Operating System: A Relic of the Past

Operating systems: A relic of the past

© 1995 Bob Frankston

Introduction 2017

I came across this paper which I presented in 1996 at a workshop on operating systems (SIGOPS?)

Today I'm more critical of Multics and am thinking more about distributed systems but that's another paper …

The original date was March 20, 1996.

Overview

The idea of operating systems have been around since the mid 60's. It is time to reexamine the basic rationale for such systems as we prepare the next generation of systems and as computers become the basic components of our infrastructure.

The direction of computer science/software engineering was set in the 1960's when the primary concerns were making effective use of expensive computers and managing what were then large efforts. Though the world has changed greatly since then, we are still pursuing the same basic directions. A general purpose computer comes as a set of hardware matched to an operating system that manages resources. Software Engineering methodologies are focused on assuring that one can specify and follow through on large projects.

But the computers are getting smaller and cheaper. PC's are only a middle stage in this evolution. Individual systems are becoming simpler but their interactions are becoming more important and more complex.

Origins

The computer operating system was a major accomplishment of the 1960's. The 1960's was the age of discovery in computers. Compilers (automatic programming), databases (network and then relational) were all important.

With the perspective of time we can rethink our assumptions and the results. The operating system has persisted as a central theme because it seems that there should be a conductor for every orchestra. But we've mistaken a powerful, though pragmatic, solution for a fundamental principle.

This is not merely an historical exercise to see why one particular set of ideas won. With computers we have an unusual degree of freedom to not only rethink the past but rework the present.

Perhaps its my own bias, but I still view Multics as the high point of operating system design. Of course, this is an idealized Multics, one that doesn't have many of the hacks and kludges of the real implementation. The mystery was why it's principles are still central.

The reason, aside from Honeywell's foibles, is that sufficiency has been more important than perfection. In fact, there is no perfection, just tradeoffs. The operating system itself is one such tradeoff.

Operating Systems

Not all systems have operating systems. Embedded systems are often written to standard libraries or direct to the "iron".

The operating system evolved from such libraries into a resource manager at the heart of a complex system. In the days of the IBM 7094, the Fortran Monitor System (FMS) was little more than a set of standard subroutines and I/O routines that operated according to a set of conventions. They supported one job at a time as part of a stream of jobs conforming to local conventions. Typically unit #5 was the input tape and #6 was the output tape. Eventually, these became magic numbers no longer associated with tapes.

By the time of Multics, we saw the operating system as a manager of system resources. The computers were expensive and it was very important that we provided for sharing. More to the point, fair sharing, of the resources among the competing interests. The operating system also contained the file system. Multics also introduced the notion of the operating system as the center of system security.

The ring structure of Multics epitomized this model. The kernel of the operating system, was ring 0. The concentric rings were intended to introduce degrees of integrity. Ring 1 evolved to contain all portions of the system that did not require very high performance direct access to the hardware. All user programs were run in ring 4. And then things stalled.

The fundamental model assumed a large system managed by a trustworthy systems manager with software provided by the systems supplier. The ring structure was to support the system database services and eventually third party systems. But problems started to appear.

One was the concept of a security kernel which added incentive to the idea of simplify the operating system to its basics so that it could be understood. This was also considered important for reliability. The file was, once central, was reduced to some basic disk management functions with the rest of the system being moved to outer levels. Interestingly, OS/360 didn't even have a file system initially, just some optional cataloging. Multics at the high end and RSX-11M were systems that separated naming of files from managing their on-disk structure. Files had a unique id or a disk block id.

The notion of rings ran into trouble when the simplifying assumption of a central authority gave way to the need to support mutually suspicious subsystems. A third party database manager could not be trusted unfettered access to all of the user's environment.

Security has stayed important but a poor step-child of operating systems since it was simply not important within small groups. Fundamentally complex, heavyweight, secure operating systems were artifacts of expensive mainframes domiciled in computer rooms.

Minicomputers were not the first small systems. We had process control computers and other specialty systems. But RSX, Unix and other operating systems represented the generation of minis that succeeded the mainframes as shared computer systems. Though much much cheaper than mainframes, it was still important to share the systems. The operating systems for these machines were similar to the ones on the mainframes. A big difference was that there was a priority on pragmatism than perfection.

Unix is central to this generation though it didn't reach full maturation until the 1980's and is continuing to evolve to the 90's.

