, ...)
- One campus (e.g. UCLA) can be divided into administration and schools (Letters and Sciences, Engineering, Medicine, ...)
- Administration can be divided into departments (personnel, legal, finance, registrar, deans, ...)
- Each School (e.g. Engineering) can also be divided into departments (Computer Science, Materials, ...)
- A department can be divided into administrative staff, support staff, and faculty
- Administrative staff can be divided into responsibility areas (payroll, purchasing, personnel, ...) ...

At each level we can talk about the mission of each group and interactions with other groups. In this way we are understanding a huge University system one layer at a time. We can also pick a particular group and look at (decompose it into) the sub-groups that comprise it. In this way we can develop an understanding of that one group, without having to understand the internal structure of all the groups with which it interacts. Both of these approaches give us the opportunity to develop (some) understanding of the University in small installments.

Hierarchical decomposition is not merely a technique for studying existing systems, but also a principle for designing new systems. If every person in any part of the University system interacted regularly with every other person in every other part, the processes would be impossible to understand or manage. But if we can devise a structure where, at each level, there is a clear division of responsibilities among components, and most functions can be carried out within a particular component, the responsibilities and relationships are much simplified, and it becomes possible to understand, and to manage the system.

Hierarchical structure is a very effective approach to designing or understanding complex systems. But the operation of these systems is not always strictly hierarchical. If we drill down into the details, we see that not all interaction is constrained to follow the tree. Course planning involves interactions between departmental administration and the registrar's office. Different departments or schools often work together to create programs and events in areas of shared interest. The Academic Senate is comprised of department faculty, but their decisions may have implications for the entire system. Such cross-communication is common in hierarchically structured systems. But, as we will see below, there may be advantages to trying to minimize non-hierarchical communication.

This principle has been described in terms of a University system. But it is highly applicable to complex

software systems. In this course we will study and observe the hierarchical design of a complex modern operating system.

2.2 Modularity and Functional Encapsulation

Taking a few million people and arbitrarily assigning them to groups of ten, and then arbitrarily assigning those groups into super-groups of one hundred ... would not be sufficient to make the system understandable or manageable. We must also require that:

- Each group/component (at each level) have a coherent purpose.
- Most functions (for which a particular group is responsible) can be performed entirely within that group/component.
- The union of the groups/component (within each super-group/system) is able to achieve the system's purpose.

These requirements are the difference between an arbitrary decomposition and a hierarchical decomposition. Such a decomposition makes it possible to opaquely encapsulate the internal operation of a sub-group/component, and to view it only in terms of its external interfaces (the services it performs for other groups/components). This enables us to look a group in two very different ways:

- We can examine its responsibilities, and its role in the system of which it is a part.
- We can examine the internal structure and operating rules by which it fulfills its responsibilities.

As long as a component is able to effectively fulfill its responsibilities, external managers and clients can ignore the internal structure and operating rules. Moreover, it should be possible to change the internal structure and operating rules without impacting external clients and managers. When it is possible to understand and use the functions of a component without understanding its internal structure, we call that component a *module* and say that its implementation details are *encapsulated* within that module. When a system is designed so that all of its components have this characteristic, we say that the system design is *modular*.

We can go a little further in defining the characteristics of good modularity:

- There are numerous ways to break up the responsibilities of a system. Smaller and simpler components are easier to understand and manage than larger and more complex components. Thus, there are clear benefits to dividing a system into a larger number of smaller and simpler components. Going back to the university example, even though the University has over a thousand departments and a hundred-thousand employees, it is not particularly difficult to understand the role and responsibilities of any particular department or administrator.

- Many functions are closely related, typically because they perform related operations on the same resources. If operations on a single resource (e.g. deposits and withdrawals in a bank account) are implemented in separate components, there is a danger that a change in one component might (through unintended *side effects*) break another component. Thus it is a good idea to combine closely related functionality (operations embodying common understandings, or with a high likelihood of affecting one-another) into a single module. Combining this with the previous characteristic: we want the smallest possible modules consistent with the co-location of closely related functionality. This characteristic of modular design is called *cohesion*, and a module that exhibits this characteristic is said

to be *cohesive*.

- There are numerous ways to apportion the functionality of a system among its components. If most operations can be accomplished entirely within a single component, they are likely to be accomplished more efficiently. If many operations involve exchanges of services between components:
  - the overhead of communication between components may reduce system efficiency.
  - the increased number of interfaces to service those inter-component requests increases the complexity of the system and the opportunity for misunderstandings.
  - increased dependencies between components increase the likelihood of errors or misunderstandings.

