A Model and Prototype of VMS Using the Mach 3.0 Kernel

Cheryl A. Wiecek
Christopher G. Kaler
Stephen Fiorelli
William C. Davenport, Jr.
Robert C. Chen

Digital Equipment Corporation
110 Spit Brook Road
Nashua, NH 03062-2698
{wiecek,kaler,fiorelli,davenport,rchen} @star.enet.dec.com

Abstract

Digital's VMS operating system has been a successful software base for our VAX processors since the late 1970's. Existing operating systems are facing many new requirements and challenges in the 1990's and beyond. This has led us to investigate new approaches for designing, implementing, and maintaining VMS. One such effort is described in this paper. Using the Mach 3.0 kernel from Carnegie Mellon University, we developed a multi-server model and prototype of VMS that emphasizes platform-independence and internal partitioning. We describe the challenges we faced, the model that we evolved to address these challenges, and the prototype that we built to demonstrate feasibility. We conclude with a discussion of our findings and possible future directions.

1 Introduction

VMS[1] (Virtual Memory System) is an operating system that was designed starting in 1976 as a general-purpose time sharing system for Digital's VAX line of processors. Over time, VMS [7] has evolved into a large collection of functions and features that support a large customer base and many applications. Preserving and making enhancements to VMS is a challenging effort in light of its dependencies on the VAX architecture and its monolithic structure. This has motivated us to investigate ways to restructure VMS to accommodate open and distributed computing requirements in and beyond the 1990's while continuing to support current VMS customers and applications.

Mach [1,9,11] includes base support for architecture-independent virtual memory management, threads, and interprocess communication. Whether or not Mach could support VMS was intriguing to both CMU and Digital. The micro-kernel [5] approach that Mach 3.0 represents also had potential for addressing architectural dependencies and structural issues facing VMS. The possibility of building a VMS environment on top of Mach using a number of servers that might be shared, replicated, and replaced as needed was promising.

After studying Mach literature sent to us by CMU, a VMS engineering advanced development effort was proposed to investigate the potential of micro-kernel technology to support a VMS environment using Mach 3.0. The goals were:

- Learn the requirements that VMS has on a micro-kernel.
- Understand the strengths and weaknesses of a micro-kernel approach for VMS.

[1]VMS, VAX, VAXstation, VAXcluster, and Digital are trademarks of Digital Equipment Corporation.

- Identify features, mechanisms, and policies in VMS that are difficult to port to non-VAX architectures and suggest alternatives.
- Investigate the implications of a micro-kernel approach for VMS-based applications.

Porting VMS to a non-VAX platform or developing a product version of VMS based on the Mach 3.0 kernel was not part of this research effort.

A small team of VMS engineers was assembled by January of 1991 and is working to demonstrate that it is both possible and desirable to build an operating system that:

- uses minimal kernel and client-server design techniques to ease change and evolution,
- promotes portability with an implementation that emphasizes architectural independence,
- has respectable performance,

  and
- preserves the VMS user environment.

An agreement with CMU gave us access to the expertise of, and support from, the CMU Mach team. One of the VMS engineers on the project relocated to Pittsburgh, PA. and worked from the CMU campus.

The project evolved into a two-phase effort: developing a model of VMS using the Mach 3.0 kernel and design philosophy, and implementing prototype software based on the model.

During the first phase of the project, we developed a model of VMS that isolates the VAX architectural dependencies and existing VMS software interdependencies. This phase lasted about eight months. The model we produced is our vision of how VMS could be structured to ease evolution, portability, and maintenance. In this model, we partition VMS into multiple platform-independent servers and use a client-server design approach to provide VMS user processes with access to this VMS environment. Each VMS process consists of two cooperating Mach tasks: a user image space where VMS process images execute, and a process server that manages the VMS process and its images.

During the second phase of the project, we developed prototype software that includes six servers to demonstrate the viability of our model of VMS. This prototype runs on a VAXstation and provides for execution of existing non-privileged VMS binary images. VMS server tasks include a library that provides access to functions in the Mach 3.0 kernel and 4.3 BSD UNIX[1]-compatible single server that we obtained from CMU. We are currently measuring and evaluating this prototype.

In the rest of this paper, we describe our model and the prototype work we have completed. This includes details on the challenges and issues we faced, as well as our model design, prototype implementation, and findings.

2 VMS Model Development

A major undertaking for the team was to come up with a model of VMS that addressed the goals and objectives we set for the project. The model we developed does not cover all aspects of VMS. In this section we describe the challenges we addressed and present some details on the model we designed. In particular, we discuss the VMS process, the set of VMS servers that create the VMS operating system (personality) environment, and a number of VMS mechanisms affected by our approach.

2.1 Challenges

An important milestone for the project was to understand the VMS personality environment and to develop a model of VMS that includes the functions, policies, and mechanisms needed to support the execution of VMS non-privileged (user) images.

[1]UNIX is a registered trademark of UNIX System Laboratories, Inc.

There were two issues that we wanted to address with this model: VMS dependencies on the VAX architecture and interdependencies between VMS subsystems. Had we chosen to address just VMS dependencies on the VAX architecture, the model would focus on replacing VMS kernel primitives with Mach 3.0 primitives, leaving the rest of VMS alone (a single-server approach). To deal with interdependencies between VMS subsystems as well, we chose to pursue a multi-server approach where the VMS operating system is partitioned into a set of servers that act on behalf of user (client) processes.