Enter the PC

PC's evolved from chipsets with little software. While the CP/M machines were imitative of the earlier minicomputers, the game computers like the Apple][lacked such amenities. They either ran basic on the iron or had small monitors to assist in writing simple applications. For those of us who wanted to deliver capability, this was fortuitous. Though well versed in operating systems, we also had experience with earlier, smaller systems and specialty hardware. If we could write operating systems, we could write applications that did the same things themselves.

Much more important was the realization that the user's didn't care what was going on inside the system, what mattered was whether they could make the machine do what they needed it to do. If the operating system were convenient then we'd use it, if not, then we would ignore it and, if necessary, work around it.

Since the Apple][and its ilk were sufficiently popular we could afford to write to the standard iron. We wrote directly to the screen. We wrote directly to the disk controller. Actually, calling it a disk controller was giving it credit, it we had to do nearly all the encoding and processing ourselves.

As the PC evolved, we got more and more services provided. But those of us who kept to the notion that the user experience was paramount would pick and choose which of these services we would use and which we would work around.

The Abort, Retry, Ignore message separated the pros from the amateurs. The pros took responsibility for handling all eventualities and the amateurs just used the standard C-language I/O packages which placed the burden of handling contingencies on the users.

The Macintosh graphics system was not an operating system. Rather, it was a graphics toolkit coupled with libraries to simplify conforming to the systems conventions. The early Macs didn't even have a file system – just utilities for dealing with the disk. The Mac was a distillation of experience with both the Apple][and the Xerox Alto..

Microsoft Windows was also originally conceived as a graphics library with some capability for running applications as a successor to the DOS command processor or shell. But it struggled for acceptance because the applications were king and the overhead was simply too much for the needs of the application. It wasn't until the 1990's when memory prices were low enough and the systems were fast enough that it found acceptance. But Windows 2 was little more than the Excel system. One would only run Windows for a limited number of applications.

It was only with Windows 3.1 that the system ran DOS applications sufficiently well and provided a modicum of concurrency that it caught it on. It was only then that the power of a common interface and integrating environment was able to assert itself. Only when it enhanced the applications did people use Windows.

Windows/NT is, in many ways, a great operating system, but it is struggling against Win95 because the latter makes the pragmatic tradeoffs in favor of supporting applications. 95's tradeoff of usability against integrity is a powerful advantage.

But both are facing a battle for survival as we continue to evolve our systems. The PC has grown far larger and more complex than any mainframe from the 70's and has lost its raison d'être.

We already see this happening within the PC as interactions between components dominates the basic systems services. We cling to the notion that the operating system is at the center as all the activity goes on around it and between systems. Yet we are still building systems as if these are just minor extensions to the current structure.

We have lost sight of the fundamental idea that the operating system is merely a set of conventions that we abstract from common practices and there is nothing fundamental. This is acceptable as long as there is enough slack in the system to allow for it and as long as the complexity doesn't overwhelm the architecture.

Exit the PC

The PC is facing two fundamental threats and a myriad of smaller ones. The two top issues are complexity and overhead. These are closely related and are replays of the demise of the minicomputer. It is also exciting whenever we get a chance to rethink and restart.

This is occurring as computing itself becomes the fundamental building block of our infrastructure. Not computers, but computing or intelligent systems. This involves creating many interacting systems. But we have no notion of how to make these complex interactions scale.

The solution is then to do what we know how to do for now and revel in the present. The key elements include:

- ***Iron.*** Really silicon, but the metaphor of going directly to the underlying layer is important. We can build chips for specific applications at the cost of $1 when the memory alone to support an operating system can be 100 times as much.
- ***Digital Connectivity.*** We are no longer just dealing with systems in isolation. More important these systems do not have any central administration nor common benevolence.
- ***Imperfection.*** This is a vague notion but important. We can't assume that we are building upon well functioning layers. We can't even assume layers. Instead each element must take responsibility at each level to assure it is delivering what it promises. Conversely, we have learned it is important to allow the consumer to choose a lower quality of service than assuring perfection in every element since doing so is expensive and ultimately futile.