  Thus, the scope of component responsibilities must be designed with an eye towards how well it will be possible to compartmentalize operations within that component. This is another type of *cohesion*. As observed above, there are surely many interactions between various administrators, but most of the day-to-day operations and communications for the personnel administrator in the UCLA Computer Science department are with other people in that department.

In this course, we will observe the way in which a modern operating system is structured and how responsibility is apportioned among sub-systems and plug-in components.

2.3 Appropriately Abstracted Interfaces and Information Hiding

There are numerous ways to specify the interface for each piece of component functionality. They are not all equally good. It seems natural (for an implementer) to define interfaces that mate easily with an intended implementation. A faucet, for instance, delivers water at a rate and temperature that is controlled by a chosen mixture of hot and cold water. So it is that, for many years, most faucets have had two valves (for hot and cold water). While this interface is very easy for plumbers to hook up, it is not a particularly good one:

- I, as a user, want a certain flow of water at a certain temperature. I do not want to guess at the (changing over time) amounts of hot and cold water that will provide it for me. An interface that enables a client to specify parameters that are most meaningful to them and easily get the desired results is said to embody *appropriate abstraction*. An interface that does not provide a client with what they want is said to be *poorly abstracted*.

- The two-valve interface is tightly coupled to a hot- and cold-water supply implementation. If water temperature was controlled by a local flash-heater (or flash-cooler), a two-valve set of controls would be both awkward and inefficient to implement. An appropriately abstracted interface that does not reveal the underlying implementation is said to be *opaque* and to exhibit good *information hiding* (not exposing implementation details that clients did not want to be forced to understand).

Two separate controls for temperature and flow rate would more *opaquely encapsulate* the water delivery mechanisms. Changing the temperature control from a simple clockwise-hot knob to a temperature-calibrated dial would create a much more convenient control abstraction for the intended uses. Such an interface better serves the clients, and provides greater flexibility to the implementers.

Obviously a better abstracted interface is easier for the intended clients. But even if the internal implementation was well suited to the intended clients, exposing implementation details would:

a. expose more complexity to the client, making the interface more difficult to describe/learn.

b. limit the provider's flexibility to change the interface in the future, because the previously exposed interface included many aspects of the implementation. A different implementation would expose clients to an incompatible interface.

In this course, we will examine many fundamental abstractions of software services and resources, how well those interfaces serve their clients and the need for evolution.

2.4 Powerful Abstractions

If every problem has to be solved from scratch, every problem will be hard to solve, and progress will be slow and difficult. But if we can draw on a library of powerful tools, we may find that most problems are easily solved by existing tools. Most of classical mechanics is based on a few basic machines (wheel, lever, inclined plane, screw, pulley, ...) and tools (hammer, chisel, saw, drill, ...). These are not merely artifacts we can employ to build new inventions; They actually set the terms in which we visualize our problems. A new technique or tool (e.g. constructive fabrication with 3D-printers) can radically alter the way we approach problems and (perhaps more importantly) the problems we can solve.

Unlike basic machines and tools (which we can observe in nature), virtually all of the tools and resources in an operating system are abstract concepts from the imaginations of system architects. As operating systems have evolved, we continue to devise more powerful abstractions. A powerful abstraction is one that can be profitably applied to many situations:

- common paradigms (e.g. lock granularity, cache coherency, bottlenecks) that enable us to more easily understand a wide range of phenomena.
- common architectures (e.g. federations of plug-in modules treating all data sources as files) that can be used as fundamental models for new solutions.
- common mechanisms (e.g. registries, remote procedure calls) in terms of which we can visualize and construct solutions

Sufficiently powerful abstractions give us tools to understand, organize and control systems which would otherwise be intractably complex. In this course we will study the emergent phenomena in large and complex systems, concepts in terms of which those phenomena can be understood, and abstractions in terms of which they can be managed.

2.5 Interface Contracts

A complex system is composed of many interacting components. Hierarchical decomposition and modular information hiding moves our attention from implementations to interfaces. Every system, sub-system, or component is primarily understood in terms of the services it provides and the interfaces through which it provides them. An interface specification is a contract: a promise to deliver specific results, in specific forms, in response to requests in specific forms. If the service-providing component operates according to the interface specification, and the clients ensure that all use is within the scope of the interface specifications, then the system should work, no matter what changes are made to the individual component implementations.