Our multi-server approach raised some interesting issues with regard to data sharing within VMS, and between VMS and user processes (including protection of certain data from access by user processes). Moving VMS subsystems into separate address spaces invalidated the shared memory assumptions made by the existing subsystems. Due to the way the VAX architecture partitions virtual memory, alternatives were needed to handle:

- unprotected data in process (P0) address space that is accessed by VMS subsystems.
- protected VMS subsystem data that currently resides in process (P1) address space. VMS subsystems, executing under process context, have access to per-process data areas in P1 space.
- protected data areas shared among VMS subsystems in the shared system (S0) address space where VMS is itself located. There are many control-block-based structures in VMS that contain pointers to other structures and affect multiple VMS subsystems.

In addition, we did not want to assume, as VMS does, that four access modes are available to protect data.

Our model impacted a number of mechanisms involved in the execution of a VMS process. Some of these mechanisms depend on VAX architecture features, whereas others are affected by our multi-server approach. These include:

- asynchronous system traps (ASTs),
- exception and condition handling,
- interrupts, interrupt priority level (IPL), and spinlocks,
  and
- scheduling-related mechanisms (*e.g.,* I/O completion, event flag wait).

We also wanted to consider the implications of multiple operating system personalities on one kernel and the possible interoperability that could be exploited. One example is allowing for multiple command language interpreters (*e.g.,* VMS DCL, POSIX [8], DEC/Shell, *etc*.). Another is investigating generic servers that multiple personalities might share (or at least partitioning servers into personality-independent and personality-dependent pieces).

2.2 VMS Model Design

In addressing the challenges just discussed, our model of VMS has the following features:

- The VMS process, called the user client, consists of a pair of Mach tasks. These two tasks provide user images with a VMS application environment.
- This user client is partitioned into operating system personality-independent and personality-dependent pieces.
- VMS servers run in user mode and do not depend on processor architectural features.
- User-generated VMS actions (like setting an event flag or delivering an AST) are implemented in an architecture-independent manner.

An overview of the model and some details on these features are presented below.

2.2.1 Overview of the Model

The VMS-on-Mach model consists of two basic types of tasks: VMS server tasks and VMS user space tasks, all depicted using circles in Figure 1. The VMS server tasks provide the VMS personality environment that runs on the Mach 3.0 kernel. Services provided by this environment are used by VMS user images executing within a user space task.

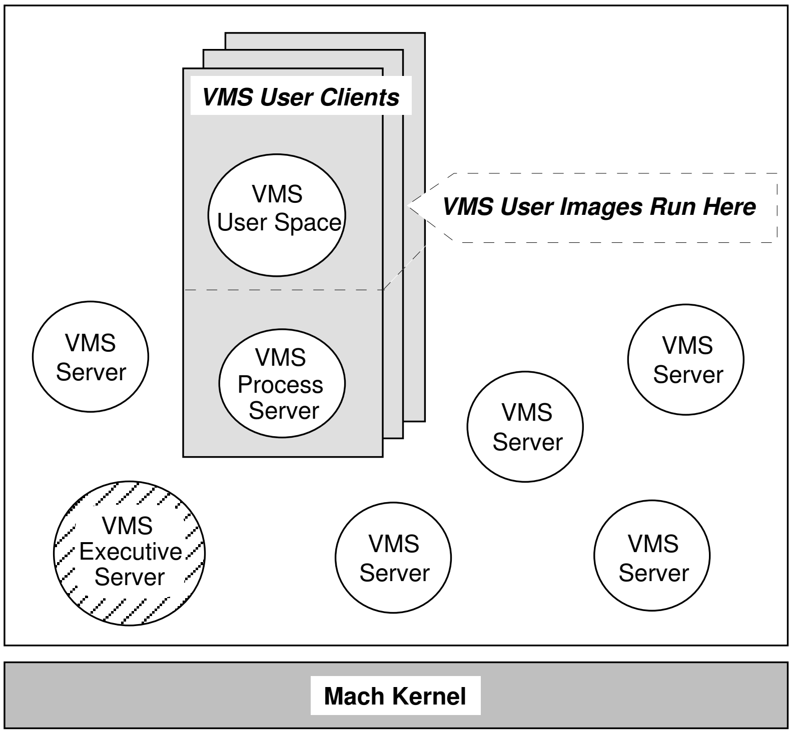

Figure 1 VMS-on-Mach Model

Each VMS process (user client), depicted by a shaded rectangle in Figure 1, consists of a VMS user space task and a VMS process server task. When system service requests are made by VMS user images, these are translated within the VMS user space task into requests that are acted upon by that task or, using Mach IPC (interprocess communication) services, by other VMS server tasks. The VMS process server task has special responsibility for providing the address space within which process-protected code and data reside (including loginout, command language interpretation, and image activation). The VMS executive server (seen in Figure 1 with diagonal shading) has special responsibilities that include bootstrap of the VMS environment. Note that hardware reboot is not required to restart VMS-on-Mach. The executive server is also responsible for managing server registration and access, as well as for managing the complete set of VMS user clients (processes).

2.2.2 The Process in the VMS Model

In order to preserve the behavior of the VMS process in our model, the user client evolved into a pair of closely-coupled Mach tasks. The particular behavior we wanted to preserve included:

- separation of the process itself from images executed within the process (in particular, protection of the process from erroneous user images),

  and

- protection of certain code and data from access by users.

