Optimizing Performance of Mach-based Systems By Server Co-Location: A Detailed Design

Michael Condict
Dave Mitchell
Franklin Reynolds

OSF Research Institute
One Cambridge Center
Cambridge, MA 02142

August, 1993

Tel: (617) 621-7349
E-mail: condict@osf.org
dwm@osf.org
fdr@osf.org

Abstract. For the last couple of years OSF has pursued a variety of microkernel-based operating system projects. The performance of these systems, while encouraging, has been significantly less than similar integrated systems. Recently, we devised a new framework for dynamically loading Mach servers (into kernel mode or into other servers) and automatically short-circuiting RPCs into direct procedure calls wherever possible. We believe it satisfactorily answers the objections to previous proposals in this area and that its adoption will bring the performance of a microkernel-based system to within a few per cent of the performance of the equivalent integrated kernel. The techniques are independent of any particular OS personality and can be used to equal advantage in user mode or kernel mode and in single servers or multi-servers, so the framework can be used to improve the performance of most Mach-based systems. This document contains a detailed design, intended to be used as a reference in the implementation of this framework, which is beginning shortly.

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Material Command (AFMC).

1. Introduction and Background

A long-held performance goal for the microkernel-based implementation of the OSF/1 MK operating system is to obtain at least 90% of the performance of the integrated OSF/1 kernel (OSF/1 IK) under all reasonable workloads. This goal has always eluded us although we've come close in several areas and actually exceeded the IK in a couple areas (e.g. uncached file writing).

After extensive performance analysis, we now know much more about why the performance of MK is poor, and some of us are convinced that the 90% goal will never be met as long as the system call path involves two Mach messages and two user-to-user context switches. The operating system services are simply too far away from the user, compared to an integrated kernel. (For example, MK spends over 21% of its time in the system call transmission path when running the AIMIII benchmark, compared to about 1.5% for IK, so without reducing this path MK cannot hope to achieve better than 80% performance no matter what reductions in MK overhead are made elsewhere.)

These experimental findings are in direct contradiction to the argument made in [1], which is that the cost of RPCs has dropped to the point where it is an insignificant overhead, compared to the cost of carrying out the system service requested by the RPC.

This performance problem is magnified when the system services are further modularized into separate user tasks (a "multi-server" configuration). In this design, some system calls may require multiple RPCs (from a user to an OS server, which then may need to do an RPC to one or more other servers).

Finally, recent attempts to optimize the critical path of untyped Mach IPC, including the context switches, have produced only modest return on a large investment of time, leading us to conclude that there is not much room to improve the performance of this particular message-based system, short of a radical redesign and rewrite. (Which is not to say that much faster IPC isn't possible in a different system — Spring [4] performs user-to-user RPC, including context switch, at approximately 4 times the rate in Mach.)

This should lead us to ask why we want to use a microkernel-based operating system in the first place. Some benefits, such as making most of the OS pageable and pre-emptible, are not exclusive to microkernel-based systems. Other benefits include providing a location transparent interface to personality independent services, though the largest set of gains stems from the improved modularity of the microkernel architecture. Without enumerating them in detail here, let us just point out that all of these benefits accrue to the OS developer, not to the users. Therefore, the user will not be willing to tolerate any performance degradation, unless it is offset by something else, such as improved reliability, lower purchase price or availability on a wider range of architectures (any of these might plausibly result from the modularity improvement).

Given that improved modularity is the only significant reason for the microkernel architecture, let us examine how this modularity has been achieved and whether it can be accomplished in a way that is less damaging to performance. In Mach 3.0 based systems, the OS is separated into two or more modules: the microkernel, and one or more OS servers running as user-mode tasks. All of these servers are in separate hardware protection domains and cannot interact with one another except by RPC, implemented and mediated by the Mach kernel. Moreover, the Mach kernel chooses to interact with the servers mainly by RPC also, although it can and does use more efficient trap-based mechanisms for some of its performance-critical services (interestingly enough).

We share the opinion of [2] that the use of hardware protection domains (i.e. separate user tasks) is often an inappropriate way to address software modularity concerns. Crossing a hardware protection boundary is expensive, and is unnecessary when trusted code is on both sides of the boundary. The purpose of hardware protection is to protected trusted system components from damage by untrusted user code, to protect users from one another, or even to protect a single user's trusted code base from untrusted code run by that same user. None of this applies to the case of a trusted OS server (or servers) running on top of a microkernel. We maintain that the best way to achieve a good modular design in this case is to use a language that provides support for modularity, such as an interface definition language (e.g. MIG), but not implement it exclusively with RPC across separate hardware protection domains. However, fault containment does mandate that systems intended to be highly secure or highly available continue to run servers in separate domains.

For almost three years, the RI as a whole has been investigating ways in which the performance of Mach-based operating systems might be improved by various architectural changes, including shortening the system call path by moving some or all of the system services back into kernel mode or by loading the servers of a multi-server implementation into a single address space (this concept has been called "server compression").

Until recently, no satisfactory way has been found to compress the system services into kernel mode (or into a single user task) without losing some or all of the benefits of multiple operating system servers layered on top of an independent microkernel. This document specifies the design of a solution which preserves these benefits. We first describe it briefly, in the following section, and then in more detail in the rest of this document.

Note that a group at the University of Utah has recently published descriptions of similar research [3] which went on independently (and without our knowledge, for the first two months of our work). The goals of their project are somewhat more restricted than in ours, being primarily concerned with loading servers into the kernel, whereas we also consider the case of multiple servers loaded into the same user task.

2. Summary of the Design

There are two completely independent parts to this proposal (making it easy to divide up the implementation effort).