These contracts do not merely specify what we will do today, but they represent a commitment for what will be done in the future. This is particularly important in large, complex systems like an operating system. An operating system will be used for a relatively long time, and is assembled from thousands of components

developed by thousands of people, most of whom are operating independently from one-another. Technologies and requirements are continuously evolving and component implementations change continuously in response. If a new version of a service-providing component does not fully comply with its interface specifications (e.g. we decide to change the order of the parameters to the *open(2)* system call), or a new client uses a component in an out-of-spec way (e.g. exploiting an undocumented feature), problems are likely to arise, in the future, perhaps resulting in a system failure.

It both logistically and combinatorically infeasible for every developer to test every change with every combination of components (and component versions) from every other developer. Interface contracts are our first line of defense against incompatible changes. As long as component evolution continues to honor long-established interface contracts, the system should continue to function despite the continuous independent evolution of its constituent components.

In this course will examine a variety of interfaces, interface standards and approaches for managing upwards compatible evolution.

2.6 Progressive Refinement

Linux did not start out as the fully-featured, all-platform system that it currently is. It started out as an attempt to complement the Gnu tools with a very humble re-implementation of the most basic UNIX kernel functionality. All of the wonderful features we enjoy today were added incrementally. Software projects that attempt to to grand things, starting from a clean slate, often fail after spending many years without delivering useful results.

- It is very difficult to estimate the work in a large project.
- It is very difficult to anticipate the problems in a large project.
- It is very difficult to get the requirements right in a large project.
- They often take so long that they are obsolete before they are finished.
- They often lose support before they are finished.

Religious wars abound in Software Engineering, but most modern methodologies now seem to embrace some form of iterative or incremental development:

- Add new functionality in smaller, one feature at a time, projects. It is much easier to identify likely problems (and solutions) in a smaller project, and hence to estimate the required work and likely delivery date. Also the work can be done by a smaller team with better communication, and so is likely to be done more efficiently.
- Rather than pursue large speculative (if you build it they will come) projects, identify specific users with specific needs, and build something that addresses those needs (while moving us in the right direction). Having a specific problem to solve leads to better requirements. Delivering needed functionality to actual users yields creates more value more quickly.
- Deliver new functionality as quickly as possible to get feedback before moving on to the next step. Most software requirements are speculative. The best way to clarify them is to deliver something and get feedback.
- It is much easier to plan the next step after we have data from the previous step. Real performance data or user feedback will guide us towards the most important improvements. If we are going down the wrong path, it is best that we find that out as soon as possible.

3. Architectural Paradigms

Over many decades of building ever more powerful operating systems we have found a few very powerful concepts of very broad applicability.

3.1 Mechanism/Policy Separation

The basic concept is that the mechanisms for managing resources should not dictate or greatly constrain the policies according to which those resources are used. The primary motivation for this principle is that a system is likely to be used used for a long time, in many places, environments and ways that the designers never anticipated. A single resource management strategy that made sense to the designer might prove totally inappropriate for future users. As such resource managers should be designed in two logical parts:

- The *mechanisms* that keep track of resources and give/revoke client access to them.
- A configurable or plug-in *policy* engine that controls which clients get which resources when.

A good example of mechanism/policy separation can be seen in card-key lock systems:

- each door has a card-key reader and a computer-controlled lock.
- each user is issued a personal card-key.
- to get access to a room, a user swipes a card-key past the reader.
- a controlling computer will lookup the registered owner of the card-key in an access control database in order to decide whether or not that user should be granted access to that door at that time, and (if appropriate) send an unlock signal to the associated lock.

The physical *mechanisms* are the card-keys, readers, locks, and controlling computer. The resource allocation *policy* is represented by rules in the access control data-base, which can be configured to support a wide range of access policies. The mechanisms impose very few constraints (beyond those of the rule language) on who should be allowed to enter which rooms when.

And, because the policy is independent from the mechanisms, we could also move to a very different mechanism implementation (e.g. RFID tags), and continue to support the exact same access policies. It should be possible to change either mechanism or policy without greatly impacting the other.

Mechanism/policy separation as a design principle that guides us to build systems that will be usable in a wide range of future contexts. In this course we will look at a few examples of services whose designs exhibit this quality.

3.2 Indirection, Federation, and Deferred Binding

Powerful unifying abstractions often give rise to general object classes with multiple implementations. Consider a file system: a collection of files that can be opened, read and written. There can be many different types of file systems (e.g. DOS/FAT file systems on SD cards, ISO 9660 file systems on CD ROMs, Windows NTFS disks, Linux EXT3 disks, etc), but they all implement similar functionality. One could write an operating system that understood all possible file system formats, but the number and range file system formats and rate of evolution dooms that approach to failure. Realistically, we need a way to add support for new file systems independently from the operating system to which they will be connected.