One task, the user space task, provides the address space within which VMS user images are executed. Calls to the VMS application programming interface (API) by an image are acted upon by a read-only user library that resides in each user space task. The user library implements entry points for VMS system services in what is known as the VMS transfer vector. These entry points redirect system service calls to the routines which implement the services. Since the transfer vector is at a fixed location in the user image's address space for all VMS systems, programs linked against the transfer vector are assured that they need not be relinked for subsequent versions of VMS. This guarantee can be made because changes internal to VMS are hidden from applications by simply changing the contents of the transfer vector. In our model, system service calls are translated by the user library into service requests that may be handled directly within the user library, or directed to the appropriate VMS server task. All of this activity is invisible to the user image, allowing some freedom for the underlying operating system environment to evolve and change.

The user space task is created and managed by a second task, the process server. This server contains code and data that are protected from user access. The process server is designed to accommodate other operating system personality environments in addition to VMS. Code and data that are generic with respect to the operating system personality environment, like system access validation, receipt of exceptional conditions, and communication with other servers, are separate from and used to invoke operating system personality features, like command language interpretation, system services, image services, and exception processing. In our model, this personality-dependent piece is dynamically loaded during process server initialization.

This model of the VMS process also affected how the state of a process (its process context) is maintained. A main VMS software structure, the process control block (PCB), contains information managed by many VMS subsystems. With our model partitioning VMS into a set of servers, we decided to distribute the PCB information among the servers within the VMS environment. This favors performance of VMS servers in responding to user requests over performance of process-related information-gathering applications.

2.2.3 Servers in the VMS Model

Our model is based on a client-server approach. Individual subsystems within VMS are implemented as multi-threaded Mach server tasks. The functions of each VMS server are kept simple to provide basic building blocks from which the VMS personality environment may be constructed. This approach helped us focus on our goals of architectural independence and subsystem partitioning. With these goals, we see benefits to VMS that include:

- the ability to take advantage of widely-available and quickly-evolving hardware platforms with minimal effort and in a timely manner,
- improved reliability and maintainability,

  and
- increased flexibility to configure and evolve the system.

In order to provide some common framework for servers, as well as to isolate servers from the underlying kernel environment, we designed a server library. This library provides services that include basic task and thread management, synchronization mechanisms like mutexes and condition variables, error handling, inter-task communication, and memory management.

Although servers perform vastly different functions, their structure exhibits a number of similarities. Each server maintains static resources (typically data structures) for each client (typically a VMS user client). Servers provide communication ports to users and other servers. One or more threads in each server process requests sent to these ports.

Our model contains a number of servers that support the VMS environment. The remainder of this section discusses several of the basic servers and mentions a number of others servers included in the model.

Most functions performed by the VMS memory manager are replaced by equivalent functions within the

Mach kernel. Two memory management servers are included in our model to provide additional VMS memory management features: a file-mapped memory manager for implementing virtual address space mapped to files, and a shared memory server to implement shared address space mappings. The shared memory server relies on either the default Mach memory manager (pager) or the file-mapped memory server to manage backing store.

In our model, the Mach kernel creates a task for the VMS executive server. The executive server is the boot server that brings up the rest of the VMS personality environment. Once it has initialized, each VMS server registers its communication ports with the executive server. The executive server maintains these communication ports and makes them available to requesting servers. During normal system operation, the executive server creates VMS user clients, globally manages these processes, responds to queries for process information, and eventually deletes the user clients. In doing so, the executive server maintains the VMS notion of jobs.

The process server that manages each VMS user client provides three categories of services: system interface services, API services, and process state management services.

- System interface services include process initialization, system access validation, loading and presentation of a selected operating system interface (command language interpreter or shell and image services), and process termination.
- API services respond to requests from the user library on behalf of user images.
- Process state management includes services that maintain process attributes such as privileges and resource limits, execution state, exceptional conditions, and user space task control.

Other VMS features or subsystems that we modeled as servers include: accounting, auditing, authorization, common event flags, device drivers, error logging, the file system, I/O services, license management, lock management, logical naming, mailboxes, queue management, record management, the security database, and transaction processing services. This is not a complete list, but is indicative of the approach we used to partition VMS.

2.2.4 Internal Mechanisms

This section explores how VMS internal mechanisms, some of which are visible to users, are handled in our model. Certain mechanisms are no longer used within our model but are preserved for use by VMS applications (sometimes with changes that may be visible to applications).

2.2.4.1 Agendas and Notices

User-invoked system services often see delays in the processing of requests and usually have special semantics associated with the completion of these requests. In order to implement such user-requested VMS actions in an architecture-independent manner and to keep VMS server interfaces simple, we use a mechanism consisting of agendas and notices. Agendas and notices provide a single mechanism for storing and processing the semantics of actions to be taken when a service request completes. These actions, like setting an event flag upon completion of an I/O operation, remain local to user clients. An agenda structure is created by the client to remember the completion semantics associated with an operation. A unique identifier associated with this agenda structure is passed as part of the server request. When the server completes the requested operation, a notice message including the agenda structure identifier and any request-specific status or data is sent back to the client for processing. Pre-defined agendas and notice messages are also used to handle unsolicited operations, like a hardware exception.

2.2.4.2 Asynchronous System Traps (ASTs)

An AST is a mechanism that enables an event to alter the flow of control in a VMS process. Users may request AST notification when using VMS system services, and VMS itself requests ASTs as a result of some operations. Our model replaces the use of ASTs within VMS by notices and agendas, while maintaining the AST mechanism for user images.

When a user image requests a service that will eventually queue an AST, an agenda structure containing the AST function address and parameter is created. When the service request has completed, a notice message is issued back to the user space task. A thread within the user library suspends the user image thread, redirects it (by pushing a call frame on the stack) to an AST dispatch routine, and then resumes the user image thread. This dispatch routine calls the user image's AST routine with its associated AST parameter. When the user's AST routine completes, a message is sent back to the user library thread indicating AST completion. AST completion indicates that subsequent ASTs can now be delivered. (VMS dictates that only a single user AST may be active at any one time.) Note that the chain of stack frames is not guaranteed to be the same as in the current VMS implementation.