The most important lesson to learn is that we can and should be able to discard the comfort and overhead of the operating system and reinvent the services we need. We need to reexamine some of the basic gospel of computer science:

- **Reusability.** It is better to build out of existing pieces than to create new ones every time. But this notion easily goes awry when the effort involved in assuring reuse overwhelms the cost of rebuilding. Consumer electronics provides great lessons in the advantage of just replacing entire systems than reusing pieces.
- **Layering.** Breaking problems down into simpler elements is a powerful technique for building systems. After sufficient experience we develop a set of conventions that allow arms length cooperation.. Out of this arose the notion of the operating system API. What is lost is the notion that these

API's must be renegotiated as the circumstances change. A network is not simply a remote disk drive.

Networks represent very different semantics from a local disk reference. Not only are there additional failure modes that should be handled, notions of performance and delay don't even have analogs when dealing with a local disk drive. When we take this into the wireless domain, the lie is put to the test and fails miserably. Yet we continue to model network as simple extensions of the local system. This only touches upon the complexities introduced by networking among mutually suspicious systems.

- ⇨ **Uniformism.** The purpose of layers is to try to provide a uniform basis upon which to build our applications. But it is often the idiosyncrasies of each system that make the system what it is.

We also need to question the scope of specific techniques and paradigms. These do represent good practices but can readily become dogmas.

- ⇨ **Objects.** Objects are a good technique for encapsulating methods and instances as unit. They are a nice way of structuring systems. Like other forms of layering they can serve as an internal structuring tool and, to a limited extent, as a way to codifying arms length relationships. Objects go beyond layering in that they are more independent of underlying systems and have relationships among themselves and isolation between themselves. This leads to complex interactions.

With objects, we take the dangers of layering – the lies and misrepresentations – and multiply them as objects build upon other objects as both layers and peers. Without oversight, these relationships drift apart as in a whispering chain. Adding the notion of distribution, implicit networking, creates a volatile mix that is likely to explode or simply fail.

- ⇨ **Minimal Kernels.** These are still operating systems but the foist the blame onto the applications by declaring all the service subsystems to be outside the kernel. This isn't necessarily bad but neither is it necessary good. It attempts to preserve the notion of a standard environment into which one can load applications. An alternative is to statically link systems together rather than relying on the dynamic environment of the kernel. Key to this is the merging of the embedded system and the general purpose system. The embedded systems come to the fore as hardware becomes a trivial part of the cost of systems. This is not to say that the notions of operating systems are obsolete. But it is as knowledge rather than code that they survive.
- ⇨ **CPU Centricity.** Just like a car might be characterized by its horsepower, a computer is characterized by its CPU. We need to identify systems of cooperating components as the entity that is important. This goes beyond the notion of the network as the system since we are not positing the form of cooperation and can reduce the network complexity and scope in these systems.
- ⇨ **Paging.** Generally paging is used to give the user the illusion of having more physical memory than there really is. Hence the term virtual memory. As long as there was sufficient memory to keep a working set in real memory, the illusion could be maintained. The problem is that as systems become more complex more and more components are required to maintain the user interface (as well as support other functions). If these aren't used constantly they will get displaced by more active components. But when one shifts tasks, the system goes into a frenzy of paging in order to bring in the main components, each of which has responsibility of a small portion of the user interface that must be repaired at each change.
- ⇨ **Secure systems.** Users have physical possession of much of their devices and much of the infrastructure. The idea of a security kernel is meaningless. We can have some security in parts of the infrastructure. Encryption allows some degree of security for information in insecure systems. As the infostructure becomes more important integrity and security of data become more important. But we can't rely on naïve notions of secure kernels and trusted packages to deliver on these needs. The mechanisms must be robust and assume both technical and human error as the norm.
- ⇨ **Software design methodology.** Obviously the idea of doing a design is not bad. What is bad is the assumption that one can design a system as a whole and then implement it. At the same time as we take more responsibility for an entire system

including the silicon, we have less control of the environment in which it runs. We are incrementally adding capability rather than building entire systems. Of course, we really don't have any idea how to do this.