Many services (e.g. device drivers, video codecs, browser plug-ins) have similar needs. The most common way to accommodate multiple implementations of similar functionality is with plug-in modules that can be added to the system as needed. Such implementations typically share a few features:

- A common abstraction (class interface) to which all plug-in modules adhere.
- The implementations are not built-in to the operating system, but accessed *indirectly* (e.g. through a pointer to an specific object instance).
- The indirection is often achieved through some sort of *federation framework* that registers available implementations and enables clients to select the desired implementation, and automatically routes all future requests to the selected object/implementation.
- The *binding* of a client to a particular implementation does not happen when the software is built but rather is *deferred* until the client actually needs to access a particular resource.
- The *deferred binding* may go beyond the client's ability to select an implementation at run-time. The implementing module may be *dynamically discovered*, and *dynamically loaded*. It need not be known to or loaded into the operating system until a client requests it.

3.3 Dynamic Equilibrium

It is relatively easy to design and tune a system for a particular load. But most systems will be used in a wide range of different conditions, and their loads (both intensity and mix of activities) change over time. This means that there is probably not a single, one-size-fits all configuration for a complex system. This is widely recognized, and most complex systems have numerous tunable parameters that can be used to optimize behavior for a given load. Unfortunately, a good set of tunable parameters is not enough to ensure that systems are well tuned:

- Tuning parameters tend to be highly tied to a particular implementation, and proper configuration often requires deep understanding of complex processes.

- Loads in large systems are subject to continuous change, and parameters that were right five seconds ago may be wrong five seconds from now. It is neither practical nor economical to expect human beings to continuously adjust system configuration in response to changing behavior.

- It is possible to build automated management agents that continuously monitor system behavior, and promptly adjust configuration accordingly. But such automatons can misinterpret symptoms or drive the system into uncontrolled oscillation.

Nature is full of very complex systems, very few of which are perfectly tuned. Stability of a complex natural system is often the result of a *dynamic equilibrium*, where the state of the system is the net result of multiple opposing forces, and any external event that perturbs the equilibrium automatically triggers a reaction in the opposite direction:

- The amount of water in a sealed pot is the net result of evaporation and condensation. When the temperature is raised, more evaporation takes place, which leads to an increase in the steam vapor pressure, which leads to an increased rate of re-condensation.
- The deer population may be the net result of food supply and predation by wolves. If the food supply increases, an increase in the deer population will lead to an increase in the wolf population, which will reduce the deer population.

If our resource allocations can be driven by opposing forces, we may be able to design systems that continuously and automatically adapt to a wide range of conditions. In this course we will examine multiple uses of dynamic equilibrium to achieve stable operation in the face of ever-changing loads.

3.4 The Criticality of Data Structures

Given the amount of code we write, it is natural for programmers to focus much attention on algorithms and their implementation. But the solution to many hard software design problems is found, not so much in algorithms, as in data organization. Most of our program state is stored in data structures, and it is often the case that most of our instruction cycles are spent searching, copying and updating data structures. Consequently, our data structure designs (and their correctness/coherency requirements) determine:

- which operations are fast and which are slow (e.g. because of the number of pointers to be followed)
- which operations are simple and which are complex (e.g. because of how many different data structures have to be updated)
- the locking requirements for each operation (and hence the achievable parallelism)
- the speed (and success probability) of error recovery (because of the number of possible errors and their detectability/repairability)

Often, when confronted with a difficult performance, robustness, or correctness problem, the solution is found in the right data structures. And given those data structures (and their correctness/coherency assertions) the algorithms often become obvious. In this course we will examine the data structures that underlie much complex functionality, and see how the right data structures have contributed to system performance and robustness.

4. Life Principles

Life is made up of huge projects involving a myriad of complex, parallel activities among numerous independent agents (e.g. growing up, getting and education, getting married, finding and keeping a job, raising children, ...). Thus it should not be surprising if many of the lessons we learn from operating systems turn out to be applicable to many other aspects of our lives (and vice versa).

4.1 There Ain't No Such Thing As A Free Lunch

There are no Magic Ponies or perpetual motion machines. Everything comes at a cost. There are common situations with simple and obvious solutions, but those solutions don't correctly handle all situations. We often have multiple conflicting goals, and an approach that optimizes one goal usually compromises another. There is almost always a downside.

Any solution we formulate is likely to involve costs, trade-offs and compromises. Before we can safely change an important system (e.g. an operating system), we must be able to predict (though research, analysis, or prototyping) the likely consequences. The we estimate their impacts, prioritize our goals, and try to optimize our net expected utility.