VMS servers use agendas and notices to mimic the function of ASTs. Servers maintain agenda structures to keep track of asynchronous operations currently in progress. Upon receiving a notice message, a thread within the server executes the functions specified by the agenda.

We also looked into the possibility of problems that could result from deviating from the existing VMS AST implementation. One of these deviations is the lack of access mode prioritization of ASTs and another is execution of code (usually privileged) in user address space. Our investigations uncovered only a few cases that needed to be addressed, all of which could be handled using Mach-provided functions.

2.2.4.3 Exception and Condition Handling

In the Mach exception handling facility [3] there is machine-dependent and machine-independent kernel software to deal with exceptions. Machine-dependent software implements exception dispatch and servicing that is specific to each hardware platform. The exception servicing code packages the information for processing by machine-independent software. This processing often consists of sending the exception out of the kernel for handling and then following up on the results of that handling. This model works well for handling VMS exceptions and conditions. When a user image exception occurs, a remote procedure call (RPC) message is sent to the process state management port within the appropriate process server. Processing of a user image exception causes the user image thread to be suspended and an IPC to be sent to the user library thread to initiate a search for a condition handler. While the condition handler search proceeds in the context of the user space task through user library thread actions, the process server returns from the RPC. Note that the chain of stack frames is not guaranteed to be the same as in the current VMS implementation. If the search is unsuccessful, or if all handlers resignal, the user library thread issues an IPC back to the process server indicating that the condition was not handled successfully.

2.2.4.4 Interrupts, IPLs, and Spinlocks

In VMS, interrupts (like exceptions) are events that require the execution of software other than the current thread of execution. Unlike exceptions, however, interrupts are unrelated to the current thread of execution and are asynchronous to it. To arbitrate among interrupt requests, each request has an associated interrupt priority level (IPL). When an interrupt is granted, processor IPL is raised to that of the interrupt request and the interrupt is handled by a service routine. Interrupt requests with an IPL that is the same or lower than the current processor IPL are blocked.

IPL applies separately to each processor in a multi-processor configuration. When symmetric multiprocessing (SMP) was implemented in VMS, spinlocks were introduced to synchronize access to shared kernel resources. To prevent deadlock, spinlocks are ranked according to their associated IPL. A thread may not acquire a spinlock with an equal or lower ranking than any spinlock currently held by the thread.

In our model, the server library provides low-level primitives for manipulating threads of control. These primitives include forking and joining of threads, protection of critical regions using mutex variables, and synchronization by means of condition variables. Spinlocks are replaced by mutexes that are similarly ranked. VMS server threads may acquire any mutex desired, but all mutex acquisition must be done in order of increasing rank. To acquire a mutex of lower rank, a server thread must first release all higher ranking mutexes. The scheduling effects of VMS spinlocks are not maintained by VMS server mutexes. Although we have not investigated these effects, a similar change occurred when BSD UNIX was transformed into a single server on Mach and no noticeable effects were observed. In summary, we rely on a server mutex acquisition protocol to achieve the required synchronization.

Hardware interrupts related to I/O devices are caught and serviced in the machine-dependent code of the Mach kernel. This is similar to interrupt-handling code currently in VMS. Where in the current VMS implementation the interrupt handler causes further processing by posting a lower-level interrupt to awaken a VMS fork process, in our model further processing is caused by the interrupt handler sending a message to the appropriate device driver server.

VMS threads of execution that are started as a result of software interrupts do not exist in our model. This activity is mimicked by VMS server threads and server library functions in the model.

3 VMS Prototype Implementation

The VMS-on-Mach prototype is a subset implementation of our model. During its development many design choices were made to facilitate its implementation. At present, our prototype is capable of supporting multiple VMS processes and running several existing VMS images. In addition to the many test programs we wrote during its development, the prototype successfully executes the following standard VMS images without modification:

- CREATE, used to create a text file,
- COPY, to make a copy of a file,
- TYPE, to list the contents of a file,
- SEARCH, to search for a string within a file,
- DIFF, to compare two files and report differences,

  and
- EDT, to perform line-mode editing on a file.

Figure 2 at the end of this section illustrates a sample session using the prototype, the development of which is discussed in greater detail in the following sections.

3.1 Approach and Objectives

Early in the project we decided that we would use a VAX platform to produce the prototype. We chose the VAXstation 3100-48 system after discussions with the CMU Mach team, who provided us with a version of Mach 3.0 on that system. We decided that this choice would best help us in demonstrating our model as it would allow for the execution of existing VAX/VMS non-privileged (user) binary images. We considered using a MIPS-based DECstation to demonstrate architectural independence for our prototype. This approach was rejected, however, as it would not have allowed us to use existing VMS sources or images.

We also wanted to take advantage of existing VMS and UNIX tools as much as possible, rather than create a new operating system and software development environment from scratch. This resulted in a VMS-on-Mach prototype that incorporates a set of servers into the existing Mach 3.0 based 4.3 BSD UNIX-compatible single-server system [6]. We built VMS servers as separate Mach UNIX processes, allowing us to take advantage of Mach threads and the BSD server. These VMS servers are for the most part new C code. Our decision to rewrite versus rework existing code revolved around how expedient it was to understand and rewrite the code, while considering how much effort was required to utilize existing sources. For most of the prototype work done to-date, rewriting was more expedient. This may or may not be true in a more complete prototype that includes, for example, the VMS record management and file systems. Our prototype implements only a small subset of the existing RMS interface on top of the UNIX file system (UFS).