The first part consists of enhancing Mach to allow a fully linked program to be loaded into the kernel address space and to run in kernel mode (but not as an integral part of the Mach kernel) and devising a similar method for loading a program into an active user task. We call this part *Dynamic Server Loading*.

The second part consists of extensions to MIG to cause it to use more efficient implementations of RPC than a two-way message whenever possible. We call this *Short-Circuiting RPCs*. The two cases where it can be done are (1) when server and client are in the same address space (same user task, or both in kernel) and (2) when the client is in a user task and the server is in the local kernel. Two different techniques are required for the two cases.

As will be shown below, the combination of dynamic server loading and short-circuiting RPCs will provide an efficient path from user to kernel-loaded OS server, without giving up the modularity benefits of a microkernel based system.

2.1 Dynamic Server Loading

For loading servers into kernel mode, we propose adding a new Mach facility that allows a privileged user to create a new kernel task, load arbitrary data into it, and initiate a new kernel thread at a specified address in the newly loaded data (some of which presumably consists of instructions).[1] This would seem to require nothing more than exposing the task port of the kernel task to privileged users, and doing the rest of the work from a user task, using facilities such as task_create, vm_write and thread_create, but it turns out that one new kernel interface will be needed.

Instructions executed by the new kernel task will behave the same as if it were in user mode. This specifically includes the use of trap instructions to request Mach services. The kernel will be prepared to deal with such kernel-to-kernel traps, and will service the requests in the expected fashion. The result is that if the loaded code and data are a complete OS server that normally runs in user mode, almost no changes are required to run it in the kernel.

Second, an almost identical technique can be used to load a fully linked server program into a user-mode task that is already running. As with loading into kernel mode, we will arrange that the dynamically loaded program behaves the same as if it were loaded by the OS in the conventional manner. One difference compared to kernel loading is that the program loaded into the user task will have to run as part of the same Mach task, sharing a common port space. In the case of the kernel, the loaded program is a new kernel task, which isolates it from the rest of the kernel and allows it to be terminated separately.

---

1. Note that a Hitachi employee on sabbatical at OSF, Takashi Nishikado, has already done a lot of work with dynamic kernel module loading. Before he left he was able to load and unload the kernel debugger, a floppy disk driver and possibly other modules. We can also benefit from the experience with OSF/1's dynamically loaded kernel modules, although our proposal differs in that we want to load fully linked binary programs, not object modules.

2.2 Short-Circuiting RPCs

The second part of this proposal, Short-Circuiting RPCs, entails extensions to MIG that will cause MIG clients to automatically replace message-based RPCs with a more efficient form wherever possible. We propose extending MIG in the following ways:

When any of the client stub functions for a particular MIG subsystem is invoked, it will, before carrying out the RPC, determine whether it is in the same address space with the server that is the target of the RPC (with the server's help). If it has been linked into or dynamically loaded into the same user task as the server, it will use a direct procedure call to reach the server-side function that implements the service. This "short-circuiting" of the RPC will bypass the code that marshals arguments into a message header and invokes mach_msg. It will instead directly call the same function in the server that would have been called by the MIG-generated dispatching code on the server side. The same mechanism will be used when the client and the server are both in the kernel. (Note that this technique will also be used to good effect by the Mach code, which can be either a client or a server with respect to modules that have been dynamically loaded into the kernel.)

If the client is in a user-mode task, and the server is in the kernel on the same node, a slightly different technique will be used. After the MIG client code determines that the server is not co-located in the same user task, it will assume that the server might be in the local kernel, so it will use a new Mach "short-circuiting RPC' service that consists of little more than a trap into kernel mode, some copying of arguments and results between kernel and user space, and a call to the correct server procedure in the kernel. Note that this will allow any message-based kernel service, including both Mach services and non-Mach services dynamically loaded into kernel mode, to be accessed via a mechanism that avoids the use of messages and transfers control more directly to the kernel function that implements the service.

In either of the above two situations, the RPC will be based on a thread migration model, where the server is a passive code module that is that is temporarily entered by the client thread to carry out the request. This is very different from the 2-way message model, in which servers have a pool of threads that receive requests from clients and carry them out on behalf of the client.

Finally, if the attempt to short-circuit the RPC by trapping into the kernel fails, because the server is not in the kernel, the MIG client stub will fall back to the current method of using messages. This will be necessary when the server is in a different user task or is on a remote node.

Whenever client and server are in the same address space, the server will register the necessary information for the client to branch directly to the correct server procedure. This will be a different address for each stub in the MIG subsystem, so a jump table indexed by the request number will be used. When the client is in a user task and the server is in the kernel, this table will be mapped only in the kernel, so the client thread must first trap into the kernel before it can use the table. See "User Access to Kernel Services", below, for how this works in detail.

Finally, we will make another extension to MIG, to avoid unnecessary allocation and deallocation of port rights and out-of-line (OOL) memory that are passed as arguments in short-circuited RPCs (i.e. in direct procedure calls). In such a case, it makes no sense to pass to the server a copy of the OOL memory, which must be allocated and deallocated, when a direct pointer to the clients's memory could have been passed instead, with no difference in semantics.

The extension consists of adding new attributes that can be declared with MIG input parame-

ters that are ports or OOL memory. Minor changes are needed in the user code if it is to obtain the associated performance benefit, but the changes are not required for correct operation. See a later section for more details on this extension. (Note: this optimization was not part of the Utah work reported in [2].)

