Design Analysis and Potentials of MOSES

The MOSES Project:

Meta Operating System And Entity Shell

Daniel J. Pezely

20 May 1991

---

1 Introduction

This paper is intended to critique the MOSES System design. We describe and detail the implementation design elsewhere, but here, we state the overall criteria for The MOSES Project and continue with an analysis to see how well the design satisfies The Commandments of Design. Project conclusions and recommendations follow. This is not a performance analysis or an end-all system recommendation.

2 The Commandments of Design

Here are the design criteria, affectionately called the Commandments of the MOSES Project. And yes, there are ten of them.

I. *Everything will change.*

This is the golden rule: The foundation system must be modifiable at run-time by the user. Therefore, all foundation's internal system variables,

temporary variables, and other states, must all be accessible to the user to simplify upgrades. Upgrades should not require the standard halt-upgrade-restart cycle which looses memory references and states. By the use of genuine dynamic linkers to replace built-in routines, upgrades may be performed at run-time without having to restart the system, thus maintaining memory references, etc.

II. *Allow anything, but more importantly, do not dis-allow anything.*

We must allow for that which we cannot predict; therefore, we cannot force anything upon the user, and cannot restrict anything unnecessarily either. A careful balance must be made between allowing for anything and consumption of resources.

III. *Provide uniform access to all resources.*

Storage, function, and communication are the three basic resources available to all entities using the system. All resources, actual or virtual, which the native operating system provides should be accessible through a system-independent interface. Typical operating systems provide only a hardware-independent interface which varies from platform to platform. Also, uniform access means that low-level resources should be accessible via high-level routines, and high-level routines should permit control by low-level commands—all by-passing any intermediate layers in the communication stacks which typically exist.

IV. *When in doubt, put the feature in a user-accessible location.*

This follows from the first two guidelines but is worth stating even if obvious: There is no reason to hide features from the user, and security is not a reason at this level. Upgrading should be made as transparent as possible, and the distinction between different system versions should be blurred and definitely not rigid. Similarly to different versions, new and improved command libraries, for example, should be replaceable modules which the user has total control over.

V. *We are in this for the long-haul.*

Virtual operating environments for the masses is a nice idea but an unrealistic design goal at this time. More will be said when we evaluate performance; however, we expect hardware technology will lag behind such implementation designs. We are implementing for researchers more than applications developers and users.

VI. *The design must last.*

This is a foundation, not a single application. Any feature which might

cause future compatibility problems or future implementation design compromises should be moved out of the system kernel and into the user (application) layer or be implemented as an external system service.

VII. *Find the re-occurring design elements.*

Recursion in design reduces complexity and ideally approaches a more abstract design. Overall, abstract designs require smaller amounts of code to implement. The less code there is, the less there is to break and upgrade. The less code there is to break or upgrade, the longer the foundation should last.

VIII. *Learn from experience, both our own and those of others.*

Research has already been done in many, many areas which our design can benefit from. Make use of that research and experience. For our own research and to gain experience, any code developed in the design stage should be considered disposable code. Of course, never discard code, but do not be afraid to put it aside to try a new path. Much of this design seems to be combining existing technologies in a new way. That is true, so take advantage of that and see what the results are.

IX. *Build upon existing platforms.*

We are not setting out to develop the ultimate system to end all systems, but rather, we are out to develop a platform which researchers and developers can be both functional and comfortable with. If our system is designed to work with other platforms, users will be able to exploit that and use platforms which they are comfortable with and might have an immediate application for.

X. *The source code* is *the implementation document.*

The technical implementation manual is the system source code. Books and papers are being written, but slightly more than fifty-percent of the source code files are comments explaining the inner workings. Documents, such as this one, are to educate client programmers with the background which the source code comments assume.

3 The Design

To make an operating environment platform suitable for wide-area, multiple users, distributed operating systems come to mind. Since this project is not concerned with developing a complete operating system, we build upon ex-

isting hardware and software technologies. The most common technologies available to us and to the research community have distributed systems features but are not necessarily distributed operating systems in the definitive sense.

Building a research system upon existing systems today means building upon the UNIX[1] operating system. Such operating system services as file systems and local memory management do not need to be redesigned. However, there is one additional service we wish to provide to these existing systems: *The DataSpace*.

We describe other design features which are common or obvious to systems people, in other papers.