Our objective for the prototype stage of the project was to implement a subset VMS environment that:

- involves multiple server tasks,
- isolates platform dependencies,

  and
- allows for some existing VMS non-privileged (user) binary images to be executed.

There were two phases to the prototype effort: a framework building phase, during which the infrastructure needed to execute simple VMS images was created, and an environment building phase, which added VMS services needed to execute more complex images. These phases are described in the following sections.

3.2 Framework Building Phase

Our goal for this phase, which lasted about three months, was to execute a simple VMS image. Work during this phase of the prototype effort concentrated on setting up the development environment, creating a server library, and building an implementation of the process server.

Our development environment is structured as a set of hierarchical directories which reflect the organization of the model. This structure is deliberate and designed to prevent unexpected cross-dependencies between the various components of the prototype. Because our software is designed using a client-server approach, there are many cross-component dependencies. This led us to develop a three-phase software build process. The first phase, interface export, descends the hierarchy generating and exporting public interface files (*e.g.,* generated IPC/RPC definitions and routines, and server-specific service definition files). The second phase, object library generation, also descends the hierarchy, this time building object libraries and exporting these libraries up the hierarchy. The final phase, image generation, creates the various servers by linking the required libraries for each image.

During early design discussions we investigated an optimization that would combine or bundle servers together into a single image. This optimization would allow for easier development and testing while providing optimized performance. These discussions evolved into a design philosophy that allows (in fact, requires) servers to be developed and tested independently of other servers, but permits these servers to be bundled into a single image at image generation time. This design philosophy is incorporated into our vbundling tool. The vbundling tool reads a file that describes all of the servers in the prototype and all of the server images that are to be generated. This tool allows us to alter the bundling configuration of the prototype by modifying a single file (the vbundling script) and rebuilding the prototype. Note that a change in the vbundling description file results in only the relinking of server images.

The first component of the prototype to be developed was the server library. Although our prototype was developed on Mach 3.0 using Mach UNIX facilities, we decided to hide as many of the underlying kernel and UNIX services as possible. This means that, in general, any Mach or UNIX service used by a server is provided by the server library. In some cases, however, it was not practical to build a server library interface to a Mach or UNIX service due to its use in a server as an expediency. For example, the authorization server uses the getpwent() UNIX service directly to scan the /etc/passwd file.

Initially, we defined three areas that the server library needed to address: memory management, server threads, and kernel task and thread management. Since each server is a UNIX process, the memory management services are implemented using the malloc() and free() functions. Server threads (sthreads) are a mechanism that allows servers to create multiple logical threads that either share or map directly to kernel threads. Since sthreads are close in function to the cthreads package [10] provided with Mach, we chose to implement them as a layer on top of cthreads. Since our design has VMS images executing within the context of pure Mach tasks (*i.e.,* not UNIX processes), we provided services in the server library for creating and managing Mach tasks and threads. To support architectural independence, we added functions for managing user image call frames that are used to build initial as well as interrupt call frames (*e.g.,* VMS ASTs or UNIX signals).

Our model defines several generic services that are used in the management of VMS processes (*e.g.*, process creation and termination). These services are defined as subsystems using the Mach Interface Generator (MIG) [10], which creates client-side and server-side bindings for each service as well as a subsystem dispatch routine. Subsystem dispatch routines are responsible for demultiplexing the set of IPC/RPC messages used by the subsystem into calls upon the appropriate server routines. Due to our server bundling methodology, each server's server-side binding for a particular routine must be uniquely named in order to avoid name conflicts when bundling several servers into a single image. As a result, we needed to define services that had only a single client-side binding but multiple server-side bindings. Our design also allows IPC/RPC service messages defined by different subsystems to be received and processed on a common port. Because of this, we could not use the MIG-provided library function to receive and dispatch messages to their service routines. To solve this problem, we implemented a server library object, the demux, which allows several subsystem dispatch routines to be grouped together. We then added a server library function that receives and processes the messages using the dispatch routines associated with a demux. This function optionally creates new threads for the processing of each received message.

Support of VMS image activation was initially developed in C on a VMS system, where we were able to verify the actions of our implementation with information obtained using the VMS ANALYZE/IMAGE program. Once the image activation code was debugged, we moved it over to our prototype. As an optimization, we load and prepare the user image within the process server's address space, and make use of Mach memory inheritance to set up the user image address space. This presented us with two problems: conflicts in the base addresses of the process server and user image, and conflicts in the location of the VMS transfer vector and the Mach UNIX emulator. VMS images have a default base address that overlaps addresses in use by the process server. As mentioned earlier, VMS uses a transfer vector to process system services. The location of the VMS transfer vector overlaps the Mach UNIX emulator that is part of every Mach UNIX process. We solved these problems by using existing VMS tools. VMS compilers generate position-independent code. This made it possible for us to use a VMS linker option to alter the base address of the image (thereby moving it out of the way of the process server). We also created an alternate version of the system service transfer vector at a different address. This allowed us to generate images that do not overlap the process server or the Mach UNIX emulator, and that are essentially the original user images. These changes allow us to load and prepare VMS images within the process server's address space.

Our next step was to build a framework for running VMS images. Our first user library function was implemented to support the starting of a VMS image. This function is called with the image's starting address information and is responsible for calling the image's main routine. Image completion, including any final status value, is signaled to the process server by the user library when the image's main routine returns or the $EXIT system service is invoked. Our initial VMS images were run by creating an sthread within the process server that executed the user library start function. Our next step was to logically separate this mechanism from the process server by using IPC for synchronization. Finally, we created a Mach task and thread to run the VMS image in a separate address space. This last step (separate tasks) was particularly difficult as we did not have a debugger that operated on pure Mach tasks.