2.3 Copyin / Copyout

In addition to the flow of control in the system call path, it is also important to optimize another important aspect of performance: the server-initiated transfer of data between the user process and the OS that takes place during the evaluation of many system calls. Note that we are not referring to the implicit transfer of the input and output parameters in an RPC, but rather to transfers explicitly invoked by the OS server in the course of carrying out a request. In UNIX-based systems, this is commonly known as “copyin/copyout”, and the prototypical examples are the data buffers specified in the read and write system calls. If emulation code is being used in the client to send “system call” RPCs to the server, there may not be much need for copyin or copy-out, since the data can be passed as “in” or “out” parameters of the RPC. But an emulator-free system such as the newer OSF/1 servers will need to transfer all data this way, even the system call arguments themselves, because no user data is passed as an argument in the exception RPCs that are used as the system call path in these servers.

When server and client are in the same address space, copyin and copyout can degenerate into simple data copies, of course, but the interesting case is copyin/copyout between user tasks and a kernel-loaded OS server.

In current, user-mode UNIX servers, at least two different methods are used for copyin and copyout. While either one will work in a kernel-mode server, one is preferred, for best performance. See *Performance Analysis*, below for further discussion.

3. Performance Analysis

The implementation of this proposal will greatly improve the performance of a microkernel-based OS. This can be seen by analyzing the path of a typical system call using the new mechanisms proposed here, with the OS server loaded into kernel mode, and comparing it to the paths of a system call in current Mach microkernel-based systems and traditional, integrated kernels.

In current systems with user-mode servers (see Figure 1), a system call begins with a trap into the Mach kernel (1). The trap may directly result in a message to the OS server, or it may be redirected into an emulator which then traps again into the kernel to send a message. Any such redirection can be ignored, since it is negligible compared to the other costs. The kernel then context switches to a thread of the user mode server (2), which it has awakened to receive the system call message. The server may need to request one or more Mach services in the performance of its duties. Most of these require sending messages to the kernel (3), i.e. the overhead of a round-trip trap plus the overhead of Mach IPC. Finally, the server sends a reply message to end the system call, causing another trap into the kernel (4) and a context switch back to the user thread that initiated the system call (5). The grand total is at least two user-user messages, which imply two user-user context switches, not counting any Mach services the OS server may request.

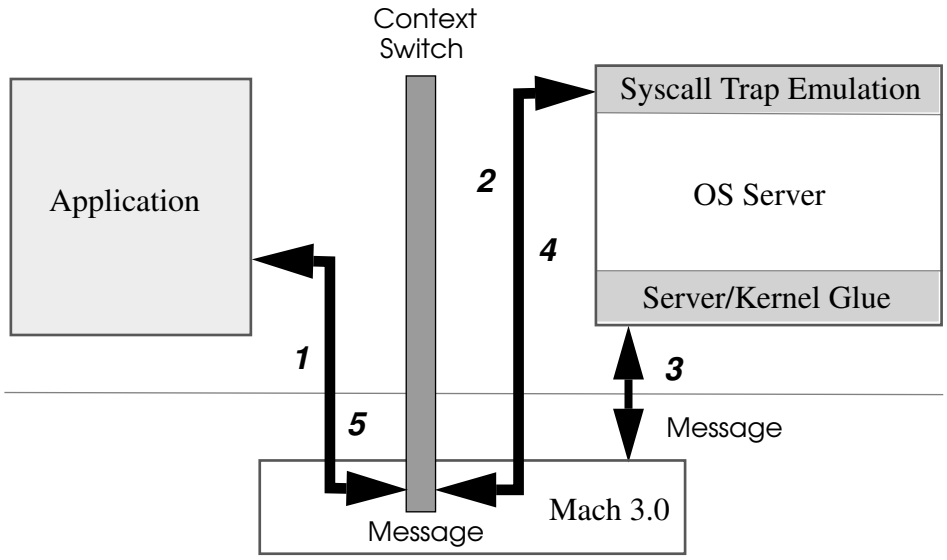

Figure 1: System call path with a user-mode OS server.

Now consider the system call path when the OS server has been loaded into kernel mode using the techniques described here (Figure 2). In the proposed architecture, the path starts with a trap into kernel mode (using the new Mach “short-circuiting” system call). Then the user thread can look up the kernel address of the correct function in the OS server. The same user thread that requests the service executes the OS server code, unlike in current systems, where there is a context switch to an OS server thread. The OS server then obtains any necessary Mach services by direct procedure call rather than by trapping into mach_msg, because it is loaded into the kernel address space together with the Mach kernel. There are no messages and no context switches at all, regardless of how many times the server requests a Mach service.

Finally, it can be seen that the proposal provides performance close to that of a Mach-based integrated kernel by comparing to the corresponding path in the integrated kernel. (Figure 3),. In