3.1 The DataSpace

Fundamentally, the DataSpace gives users the illusion of exclusive ownership of the unity of all memory and devices while actually being a shared memory, distributed over a potentially wide area. This illusion and actual implementation is possible using the notion of an *entity*.

3.2 The Entity

Everything is an entity. Any information representable symbolically forms an entity. This information may be functions or data. Functions provide the system task-capabilities and interface all systems resources. Data may be anything, but the concept of different kinds of data gets replaced by the concept of more or different data, and users impose classifications, not the system. The DataSpace stores all Entities which are both function and data.

4 Analysis of Design

To reiterate what we just presented, we must build a wide-area operating environment platform based upon existing platforms, and we simply add the DataSpace to those existing platforms providing a residence for Entities.

[1]UNIX is a Registered Trademark of AT&T Bell Labs.

4.1 The Design versus the Commandments of Design

Before we show possible applications, let us compare the design with *The Commandments of Design*. We present additional design details below as arguments needed supporting the Commandments, but refer to [1] for all features.

4.1.1 Allowing for Change

By giving the user total system control and permitting run-time modification to any program or resource, change is allowed. We implement total control by, in operating systems terms, putting everything in the user memory and/or allowing users access to the privileged memory.

Of course, two things come to mind: security and resource consumption. We avoid petty security issues at this level; restricting resource access can be done externally. The once extremely scarce and expensive resources are now abundant and inexpensive; for that reason, we permit what appears inefficient resource use. For example, memory resources may be grossly consumed. This is very much a user-centric design: user-beneficial design is the Lab's goal.

Giving the user total control should make possible run-time command library changes. The changes are possible by installing a second command library, making the new one the default. Such installation allows upgrading while the system is running, thus avoiding the usual halt-upgrade-restart cycle, thus avoiding needless downtime. This upgrading technique also allows installing user applications as system command libraries, thereby extending the whole system's functionality.

Genuine dynamic linking [2] permits change and run-time upgrading. Entities and the DataSpace make dynamic linking transparent to the user. The entity notion provides for change. Since the DataSpace stores entities which are functions and/or data, just as programs interchange data, dynamic linking of DataSpace segments allows interchanging functions also. Run-time upgrading simply changes specific DataSpace entities (associated functions and data). This upgrading, of course, assumes the command library is located within the DataSpace.

However—yet a very subtle point—the system *physics* must have some fixed elements. That is, some commands, some features, some structure, or some side-effects must reside within the system allowing for total control,

flexibility, and genuine dynamic linking. Such fixity exists in our system too. But—genuine dynamic linking permits adding new routines which may take-over even the structural commands' functionality, abandoning the old structure and commands. In operating systems terms, this would be like terminating the process manager but first starting a new process manager which orders the termination of the first. So this taking-over of one command library by another minimizes the system's fixed elements to being irrelevant.

Satisfying this Commandment of Design in this fashion directly satisfies most other Commandments, and this Commandment is our golden rule.

4.1.2 Allowing Anything, but Not Dis-allowing Anything

By keeping all system routines user-accessible, anything is allowed and nothing is dis-allowed. We allow anything because the only limiting factors are the developer's creativity. We consider such accessibility similar to UNIX utility pipelining but with extended resources. Remote resources may be used when local ones become limited, since this is a distributed system. We distribute memory, an often limited resource, through the use of the DataSpace, so memory-hogging applications may share remote memories.

We dis-allow nothing the same way we allow anything: the only limiting factor is the developer's imagination. As explained by *allowing for change*, even the residual structure may be over-ridden; thus, we are not dis-allowing for future dynamic linker modifications. This is very useful for saving memory contents (data) while replacing the command library (functions). Of course, such a replacement often requires data conversion, as many upgrades do, but the system may avoid the halt-upgrade-restart cycle and only perform the upgrade.

4.1.3 Providing uniform access to all resources

Uniform access allows manipulation of the communications stack lower-levels with high-level commands which by-pass all intermediate stack layers. Likewise, high-level layers are controllable via low-level, efficient commands which by-pass all intermediate layers. Such manipulation via control messages should apply to *all* resources: memory, functions, and communication.

Uniform resource access is simple but tedious and usually consumes other valuable resources such as memory for storing the access routines. In today's

hardware systems, mass-storage resources are becoming much larger and less costly. Therefore, we actualize this request through brute force implementation efforts.

4.1.4 Keeping Features in User-Accessible Locations

By keeping all features and resources user-accessible, the user can potentially change every system element while the system keeps running. This is another criterion satisfied by the solution to *allowing for change*.