3.3 Environment Building Phase

Our goals for this phase, which lasted about three months, were to expand the prototype and demonstrate a subset VMS environment that allowed users to login and logout, as well as execute some VMS programs. These programs included variations on *hello world* and existing VMS utilities.

The Digital Command Language (DCL) is the primary command language for VMS. DCL differs from other command languages in that the syntax for all commands is described in a set of files that are compiled into the special VMS image known as DCLTABLES. Each DCL command either invokes an internal DCL function or executes a specific VMS program. As we did with image support, we developed C code on VMS that read and parsed DCL commands. When our code was ready, we moved it to the prototype and integrated it into the process server. This required modifications to the existing image services to support reading the DCLTABLES image.

We also developed a simple authorization server that validates usernames and username/password combinations against the UNIX /etc/passwd authorization file. The process server was modified to request login information and validate the login request by using services provided by the authorization server.

At this point of the prototype we had two servers: the process server and the authorization server. Communication ports for these servers were managed by the system name server (snames) provided with Mach UNIX. Recognizing that we would soon start building additional servers, we developed a set of port name services within the server library to provide port registration and look-up functions. These services, which were initially developed as an interface to the system name server, were later modified to use the executive server as the prototype's port name server.

The executive server was developed with the primary intention of managing servers and processes in the prototype. Initially, the executive server acted only as a name server which processed server and process registration requests and converted port name look-up requests into authorized ports. As the prototype evolved, the executive server was enhanced to support system and process information queries, and to read a configuration file containing start-up information. Port name services are provided to other servers by the server library (which handles communication with the executive server). Information needed to process system queries is defined in the executive server's configuration file, which also contains a list of server images that should be automatically started by the executive server. The executive server uses the UNIX fork() and exec() functions to start other servers. The executive server supports process information queries by forwarding these queries to the appropriate process server. This implementation allows cross-process queries to be directed to individual process servers in the prototype.

As the number of servers increased, we found it increasingly difficult to debug the prototype. Consequently, we added a debug facility in the process server. Ideally the debug facility should be associated with the executive server or system console, but since the process server was the only server that performed terminal input, it was the obvious choice. Each server was modified to contain an array of debug flags that could be enabled or disabled via special IPC calls. A command line interface was built that allows any process server to communicate with a specific server and enable or disable any combination of debug switches. These on-the-fly debug switches have proven quite valuable in debugging the prototype.

File and terminal services are provided by the I/O server. Given our time and resource constraints, we first identified the minimum set of services we needed. The I/O server provides disk I/O that is used by image activation, the file-mapped memory server, and all file and data operations required by the VMS record management (RMS) services implemented in the user library (open, close, read, write, and flush). It additionally performs all terminal input and output for the prototype. The I/O server is implemented using the UNIX file system services provided in the Mach UNIX environment. Its public interfaces, however, provide services that are more generic than the existing UNIX interfaces, and are used to facilitate the implementation of RMS in the user library. All reads and writes are currently limited to a maximum of 512 bytes per operation and are implemented using the UNIX read() and write() functions (*i.e.,* they are treated as raw reads and writes).

VMS uses the notion of section files to refer to disk files that are mapped into memory. The most common use of section files by VMS is image activation. Each VMS image contains multiple image sections, each of which is a series of 512-byte blocks that are mapped to various locations in memory. The file-mapped memory server implements section files for the prototype. The process server communicates with the file-mapped memory server to initialize mappings for various portions of image files. The file-mapped memory server creates memory objects to represent these mappings, and is the Mach external pager for the objects.

As our prototype development continued, we were able to optimize many user services by moving them from the process server into the user library (*e.g.,* condition handler search and AST processing). This led us to revise our model and design of the user library. In the new design, the user library contains a thread which acts as the first-line manager for the user image. This thread receives and processes completion notices from other servers and initiates condition handler searches and AST delivery.

When the user address space is created to run a user image, the user library thread is started. This thread initializes the user library and informs the process server when initialization is complete by returning a port which the process server can use to communicate with the user library thread. Upon receipt of this user library port, the process server starts the user image thread, which is responsible for calling the main routine of the image. When the user image exits, an image completion message is sent to the process server, notifying it to terminate the user space task and resume processing in the command language interpreter (CLI). Note that if either the user image thread or user library thread incurs an exception that is not successfully handled, the process server terminates the user space task and resumes processing in the CLI.

VMS uses logical names quite extensively. Logical names provide a mapping of a string to zero or more replacement strings. They are grouped into tables which are either private to a process or shared within a process tree, a login group, or across the system. Note that VMS provides extensive mechanisms for protecting and authorizing access to shared logical name tables. A simplified logical name server was implemented in the prototype. This server implemented a single logical name table that is unprotected and is shared among all servers and processes. The logical name server supports a configuration file that allows for the definition of names. RPC services are provided for requesting logical name translations. These services are used by the process server and user library to resolve VMS file names.

In summary, our prototype implements the following components:

- a user library that implements the VMS transfer vector and system services,
- a server library that isolates kernel primitives from the servers,
- an executive server that provides server and process registration and look-up services,
- a process server that implements DCL and manages the user space task,
- a file-mapped memory server that provides support for section files,
- an I/O server that provides access to terminals and files,
- an authorization server that validates login requests,
- a logical name server that provides name translation services,

  and
- a common event flag server that provides multi-process synchronization services.