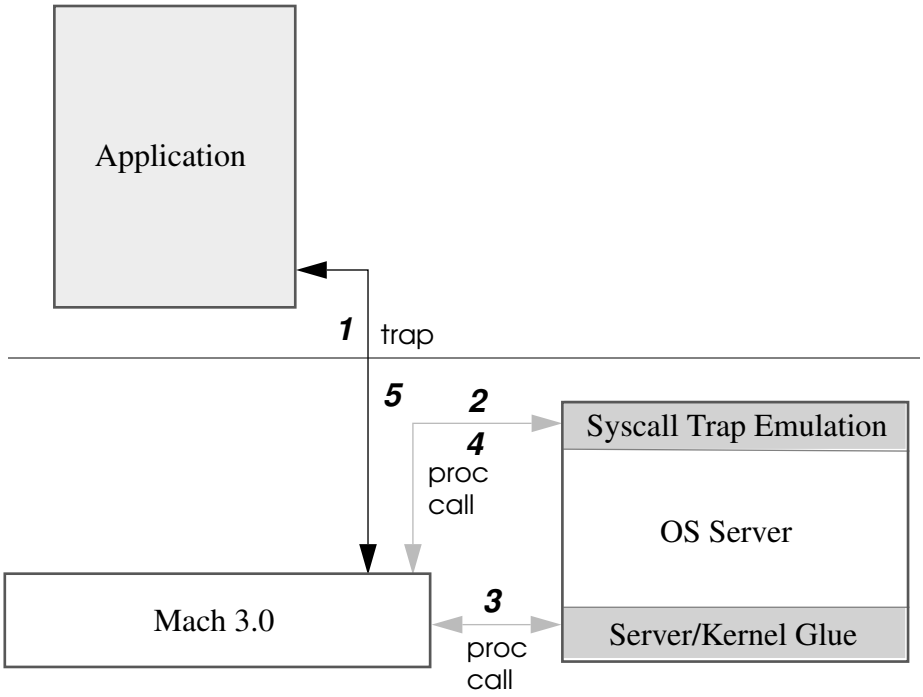

Figure 2: System call path with a kernel-loaded OS server.

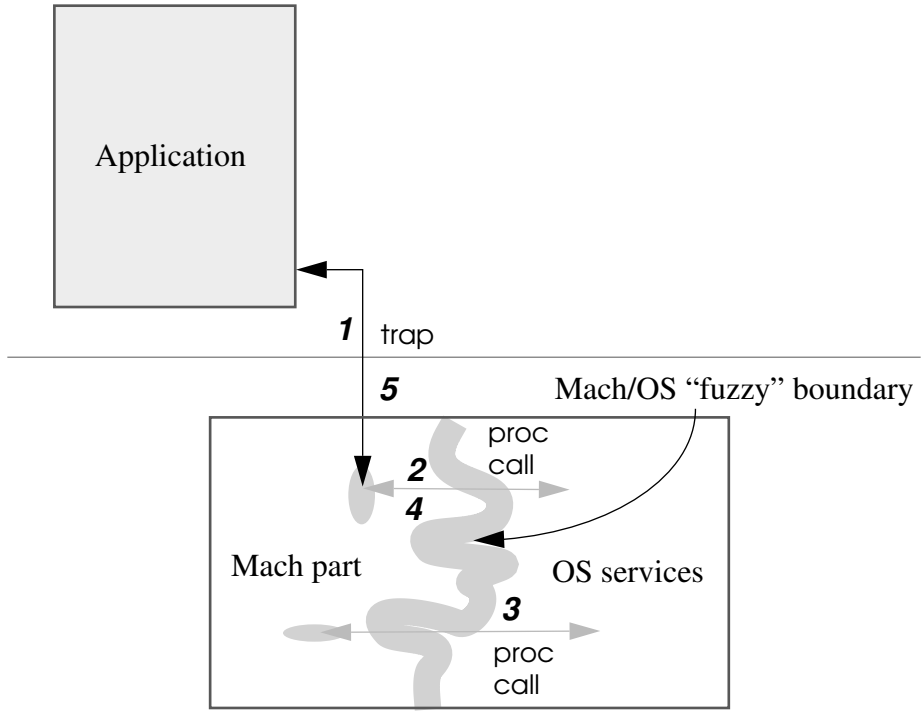

Figure 3: System call path in an integrated kernel.

an integrated kernel, a user traps into the kernel (1), which determines the correct OS function

based on the trap argument. The thread that trapped is transferred (in kernel mode) to the OS function (2), where it carries out the requested service, possibly requiring one or more procedure calls to the Mach part of the kernel (3). When finished, it returns back to the top-level trap interface of the kernel (4), which returns back to user mode (5), allowing the user to continue. The only important performance differences between the kernel-loaded server and the integrated kernel are:

- The procedure calls in the integrated kernel are slightly more efficient, since the short-circuiting of RPCs described in this proposal requires a few checks and pointer references (and possibly an extra function call) in addition to the actual call to the server function. This overhead should be almost undetectable, given the overhead of trap handling.
- The OS code in the integrated kernel can directly access the Mach data structures, whereas the dynamically loaded OS server must go through the documented Mach interfaces. But the prevention of this violation of modularity is the whole point of a microkernel-based OS. This proposal preserves the modularity at a small cost in efficiency, unlike the user-mode OS server, which enforces the modularity at a much higher cost.

Finally, the OS server returns back to the user task by a simple return from interrupt, like any integrated kernel. Unlike today, there are no context switches involved in the execution of a (non-blocking) system service. The system call path is essentially the same as would be found in an integrated kernel implementing the same functionality. This is why we can expect the same performance, to within a close approximation.

As stated in the previous section, we also need to analyze the copyin/copyout path, that is the performance of server-initiated data transfers between the user task and the OS server.

In the case of a user-mode OS server, this data transfer can be quite expensive. Various methods have been used in the past, including the following (we describe the copyin path, but copyout is similar):

- "Emulation" code in the user process copies the user data to a block of memory that is shared with the OS server (the sharing is set up by the server as part of process creation; see Figure 4). The server can then directly access the data. This entails an extra copy of the data compared to an integrated kernel. Also, it is not possible in a "no-emulator" architecture.
- "Emulation" code in the user process sends the user data "out-of-line" in the system call message (Figure 5). The server can then directly access a virtual copy of the data, but is required to deallocate the data when finished with it. This avoids the extra copy, but entails a significant amount of VM overhead and is often slower than the shared memory approach, on modern architectures. Furthermore, this technique cannot be used for copyout without an extra copy and deallocation. That is, because of possible page misalignment and the inability to control the destination virtual address of out-of-line memory, the data must be sent to the emulation code in the user task, which then copies it to the correct place in the application data and then deallocates it. Again, this method is also not possible in a "no-emulator" architecture.

