An Overview of the VSTa Microkernel

Andrew Valencia

Valencia Consulting

ABSTRACT

VSTa is an experimental kernel which attempts to blend the design of a microkernel with the system organization of Plan 9. The result is a small privileged kernel running user-mode tasks to provide system services such as device drivers, filesystems, and name registry. Like Plan 9, each service provides a filesystem-like interface.

1. Motivation

Two operating systems have emerged in recent years which show considerable promise. The POSIX-conforming QNX operating system[1] is a relatively "clean" microkernel, with a 4K executive and most traditional system functions running as user-mode processes. Meanwhile, Plan 9[2] has emerged as a next-generation view of computing from the original architects of UNIX. While each system represents significant new system functionality, the frustration has been in the proprietary nature of each system--synthesis and experimentation by an outside party is difficult or impossible.

2. A Platform for Experimentation

2.1. A New Source Base

With these interesting new ideas to try, and without a suitable code base on which to try them, the only option was to wait for the authors of these systems to publish more papers. When the author recently had the opportunity to take a 6 month sabbatical, the result was the VSTa operating system, written from scratch. VSTa is a non-proprietary source base permitting experimentation in many of the areas first broached by Plan 9 and QNX. While the full design is beyond the scope of this overview, several general design decisions are worth mentioning.

2.2. Symmetric Multiprocessing

Microkernels have a surprising interaction with multiprocessing support. In a monolithic kernel, vast amounts of system functionality all reside in the same kernel address space. When the kernel first supports multiprocessing, all of this code must be revamped to handle multiple parallel threads of execution.[3] Much of this "kernel" code is moved out to individual tasks in a microkernel organization. With such an organizational change, the job of converting this code to the appropriate level of parallelism is divided. A process can be single-threaded; it can be marked serially-reentrant (multiple threads, but only one running in the address space at a time--like a uniprocessor UNIX kernel); or the threads can run fully in parallel. The decision is made per-process.

While real-time attributes are discussed later, it is worth noting that the job of writing code to operate correctly in the presence of parallel CPUs has some synergy with the goal of allowing kernel preemption. In both cases the programmer must always consider the possibility of the same code paths being entered and reentered at arbitrary points. Kernel preemption imposes additional demands, especially in code paths

---

[1] Hildebrand, Architectural Overview of QNX

[2] Pike et al, The Plan 9 Operating System

[3] Operating systems like System V.4.2 MP try to mitigate this with serializing code "wrappers." Few people seriously present this as an appropriate solution, or even an acceptable temporary fix.

like exit().

With the smaller kernels achieved by moving so much functionality out to processes, the remaining code is much easier to design, code, and test. Since writing parallel, preemptive code is harder to write than traditional single-threaded code,[4] the reduction in source size is desirable.

VSTa's machine-independent layers were written for a shared memory symmetric multiprocessor. A P/V semaphore interface is used for sleep-oriented interlocks. A P/V spinlock is used for spin-oriented interlocks, and also used to interlock against interrupt service procedures. The machine-dependent code, sadly, is only written for a uniprocessor i386--the only machine available to the author.

2.3. Real Time Facilities

While VSTa is not a real-time operating system in itself, numerous features associated with real-time systems offer themselves naturally to solve microkernel design issues. Process memory locking is necessary in order to allow a disk driver task to run as a user process (as otherwise, of course, you will deadlock when your disk driver tries to demand page in a piece of itself from swap.) Non-degrading priorities are necessary to permit critical system services to respond to many users without being penalized for their apparent heavy CPU use. Low-latency process dispatch is necessary to allow interrupt service code to run in a deterministic amount of time after a device event--especially important in the case of heavy data sources like dumb serial ports and LAN interfaces.

VSTa was designed with memory locking and real-time priorities. Except when a spinlock is held, a thread is preemptable even when running in kernel mode. Most spinlocks do not involve interrupt-driven code; for these, interrupts are still accepted and queued even while the spinlock is held--preemption to a real-time process is delayed until the spinlock is released.

2.4. Scheduling

VSTa uses a very conventional priority-driven scheduler for real-time processes. However, most processes in the system run under an interactive, timesharing scheduling algorithm with unusual properties. The scheduler is driven from a tree where runnable processes are leafs. The internal nodes represent the partioning of groups of processes into percentage "slices" of the CPU pool (much like a fair share scheduler), with the lowest nodes containing the threads within a process.