Figure 2 on the next page shows a sample prototype session that illustrates the use of the process server, the executive server, the authorization server, the I/O server, the file-mapped memory server, and the logical name server.

```
VMS-Process

       VMS-On-Mach

Username: VMS_TEST

Welcome to VMS-On-Mach (Process-Server X1.0)
VMK TEST CLI X1.0

Your default directory is VMK$:[VMS_IMAGES]

$ type hello.world
Hello world!

This file contains 9 lines, and serves as demonstration of I/O
via the I/O server within VMS-On-Mach.  The user program (TYPE)
is reading this file one record (line) at a time using the RMS
$GET system service.  Output is being performed to the terminal
by the I/O server in response to RMS $PUT system service calls.

Last line of hello.world file
$ edit hello.world
    1       Hello world!

*s/world/USENIX/
    1       Hello USENIX!
1 substitution

*exit hello.usenix
VMK:[VMS_IMAGES]HELLO.USENIX 9 lines
$ diff hello.world .usenix
System service $GETCHN not implemented
************
File VMK:[VMS_IMAGES]HELLO.WORLD
    1   Hello world!
    2
******
File VMK:[VMS_IMAGES]HELLO.USENIX
    1   Hello USENIX!
    2
************

Number of difference sections found: 1
Number of difference records found: 1

DIFFERENCES /IGNORE=()/MERGED=1-
    VMK:[VMS_IMAGES]HELLO.WORLD-
    VMK:[VMS_IMAGES]HELLO.USENIX
$ 
```

Figure 2   VMS-on-Mach Prototype Session

4 Conclusions

Developing the VMS model and implementing the prototype software was a valuable experience for us. In this section we present lessons we learned and future directions we believe useful to pursue.

4.1 Findings

We found the Mach abstractions helpful both in designing the model and in implementing the prototype. The client-server approach that Mach supports was very useful conceptually in designing our partitioned model of VMS-on-Mach. In implementing the prototype we also found that the partitioning lent itself to ease of construction and debug. However, we also found issues for Mach. This section describes these findings.

4.1.1 Authentication

We found it efficient to have VMS servers conduct authentication once for each client, and on success reward the client with the assignment of a server port. Since Mach tasks cannot manipulate ports except via Mach kernel calls, we can be sure that all messages delivered through this port originate from the client to which the port was assigned. Furthermore, Mach developers pointed out that the port name can be set to a virtual memory address, and, since this name is supplied by the Mach kernel for received messages and cannot be forged, it can be used to point directly to the data structure element corresponding to the client to which the port was assigned. This eliminates a number of checks that would otherwise be required. Use of ports in this way means that for each class of service, where we would otherwise use a single service port, we now use one port for each client so that we can identify the client using the port. The total number of ports required is therefore on the order of the number of service classes multiplied by the number of clients. The resource implications of this require further study.

4.1.2 VAX Access Modes

The VAX architecture defines four access modes, which implement a protection-ring scheme for limiting access to data residing in the virtual address space. In place of rings superimposed on a common virtual address space, our model uses separate virtual address spaces in the form of separate Mach tasks, resulting in a more constrained strategy for data sharing.

The VAX architecture allows the current VMS implementation to access data of less privileged modes using direct memory references. With a client-server model, access to this type of data typically requires one or two IPC messages. Our model minimizes the need for these IPC messages by data partitioning and localized access. We did not, however, encounter any difficulties in preventing access to data of more privileged access modes.

4.1.3 Scheduling and Memory Management Policies

It is not feasible to reproduce the VMS scheduling policies using Mach. VMS decrements the size of the time quantum assigned to the currently executing thread each time the time quantum expires, and boosts the priority of the currently executing thread when involuntary waits occur. While the exact VMS semantics may not be possible, the Mach time sharing scheduling policy [2] does have the same basic goals as the VMS scheduling policy, namely to be responsive to interactive processing while preventing starvation. Therefore the difficulty in implementing the exact semantics of the VMS algorithm at the server level is not a major issue. An exception to this occurs in the case of deadline scheduling, which may be needed for implementing network protocols.

A similar situation exists with regard to memory management. Mach does not use the same mechanisms as VMS to balance free-page deficits. Swapping, for example, is not used by Mach. We also did not attempt to model VMS process working sets (the process' virtual pages that are currently valid and in physical memory). The Mach kernel does not currently export an association of resident pages to tasks or allow externally influenced per-task page replacement, both of which are needed to implement working sets.

4.1.4 Kernel Resource Management

Management of kernel resources is needed to isolate the ill effects of any one VMS user image from other VMS processes and servers, and indeed from future user images of the same VMS process. Tasks that do not cooperate must not be able to interfere with each other by tying up resources. Such management is performed within VMS by the use of quotas. The Mach kernel does not supply any way to impose limits, such as quotas, on tasks. In an implementation of a production operating system environment, a resource model is needed to control memory and processor usage on a task basis.

4.1.5 Device Drivers

In the version of the Mach kernel used in our model, device drivers reside in the kernel. While an interrupt service routine needs to be located in the kernel, it should provide minimal function, relegating most of the work to a user-level task. Also, device drivers can be partitioned to reduce the amount of replicated code. These modifications are being addressed by Mach developers at CMU [4].

4.2 Future Directions

We have not fully addressed a number of topics that we nevertheless feel are important to explore if VMS-on-Mach is to become a practical operating system environment. These topics are described in this section.