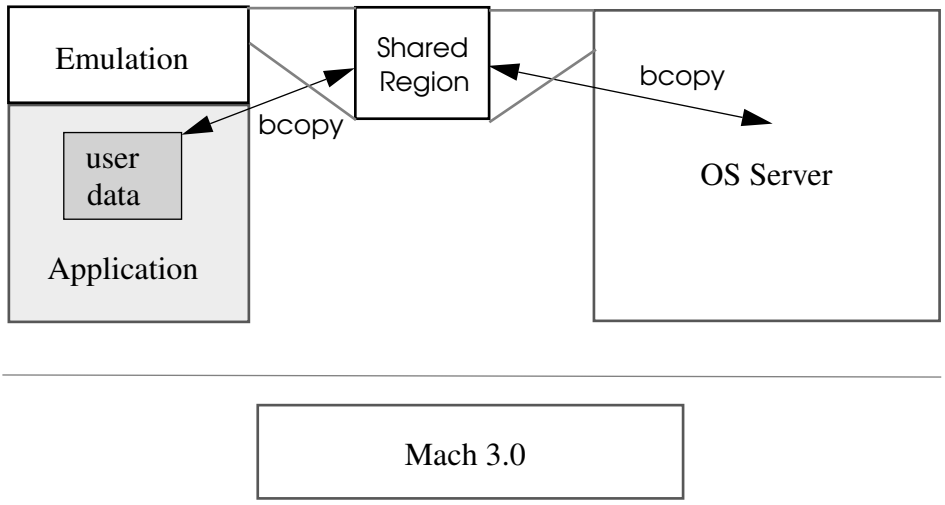

Figure 4: Copyin/copyout using static shared region.

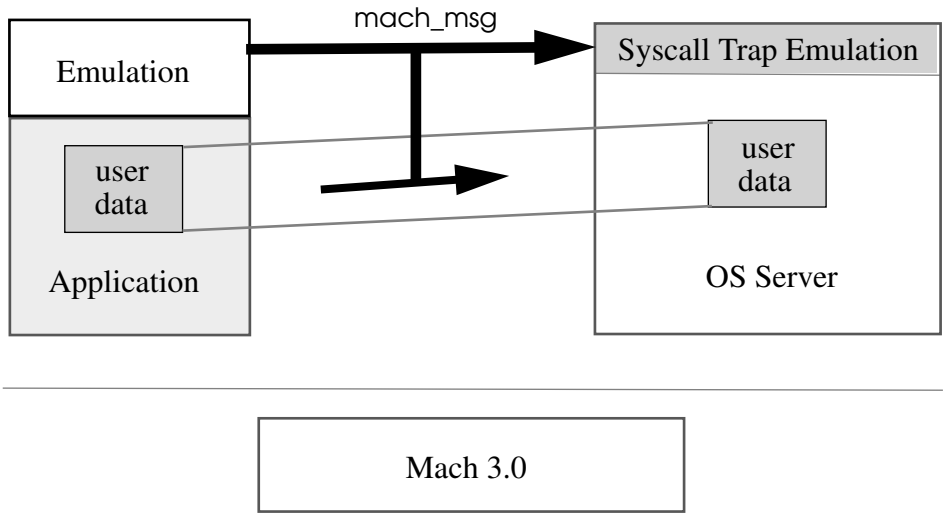

Figure 5: Copyin using out-of-line memory.

- The server can use `vm_read_overwrite` and `vm_write` to access the user data. This is slower than the either of the two previous approaches, since it involves extra traps into the Mach kernel and considerable VM overhead.
- The server can dynamically map any or all of the user's address space into its own (set up sharing between itself and the user's data), using the new Mach interface, `vm_remap`, designed for that purpose (Figure 6). This VM manipulation is too expensive for use on each copyin/copyout, but these shared memory regions can be cached in the server and reused on subsequent system calls. In a user-mode server, this is the fastest known

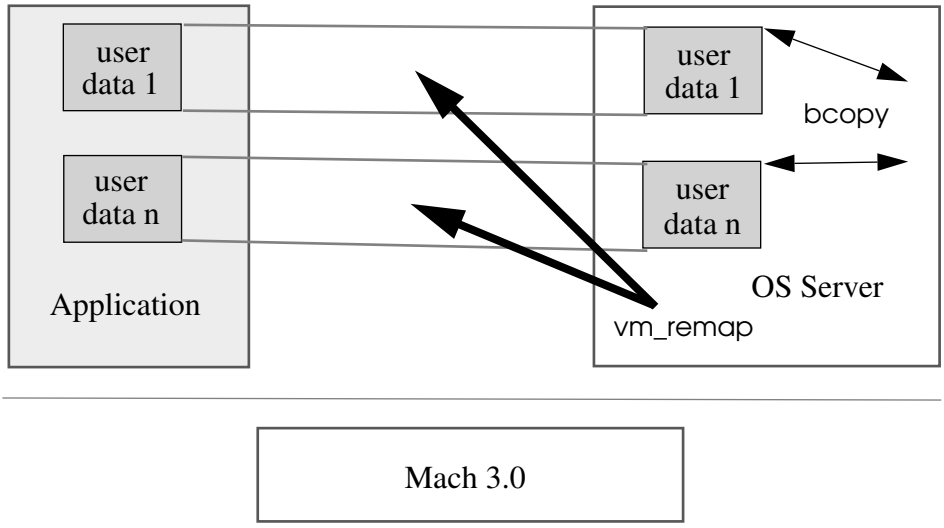

Figure 6: Copyin/copyout using vm_remap.

way to do copyin/copyout, but it is still significantly slower than in an integrated kernel, where (for many architectures) all of the user address space is directly accessible to the OS without any set-up on its part.