- ⇨ **Messaging and signaling.** Sending messages between systems is a standard way of building cooperating systems. An alternative is a shared environment. This sharing is really one of common representations and ability to reconcile differences. These is the basis for more robust cooperation since it provides a mechanism for repairing damage and limiting drift. It must occur between cooperating systems at whatever level they operate. It is not necessarily a uniform capability though common techniques can be used.
- ⇨ **Trust and reliability.** These are at the heart of how computers have differed from other appliances. We have had the luxury of factors of a trillion in scale. We are now at the limit of the complexity of interactions compounded by the distributed authority that frustrates the ability to assure proper behavior even if such behavior was well-defined.
- ⇨ **Common purpose.** We don't really have the luxury of designing a system as a whole. Not only are there the software design issues and trust issues, we are writing application in a real world of disparate parties, with little in commone. The ability to add function quickly and incrementally will win over a full blown design that doesn't add sufficient additional value and will win over a design that requires too much cooperation between competing parties.
- ⇨ **Error messages.** It does little good beyond frustrating the user to report that something has gone wrong. Add the words "fatal" is just an attempt to heighten the user's anxiety and bring on a heart-attack. Once the user is dead, the actual system failure becomes less important. Rather than reporting errors, we must report solutions – what should or can be done to resolve the problem. This is nontrivial because the explanation must be tailored to each user and each situation. A failure of a network server means that the system administrator

What to do?

Simplify.

I've always been a skeptic of the *less is more* school. After all, we had these powerful engines that could do just about everything. I've always enjoyed pushing operating systems and tools to their limits. But we must discard these as training wheels and face the hard problems of building the new infrastructure. We must be well versed in the old techniques and learn the new ones.

The irony is that the world is going more and more to cooperating systems of intelligent devices. But it is doing so only as fast as we can deliver. And we need to face up to what we don't know how to do.

These comments about what we can do are only the barest starting points. The main point in this particular essay is that operating systems are not the magic answer. It is important to pursue these ideas in much more detail but that's not the purpose of this overview.

- ***Build simple intelligence devices.*** The notion of simple given that we can place the equivalent of an early PC on a single chip has grown. But just like PC's freed us up from the need to assure full utilization of all the hardware, these new devices needn't do more than one or, at best, a handful of functions. A watch with the power of a PC is given to three year olds as a reward for buying a cheeseburger and no one mourns if it's lost by bedtime. Instead of reusability build for function. If one is building a golf watch, make it have a golf scoring button and color it green and use a different watch off the course.
- ***Build in simple cooperation***. We can design some simple protocols for cooperation and evolve them over time. This is the secret of the Internet. Perhaps the golf watch can transmit scores to the PC but it doesn't need to be your digital communicator since you've got a phone in your pocket anyway and don't want the weight on your wrist.
- ***Assume failure.*** It is normal for systems to fail. You should expect that services you call upon are unreliable. You should depend on what runs locally and be able to survive failures as minor annoyances. Do not do any nontrivial error recovery since the interactions between failures are major sources of untestable failures of their own. Better

to isolate the failure than to do complex recovery. At the same time, you shouldn't hide failures since then the system still fails but behaves perversely. The World Wide Web provides one example, other servers are likely to fail but you can still use the rest of the infrastructure and can retry if necessary. This is more problematic as we build layers of middleware upon the Web.

- ***Learn by doing.*** If we are to build large systems out of simpler systems, the individual components should be effective individual. It follows that sets of them should also be useful. An incremental implementation not only assures early utility, it allows for learning as one implements. But this must be done with an expectation of reworking and revisiting earlier decisions. Loose coupling between components helps maintain the system through change as welling as making it robust against minor failures and mismatches. Despite these, we are still subject the to complexities of the growing interactions between systems.
- ***Learn how to composite systems.*** We don't have an understanding of how to manage the interactions as we composite individual systems. How do the policies interact? We cannot know the full consequences of these interactions so how do we prepare for the contingencies? I think of this as a policy interaction. One general approach is to have an overseer for each interaction but how does on do this in practice. This is a vital area in research, one long past due.

Summary

The operating system has come to characterize the substrate upon which the applications and services are to be built? This has been a powerful idea that has been the underpinnings for much that we have done.

But each time we reinvent computing we need to reexamine these notions. And with each generation the operation system becomes less the core issue. Minis were dependent upon operating systems for their generality. PC's didn't get full operating systems until very late in their evolution.

The next generation of computing is moving rapidly beyond the purview of the operating system. While the fundamental concepts remain we need to discard our training wheels and fly. If metaphors must be mixed, so be it.

Followup

Please send comments and suggestions to me at *Bobf@frankston.com*. (in the Web version this will be a mailto: reference).