This organization has two desirable properties. First, it allows users and groups of users to partition the CPU resources fairly among groups based on local policy. With the ratio of CPUs to users approaching 1:1, the classic departmental computing scenario may never arise. But it can be convenient to guarantee that some particular server will never consume more than half the CPU time (unless it would otherwise be idle.)

Such a scheduler also provides many of the properties of a gang scheduler. When the classic UNIX scheduling algorithm is used to run closely cooperating processes, its global nature allows any runnable processes to compete directly with the threads. Since all threads under VSTa exist under a common scheduling node, the threads can voluntarily relinquish the CPU; the CPU time relinquished remains within the "pool" of the node, so only other related threads under the node will complete for it.[5]

2.5. Messaging

In a microkernel, the goal is to identify a small, cohesive set of kernel functions upon which all current and desired future system functions can be implemented as user mode tasks. Like Plan 9 and QNX, VSTa uses a messaging engine as the underlying microkernel mechanism. VSTa, like QNX, supports scatter/gather lists for messages. VSTa, however, had to use significantly different techniques than QNX--VSTa supports virtual memory, where QNX can always assume the presence of physical memory for its messages.[6]

---

[4] The author's estimate is 2-3 times harder.

[5] Of course, if there are no runnable threads, the CPU time will be parceled out elsewhere in the scheduling tree.

[6] The QNX designers have designed, but not yet implemented, a virtual memory capability for their system.

VSTa structures the exchange of messages between client and server. A would-be client requests connection; a connection indication including the would-be client's capabilities is presented to the server. The server accepts or rejects the connection. If accepted, the client can then send messages. Each element in the scatter/gather list of the message is made available in the server's address space when the server receives the message. The contents are mapped on demand as the server makes reference to the data. Alternatively, the server can merely pass the message on without touching its contents (for instance, a middle module in a protocol might add a new initial buffer without needing to examine the contents being encapsulated.) The data was never mapped or read; the passing of data in these cases is thus quite efficient.

If the server returns data it is copied into the client's address space before the client's request is completed. It would be desirable to use the same "lazy" semantics for message mapping as the server, but this would make it difficult for the server to know when the client is done using the data. Techniques involving the "handing off" of pages of data are possible, but many servers would then have to copy data into new pages; any performance benefits can easily be lost.

2.6. Filesystem Interface

Plan 9 imposes a filesystem-like interface onto all objects in the system. QNX permits a server to provide a filesystem interface, but does not pursue such a thing as a goal in its own right. Plan 9 allows each process to build its own filesystem view by attaching objects at arbitrary points in its filesystem namespace. QNX provides a global view, but does not implement these mount points in the classic UNIX (and Plan 9) way. QNX instead looks at its "mount table" as a simple mapping of leading strings for mount points, and a corresponding server.

VSTa borrows much from each operating system. Like Plan 9, all servers provide their services in terms of a filesystem-like interface. The format of the messages sent through the microkernel is standard and implements a filesystem protocol similar to Plan 9's 9P protocol.[7] VSTa use a string table approach like QNX, although the table is private to each process. In fact, the table and its interfaces are all entirely with the C library; neither the VSTa kernel nor its servers have any control over a process' mount table.

2.7. Capabilities

VSTa breaks from current operating systems in the way it structures capabilities. The desire was to allow a hierarchy of abilities, and to allow a given user to create sub-abilities without the intervention of a system manager. The resulting system is powerful, but does not map well onto a POSIX interface.

In VSTa, an ability is represented as a dot-separated sequence of numbers, called an *ID*. The numbers become more specific reading from left to right. An object in VSTa[8] has a label with such a label, and for each position, another bitmask indicating what actions are permitted (It also has a default access, which is just OR'ed in with any other bits granted.) For instance, assuming no default:

1.	2.	1.	3
EXEC	READ	WRITE	DELETE

Would indicate that someone possessing 5.3 could not access the object; someone with 1.5 could only execute it (a mismatch discontinues the accumulation). A hierarchy of "super users" is gained with the last rule: someone with 1.2 would gain read, write, excute, and delete abilities! When someone possesses an ID which matches a label to the length of the ID, the ID is said to *dominate* the label. The remaining bits are OR'ed in as if the match continued to the end of the label. The superuser of a VSTa system, therefore, is someone who has an ID of length 0.