We want to regain the efficiency of an integrated kernel's access to user data for a server that has been loaded into kernel mode, while still doing the best we can when the same server is run as a user-mode task. This will probably require two different mechanisms for copyin/copyout, based on whether or not the server is running in user mode or kernel mode. In user mode, vm_remap is the best choice, as described above. But in kernel mode, the implementation of vm_read_overwrite and vm_write will be the same as an integrated kernel's implementation of copyin/copyout, at least for the relevant case where the target task of these operations is the same user task that owns the thread requesting the operation (i.e., the thread that trapped into the kernel and then branched into the OS code). Moreover, it would be wasteful to use one of the schemes based on mapping the user data into the kernel address space, e.g. vm_remap, since on most architectures we already have direct access to the user address space from kernel mode. Therefore we will use vm_read_overwrite and vm_write when the server is in kernel mode (Figure 7) and vm_remap, when in user mode.

It should be emphasized that either method can work in either mode; performance is the only criterion for the choice. See the detailed design discussion for more information.

Since we will use the same mechanism for copyin/copyout that is used in integrated kernels, it is reasonable to conclude that the efficiency of these operations will also approach that of an integrated kernel (ignoring the small overhead of short-circuited RPCs from the server to the Mach kernel).

We have shown that the efficiency of the null system call path will be close to that of an integrated kernel and that the cost of copyin/copyout and the server-to-Mach interface will be similar as well. There are only three other areas where performance could be different from the integrated kernel, since all other code paths are identical to the integrated kernel. These areas are:

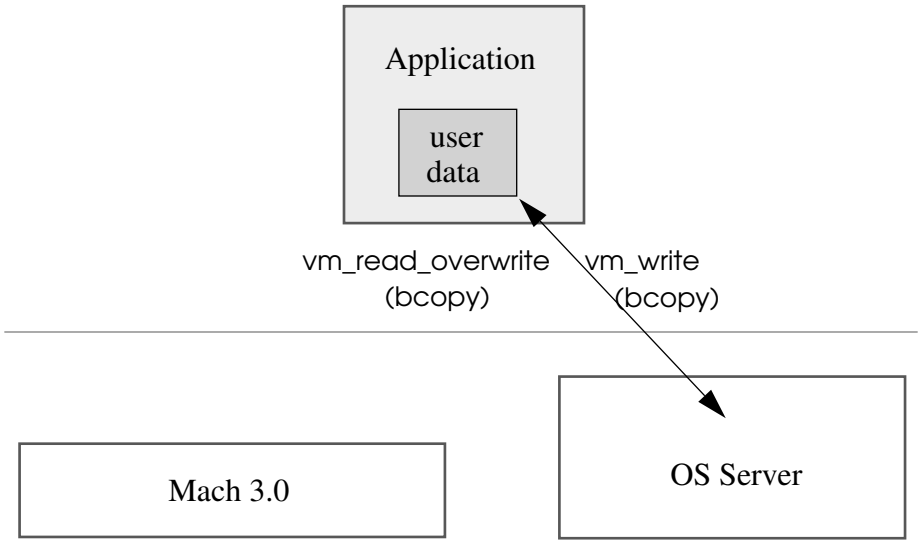

Figure 7: Copyin/copyout with a kernel-loaded OS server.

- The thread synchronization (sleep/wakeup) within the server. Currently this is based on the sending and receiving of null messages, in the OSF/1 servers. This is much worse than the corresponding operation in an integrated kernel, which consists of little more than setting thread state and calling the scheduler function. Although outside the scope of this project, this performance problem should be addressed by adding efficient thread synchronization primitives to Mach. (At this time, the most likely candidates are a modified version of thread_depress_priority/thread_depress_-abort.)
- The implementation of the Mach 3.0 services, where it differs from the Mach 2.5 services found in the OSF/1 integrated kernel. The two Mach implementations have little code in common, so this is a concern. In particular the device interface of Mach 3.0 currently makes an efficient implementation problematic. Improving this situation is also outside the scope of this project, but an in-kernel server will of course benefit from any improvement made here.
- One-way messages from Mach to the server, such as pager messages, network input messages and I/O completion messages. It is not at all obvious how (or whether) to short-circuit one-way messages into something more efficient. Note that a one-way message is not equivalent to a void procedure call, since its semantics require that both sender and recipient be allowed to proceed in parallel. This seems to require that the MIG routine arguments be copied from the sender's stack, so that the recipient can be accessing the copy while the sender goes on to do something else (including, possibly, modifying the arguments).Thus the cost of marshalling arguments into a message buffer apparently cannot be avoided. Neither

can the cost of the thread synchronization, since before the operation there is one runable thread involved, while immediately after the message is sent, there are two. On the other hand, there may be ways around these obstacles, if we optimize for the case where the servicing of the one-way message does not cause the server to do any blocking operations. This is explored further in the Detailed Design section.

After the above items are addressed, it is reasonable to believe that the resulting system will indeed have performance very close to an integrated kernel.

4. Comparison to Previous "Server-Compression" Proposals

Here we compare the current design to previously proposed methods for putting the OS services into the kernel, including (1) doing it at compile-time with ifdefs and (2) doing it at link time with different libraries. We list objections that have been made to each of these methods and explain why the new proposal does not suffer from the same problem. The justification for these claims is contained in the detailed design described in subsequent sections.