4.2 The Devil is Usually in the Details

Every once in a while we find a great solution: some beautiful set of paradigms and principles that eliminate nasty problems and greatly simplify the system. But our first articulation of that vision is seldom right, and it must be refined as we try to understand how we will apply it in all of the required situations:

- Enumerating (or even identifying) all of the interesting cases is a great deal of work.
- In many cases, it may not be obvious how to make a particular situation fit in to our beautiful high level structure.
- Sometimes we encounter one nasty case that simply refuses to fit into the new model.
- Sometimes we can extend our model to embrace the troublesome cases.
- Sometimes we can sub-case, preclude or exclude the troublesome cases.
- Sometimes we conclude that, while beautiful, our idea doesn't work.

Having an inspiration is fun, exciting, and rewarding. But it isn't real until you have worked out all of the details. (Ask any String Theorist.)

4.3 Keep It Simple, Stupid and If it ain't broke, don't fix it

Einstein was once quoted as saying:

> *Everything should be made as simple as possible, but no simpler.*

As should be clear from this paper, complexity is the enemy. The problems we face are complex enough, but sometimes we voluntarily add gratuitous complexity to our solutions. It is natural to look at a solution and recognize enhancements that would make it smarter, or more complete, or more elegant. These insights are fun, but they often prove to be counter-productive:

- Many optimizations require extra work to better handle special cases, and that extra work may greatly reduce the benefits of the optimization.
- Many ideas that seem simple at first become more complex as we examine all of the cases that have to be correctly handled.
- More complex solutions are likely to have more complex behavior, including unanticipated consequences. If we cannot predict the behavior associated with changes, we are likely to find that we have created new problems.
- More complex solutions are more difficult to understand, and hence more likely to have undetected errors and more difficult to maintain.

There are surely problems that call for very smart solutions, but it is often the case that a simpler solution works better than a more clever solution. It is best to avoid more complex solutions unless we have hard data that simpler solutions are not viable and that the smarter solution would fix the problem.

4.4 Be Clear about your Goals

We occasionally build software for fun, but usually we do so with a purpose in mind. It has often been said that if we do not clearly understand our goals, it will only be by accident that we achieve them.

As we pursue a project we will be confronted by many decisions. Some questions are of the form "will this work", and can usually be answered by research, analysis or prototyping. But we will also encounter "which

is better" questions. Some of these will have objective (e.g. measurable performance) answers, but some will come down to priorities and perspectives. A clear understanding (and prioritization) of our goals can help us with these questions, by allowing us to ask how each of our alternatives would impact each of our goals.

As we pursue a project we will see many problems and opportunities. We are in a creative problem solving mode, and it is natural to automatically leap to approaches to address problems and exploit opportunities. But not every opportunity we encounter is on the path to our ultimate goals, and not every problem we encounter is actually blocking the path to our ultimate goals. If we expand the scope of our effort to solving non-critical problems or achieving non-goals, we may find that we have over-constrained our requirements ... making it much more difficult to achieve our goals.

It is also important to keep in mind the high level goals, from which our detailed goals may have been derived. Our detailed goals might be to speed up some operation and eliminate a confusing parameter from some interface. But the real goal was to make the system faster and easier to use. If we myopically focus only on our micro-goals, without periodically assessing ourselves against the higher level goals, we may find that we have won all the battles, but lost the war.

Having a clear understanding of our goals will enable us to resolve differences of opinion, make better decisions, and avoid distractions.

4.5 Responsibility and Sustainability

We are going to live for a relatively long time, and when we are gone, our children will live in the world we left them. We need to understand the long-view consequences of our actions, how they will eventually effect us and others, and then act responsibly for the best long-run outcome.

The same principles apply to operating systems. No software-managed resource can ever be lost. All memory and secondary storage will be recycled over and over again. Errors cannot be dismissed as unlikely events, and no error can be left un-handled. Every reasonably anticipatable problem must be correctly detected and handled, so that the system can continue executing. It is commonly observed that the main differences between professional and amateur software are completeness and error handling. We are all responsible for anticipating and dealing with all the consequences of our actions. OS designers are not, by nature, optimists.

5. Conclusions

Despite the fact that Operating Systems encompass ever more responsibility, relatively few people will actually build operating systems software. But most of us will work on complex software, and we will encounter problems that have already been encountered, understood, and solved by computer scientists, architects, and developers in the field of Operating Systems.

As you read about various mechanisms and issues within operating systems, study them with these principles in mind. As we analyze implications and contrast alternatives, try to pull these principles into the discussion and see if they shed light on the problems. Understand these principles, learn to recognize these problems when you encounter them, and how to apply these solutions to new problems.

> *Powerful tools these are.*
> *Serve you they will.*