When a user requires new features in a system, users must either wait for a completely new system version or modify source code if possible. New and improved command libraries are simply dynamically linked into an existing system since all features are user-accessible. Incidentally, this blurs the distinction between differing system versions since an old system could be running the latest command library.

4.1.5 Planning for the Long-Term

Our system is not optimized for today's hardware. Our system features consume resources still considered precious and expensive, such as computation. To see what the long-term will deliver, we are developing this system for researchers, not application developers or users.

We are creating an operating environment, not currently a native operating system or a specific virtual environment application; rather, this is an open-ended system. Being open-ended, we cannot possibly imagine all of the uses of our system. Keeping some potential uses in mind, we should not restrict ourselves by optimizing for today's technology.

4.1.6 Making the Design Last

Users want stability which means less incompatibilities between system versions, yet users desire newer, better features as existing features seed new ideas. Users suggest design changes, and to satisfy the users, the designs change. By allowing the users to independently change the system internals at will, boundaries between different system versions become extremely blurred and for all practical purposes, become irrelevant. Yet again, by satisfying *allowing for change*, we satisfied this criterion. We implement *allowing for change*, again, through the use of genuine dynamic linking—dynamic

linking of every routine in the system, both internal and user-command routines.

4.1.7 Finding the Re-occurring Elements

The re-occurring element which effect users directly, deals with upgrading both command libraries and applications. By making the command library effectively identical to applications, we reduce the system complexity while providing much more user flexibility.

The notion of an entity provides the flexibility which the user needs. Everything in the system is an Entity, both functions and data. We treat all entities equally by not distinguishing between functions and data at the lowest software level. Also, entities do not distinguish between different types of data: some just have more of different information.

4.1.8 Benefiting from Experience

Much research needs to be done with the design and implementation of our system platform, but more research needs to be done using the platform. Therefore, we should produce and evaluate a platform before heading in new design directions. Such a production effort usually locks users into one implementation; however, our design research provides for a suitable design which avoids locking users into any one implementation version.

4.1.9 Building Upon Existing Platforms

As is heard so often but still ignored by many implementors: we should not re-invent the wheel. Also, since there are many applications for systems such as UNIX, we make those systems and applications user-accessible by making our system run as a UNIX application. Such accessibility also extends the native system, thus making our system more than just an operating environment but another tool for the base system. This extension provides even greater flexibility for uses.

4.1.10 Making the Source Code an Implementation Document

We incorporate text documents describing the implementation into the source code. This incorporation permits and encourages implementors of low-level

tools to see explanations of features along with code blocks. Seeing these features and code encourages duplication and modification. We shall donate our system source code to the public domain further encouraging development and improvement.

4.2 Possible Applications

Some early applications benefiting users are:

1. Tele-Conferencing – A channel permitting multiple users over a potentially wide area to communicate with and share information with each other.

2. Interactive Simulation – Any simulation which could benefit from distributed systems, shared memories, and transparent communication between systems could make use of our operating environment simply by using communications routines we develop.

Note that these possible application descriptions do not mention display graphics. Output of any form (audio, graphic, etc) may be as extensive as the programmer desires. The choices of colors, 2 dimensions, 3 dimensions, etc, are completely up to the programmers and tool-builders.

5 Conclusions

After analyzing other operating systems and distributed systems designs, we feel our meta operating system design for an operating environment will produce a sold platform.

To make the meta operating system complete, many low-level and high-level tools need to be built. Most early operating environment users will be client programmers writing such tools.

6 Recommendations

We described and analyzed the theory of our operating platform in this suite of papers.

Our next step should be to fully implement the core of the meta operating system:

- The DataSpace command library

  management routines

  user routines

- The Communications Entity command library

  primitives

  user routines / toolboxes

Additionally, Communications Entity protocols should be developed and standardized. Programming interfaces to the primitives should also be standardized by the kernel implementors and client programmers together.

Tool design is another topic for an implementation and analysis suite of papers.

References

[1] Pezely, D.J., *MOSES Kernel Implementation Design*, Human Interface Technology Laboratory, Washington Technology Center, University of Washington, Seattle, WA, 1991.

[2] Ho, W.W., Olsson, R.A. "An Approach to Genuine Dynamic Linking," *SOFTWARE–Practice and Experience*, Volume 21, Number 4, April 1991, pp. 375-390.