Objection 1. The solution is specific to OSF/1 and would have to be re-implemented for each different OS server.

Answer: Unlike previous techniques (for instance, OSF/1 IK vs. MK as a compile-time configuration option), this proposal does not assume anything about the OS services provided and need only be implemented once for all servers. It does requires a certain discipline in the servers in the areas of OOL-memory and port deallocation, but this is a very minor change and is optional.

Objection 2. Only one OS personality at a time can be put in the kernel, so this one has an unfair performance advantage over the others.

Answer: Any number of servers can be loaded, at least if their text/data addresses don't conflict. While we could use PIC (position-independent code) to avoid address space clashes, there is a significant cost incurred using PIC, in both code size and run-time performance. Hence we have chosen to simply link servers at different addresses.

Objection 3. The modularity enforced by having the Mach kernel and the OS services in different address spaces would be lost, allowing accidental violations of modularity to creep into the code.

Answer: The server is compiled and linked separately from the Mach kernel. It cannot access any Mach functions or data by name, nor can Mach access any of its functions or data by name. The only interaction between the two is safely provided by MIG. Bugs involving wild pointers can cause the server to crash the Mach kernel, but servers can still be debugged while running in separate user-mode protection domains, being loaded into the kernel only after the server code has proven to be reliable and trustworthy. Highly secure systems will, of course, mandate complete failure containment, and still require servers to be run in separate address spaces.

Objection 4. The solution requires significant rewriting of all the existing servers; the required resources are too large to justify the gain.

Answer: Only very small changes are necessary to existing servers. Such changes are limited to the deallocation of OOL-memory and ports received in messages, and even this is optional for better performance. It no longer makes much sense for us to severely constrain servers in the interest of better code re-use with the integrated kernel, given that OSF is now adopting the microkernel as the base for its offering.

Objection 5. The solution produces two different configurations of the OS, which greatly increases the resources necessary to maintain it.

Answer: There would only be one compile-time configuration of the server. The same binary runs in either user space or kernel space, simply running fast when loaded into kernel space.

There are still two code paths which need testing and maintenance, but those dual paths are primarily restricted to the MIG glue code - the paths which accomplish most of the work done by the Mach kernel would not be duplicated.

Objection 6. The usefulness of the Mach kernel for real-time applications would be reduced, because just like in the integrated kernel there would be unpredictably long code paths implementing OS services in the kernel, during which the executing thread cannot be pre-empted, even by higher priority threads. In the bare Mach kernel, such code paths are shorter.

Answer: The Mach kernel will arrange that whenever any thread is in dynamically loaded server code, it is preemptible. This can't cause any problems to Mach, since the thread would be preemptible if the code were running in user mode.

Objection 7. We lose the memory-usage advantage of a small non-pageable kernel with a completely pageable OS server running on top of it.

Answer: This design provides a very clear delineation between code that is pageable and code that is non-pageable. Server code that is currently pageable will remain pageable when executing in the kernel's context.

Objection 8. Kernel continuations are not compatible with dynamically loaded OS servers. A thread in the dynamically loaded OS code that invoked a Mach service by direct procedure call would not have its state preserved if the Mach code blocked using continuations.

Answer: The use of continuations is not mandatory. The Mach kernel will know whether a thread executing Mach code has entered it from a kernel-loaded server (rather than by a trap from user mode). If so, it will not use continuations.

Objection 9. There will inevitably be places in the OS server where the optimal way of performing a service is different in kernel mode than in user mode.

Answer: There will be fewer than one might think. Nothing prevents the OS server code from defining a simple boolean function "`in_kernel()`" and using it to make decisions about the best way to carry out a service. Further, use of "`if`" is preferable to "`#ifdef`", since it avoids compiling two different configurations of the server. Note, however, that this will be done only as a last resort, in places where performance is of paramount concern. Correct semantics will be maintained regardless of the method chosen.

5. Detailed Design

In the following two sections we present a detailed design of the two proposed extensions. We believe the design is complete enough that the implementation of it could begin immediately. There are no unsolved problems or other difficulties that we are aware of, although there are still some choices to be made about details of the semantics and alternative implementations in some areas. Of course the design is still subject to improvement and change based on the results of the ongoing review.

5.1 Dynamic Server Loading

Bootstrap Process Review

Since the bootstrap process is different between older (AD1) and newer (AD2) versions of the microkernel, we will review the two procedures in the interest of avoiding confusion between the older, emulator-based design and the newer, no-emulator architecture and establishing the background for the dynamic server loading process.