4.2.1 Performance Optimizations

In order for VMS-on-Mach to be a viable alternative, it must have performance comparable to that of VAX/VMS. Preliminary measurements indicate that unoptimized performance is likely to be unsatisfactory, but large gains can probably be made through judicious optimizations that do not detract from the advantages of the multi-server model. In order to identify the needed optimizations, we must carry out carefully designed measurements. Most of these measurements (and therefore most of the resulting optimizations) can only be made after more of the VMS-on-Mach model has been implemented. Our thoughts on possible optimizations are described here.

4.2.1.1 Proxies

A proxy [10] is a section of code and/or data resident in a client's address space, which acts as an agent for a server. The client may be a user space task or a VMS server. The proxy is considered part of the server and can be dynamically loaded into the client by the server. The advantage of using a proxy is that a client's address space can be accessed directly. Invoking a proxy incurs only the cost of a local procedure call instead of the cost of an RPC to the server.

One of the primary uses of a proxy is to provide a client-side access window to information maintained by a server. Proxy use can considerably reduce client-server communication, thereby reducing the overhead and latency of accessing the data.

Currently we have not identified any specific use of proxies, but we believe the concept would be useful in a full operating system implementation using our approach. For instance, one possible use might be in implementing I/O read buffering.

4.2.1.2 Distributed Data

Strictly speaking, the client-server paradigm used in our model dictates that a single server is responsible for managing a specific piece of data, and that all references to the data are performed by RPCs to the managing server. This philosophy makes updates cheap but references expensive, and is inefficient for data that is frequently referenced but seldom modified. For this reason, we envision relaxing this paradigm to allow copies of selected data to reside in multiple servers, with an RPC being used to update these copies. To date, however, we have not identified specific uses of this technique.

4.2.1.3 Message Transport Independent of User Interface

The Mach kernel provides in-line and out-of-line modes of message transport. With in-line mode, the data is sent along with the message; with out-of-line mode, the message specifies the data location but does not actually include the data. In-line mode is appropriate for short messages while out-of-line mode is preferable for long messages. The choice of passing data in-line or out-of-line is an optimization that should be made by the message subsystem. Software changes should not be required.

If two servers that send messages to each other are bundled into the same task, server code should not be required to optimize the transport mode used to send messages between these servers. For instance, shared memory might be utilized as the basis for an optimized transport mode.

To avoid intrusion of optimization decisions into the source code of VMS-on-Mach, we looked into designing a mechanism in our model so that the message transport mode may be chosen automatically, and the user does not need to explicitly indicate the transport mode.

4.2.2 Other Interesting Topics

A number of important operating system features and subsystems were not examined in our effort so far. These need to be incorporated into our model.

VAXcluster technology provides users with an available and distributed computing environment. Multiple computing resources are used to create a cohesive environment that facilitates the dynamic sharing of devices and files. The use of micro-kernel and client-server technologies provide us with additional opportunities for delivering distributed computing. In light of this we need to examine how these technologies might be used to provide for, and enhance, the features of VAXcluster systems.

We need to define the generic I/O, file system, and record management services that the I/O server, file system server, and record management server will provide, and identify the work that must be done in the user library to build an I/O subsystem for user images. High performance of the I/O and file system servers is crucial to the success of our model. This is achieved by ensuring that overhead-inducing layers of software can be bypassed if their capabilities are not required.

We are interested in exploring the issues related to orchestrating multiple operating system environments on the same kernel, as well as to cooperation among distributed (multi-) computing systems. For example, a philosophy is needed to provide for management of underlying kernel resources and multiple personality environments. In addition, thought needs to be given to the networking model for such a system.

References

[1] Accetta, Mike, Baron, Robert, Golub, David, Rashid, Richard, Tevanian, Avadis, and Young, Michael, *Mach: A New Kernel Foundation For UNIX Development*, Technical Report, School of Computer Science, Carnegie Mellon University, August, 1986.

[2] Black, David L., "Scheduling Support for Concurrency and Parallelism in the Mach Operating System", *Computer*, vol. 23, no. 5, pp. 35-43, May, 1990.

[3] Black, David L., Golub, David B., Hauth, Karl, Tevanian, Avadis, and Sanzi, Richard, *The Mach Exception Handling Facility*, Technical Report CMU-CS-88-129, School of Computer Science, Carnegie Mellon University, April, 1988.

[4] Forin, Alessandro, Golub, David, and Bershad, Brian, "An I/O System for Mach 3.0", *Proceedings of the USENIX Mach Symposium*, pp. 163-176, November, 1991.

[5] Gien, Michel, "Micro-kernel Design", *UNIX REVIEW*, vol. 8, no. 11, pp. 58-63, November, 1990.

[6] Golub, David, Dean, Randall, Forin, Alessandro, and Rashid, Richard, "UNIX as an Application Program", *USENIX Summer Conference Proceedings*, pp. 87-95, June, 1990.

[7] Goldenberg, Ruth E. and Kenah, Lawrence J., *VAX/VMS Internals and Data Structures, Version 5.2*. Digital Equipment Corporation, Order No. EY-C171E-DP, 1991.

[8] *IEEE Standard Portable Operating System Interface for Computer Environments, IEEE Std. 1003.1-1988*, Institute of Electrical and Electronics Engineers, Inc., 1988.

[9] Loepere, Keith, *MACH 3 Kernel Principles*, Open Software Foundation and Carnegie Mellon University, 1991.

[10] Loepere, Keith, editor, *Server Writer's Guide*, Open Software Foundation and Carnegie Mellon University, 1990.

[11] Rashid, Richard, "A Catalyst for Open Systems", *Datamation*, vol. 35, pp. 32-33, May 15, 1989.