One can *forge* a new ID; it is permitted if at least one of the current IDs dominates the new ID to be forged. Thus, someone who logs in with the ability 5.7 could store all sensitive data with a label of, say, 5.7.1 with all access requiring a full match:

---

[7] The protocol would have *been* 9P, except that it was designed before this level of detail was available for Plan 9.

[8] More precisely, the objects offered by standard VSTa servers.

5.	7.	1
(0)	(0)	READ|WRITE|EXECUTE|DELETE

If this same user then wanted to run a somewhat suspect application, he could forge a new ID of 5.7.2, disable his current IDs, and run the application. Since the application does not possess an ID which allows access to 5.7.1, the user's data is protected. Because such ID manipulation can be done by any user, fine-grained protection designs are possible in a way which UNIX forbids without extensive use of super-user powers.

3. Current State

The VSTa kernel currently comprises 2700 lines of portable C, 750 lines of i386-specific C (using the "count the semicolons" method), and 400 lines of i386 assembly code. It compiles into roughly 50K of 32-bit i386 object code, with another 10K of data. Device servers exist for the keyboard, screen, floppy disk, ST-506 hard disk, and real-time clock. Servers also exist for a flat contiguous-allocation filesystem, a DOS filesystem, an environment server (used by getenv()/setenv()), a swap manager (to manage multiple swap partitions and permit dynamic swap partition additions and deletions), and a service registry database.

The system boots and runs on top of either a flat contiguous filesystem or a DOS filesystem. Further device drivers and filesystems can be started and stopped from the command interpreter. The system does demand-paging of executables and does page-stealing when memory becomes scarce using a two-handed clock algorithm.

At the application level, a C library and include files have been written to conform to the POSIX specification. GNU C, as, and ld have been ported and run natively under VSTa. Emacs and other amenities are also available. As each port identifies a missing area, it is coded up based on the POSIX standard. Thus, the system deviates from POSIX more by omission than otherwise. The exception is the area of protection, where this incompatibility was foreseen and accepted at the conception of the project.

4. Future Projects

It is satisfying to have a freely accessible environment on which to prototype further facilities. Several categories of experimentation are of active interest to the author.

4.1. Performance

VSTa provides very good interactive and compilation performance on a single-user 25 Mhz i386 PC. However, several areas could be streamlined. For small buffers of data the current page-mapping messaging techniques are overkill; it would be much cheaper to just copy the actual data around. A "fast path" for small buffers could shorten code paths for keystrokes, RS-232 bytes, and small writes to the screen. This must be traded off against the additional complication in the messaging machinery--but a prototype is certainly called for.

Some servers, for the sake of simplicity, do not take full advantage of the scatter/gather functions. At least the disk drivers and filesystems should be carefully optimized to make best use of scatter/gather lists.

4.2. Clustering

Accessing services through messages immediately leads one to ponder the possibilities of remote access. The scatter/gather organization and lazy referencing of VSTa should make it possible to move the data from a client to the network interface without any intermediate copying of the data. Incoming connections must be mapped into a local VSTa server connection; the networking daemon must forge appropriate IDs for the client. A mapping database used by the daemon might suffice:

```
theirbox:1.* -> mybox:99.*
trustedbox:* -> mybox:*
*:* -> REJECT
```

This would allow both access control (you must have 1.* on **theirbox** to log in; **trustedbox** has the same accounts as us) and dynamic translation of IDs between systems.

The name server is currently a local entity. However, once remote service access is available, it is a simple matter to import other node's name servers and use them locally. Because VSTa supports Plan 9-style union directories, you could even mount each name server at the same point in your local filesystem name space, with your local name server coming first. Ultimately, a network-aware database system must be implemented, but it is interesting to ponder how far these simple and powerful Plan 9 techniques can take one.

5. Acknowledgements

The knowledge and vision of the Plan 9 and QNX developers is appreciated, both in their papers and on Usenet. This project would not have been possible without the outstanding tools from the Free Software Foundation--especially the GNU C compiler and its accompanying utilities. Finally, I would like to thank the Bill and Lynne Jolitz for the 386BSD source code. Its availability as a reference helped me unravel many an intricacy of the i386.