First, we will review the older, AD1 process. The makeboot program takes a microkernel binary and a default pager binary and concatenates them into a single file with some extra structures inserted describing the default pager size and other things like symbol tables. It prepends a minimal a.out header that describes the whole image as just one big text segment and indicates where the microkernel entry point is. The initial boot loads the image created by the makeboot command into memory and starts it running. The microkernel startup reads the structs added by makeboot and moves them and the default pager out of the way (they start out sitting in the microkernel .bss). After the microkernel is initialized, a task is created and the default pager is mapped into the new task. The master host and device ports are inserted into the ipc space of the new task and the kernel passes their local port names (small integers) as program arguments. The microkernel then starts the default pager task running. The default pager implements the bootstrap service in addition to the paging services and is responsible for loading the primary OS personality. During startup it reads its argv, using atoi() to get the master host and device priv ports. It also gets the root name and boot parameters from its argv. It then uses a Mach version of the ufs code to lookup "/dev/<rootname>/mach_servers/paging_file" (usually a symlink) and uses this as a paging file. It loads the "/dev/<rootname>/mach_servers/startup" file into memory using a Mach version of the OSF/1 execv(2) code and starts the OSF/1 server. The OSF/1 server sends a hand crafted Mach message (msgh_id == 999999) to its bootstrap port, which the default pager responds to with the master host and device ports. In the case of an OSF/1 second server, the bootstrap message is ignored by the first server and the second server does task_by_pid(-1) and task_by_pid(-2) OSF/1 syscalls to get the master host and device ports. The mach_host_priv_self() call used by other user tasks that need the master host port also uses the task_by_pid(-1) interface. The OSF/1 server runs the mach_init program, which implements the "service" service, and it will fork, running /sbin/init in the parent, so that init has a Unix PID of 1, as it expects. The child of the fork uses task_by_pid(getppid()) to get the parent task port and then using ports_lookup/ports_register() will stick the service and other mach server ports into the parent. The netname service (snames) server is run like Unix system daemons and will send a request to the service server to get the netname service receive right.

Second, we will review the newer bootstrap procedure used by the no-emulator system on

which this co-location work will be based. The initial boot loads the microkernel binary and the bootstrap task binary into memory. The bootstrap task will not be loaded into the microkernel .bss and will not need to be moved by the microkernel later. Structures similar, if not identical, to those previously created and added by makeboot will be created by the initial boot. The microkernel creates a task and maps the bootstrap into it. The bootstrap task special port of the bootstrap task is set to a port with the kernel as the receiver. The microkernel then starts the bootstrap task running. The bootstrap task sends requests to its bootstrap port for the master host and device ports and the boot parameters. The bootstrap task implements the bootstrap and service servers and handles exceptions for other servers it starts (unless they decide to handle their own). It creates all the Mach service ports and registers them into its own IPC space. The bootstrap task will create a task and will load and start a stand-alone Mach netname server. The netname server gets the netname service receive right from the "service" server (bootstrap) and will wait for netname requests. The bootstrap task will create a task and will load and start the default pager. The default_pager starts up and waits for pager requests. It will contain no bootstrap or filesystem code. The bootstrap task will ask the default pager for a port to use to add paging space and will send to that port the device port and disk block description for the default paging space. It will then create a task and will load and start "startup" (the OSF/1 server). It will then wait for bootstrap and service requests. The OSF/1 server asks the bootstrap service for the master host and device ports and the boot parameters. The OSF/1 server implements the bootstrap service for its child tasks and is responsible for controlling the bootstrap ports of those children. In the case of a second OSF/1 server, the behavior is the same. The first server can (and probably will) interpose on the master host and device ports. A second server should not be able to tell that it is not the first server. The mach_host_priv_self() call used by other user tasks will use their bootstrap special port to request the master host port. How an OSF/1 server responds is its own policy decision. The OSF/1 server runs /sbin/init. There is no need for mach_init as the service server is already running in the bootstrap task and the netname service is already running as well.

This rearrangement of bootstrap services cleans up the division of responsibilities between the Mach microkernel and OS personalities in several different ways. The older Mach code contains functions that match the behavior of the OSF/1 server. These include ufs filesystem and inode support, directory support, and execv() behavior. The stand-alone Mach environment is being modified during this work to replace these functions with more OS personality neutral implementations. This includes removing dependencies on OSF/1 execv() behavior and the elimination of ufs filesystem code. This results in two classes of Mach programs, namely "Mach" aware and "OSF/1 and Mach" aware. The microkernel, netname server and default pager will only be Mach aware and will not contain any code that is biased towards an underlying OSF/1 system. The initial boot, bootstrap task and OSF/1 server will be aware of non-Mach host machine bootstrap issues, but to varying degrees as needed to perform their defined functions. Program loading performed for all standalone Mach tasks uses the OS personality neutral startup and does not include any support of passing program parameters on the initial stack. All of the stand-alone servers will communicate explicitly through defined Mach interfaces. There is no requirement that any of the Mach stand-alone program images be loadable by any task other than the initial boot or the bootstrap task. Filesystem dependencies are added to the initial boot and the bootstrap task as needed to perform system startup and may not necessarily be the same as those supported by an OS personality. Knowledge of data structures on disks in particular should not be present in the microkernel or the default pager.

Alternatives

The most obvious way to allow code and data to be loaded into the kernel is to expose the kernel task port to privileged user tasks and arrange that all the VM and thread interfaces work as expected when applied to it. Then vm_allocate can be called with the kernel task as its target in order to load the code and data of a server into kernel mode, and task/thread_create/resume can be used to start threads or new tasks in the kernel. This would allow a privileged user task to load any number of servers into the kernel at any time. Note that this user task would have to know how to read binary executable files, but such code is readily available, since it is already part of the Mach bootstrap and the OSF/1 exec system call (the latter is AT&T encumbered code).

Another possibility is to just modify the internal bootstrap code of Mach so that it can be supplied with a flag telling it whether to load the initial server task into kernel mode or user mode. This would only allow one server to be loaded into the kernel, and only at bootstrap. Of course this server could implement a service that loads other servers into the kernel, but such a service wouldn't work when the server is running in user mode so there would be a difference in functionality between the two modes, which we are trying to avoid.

Finally, a new Mach service for kernel loading could be invented, but it is hard to see the point, when the appropriate functionality can be obtained more simply with existing services, using the kernel task port.

Design

We have chosen a variant of the first alternative, namely to expose the kernel task port and load servers into kernel mode with a privileged user task, since this is more flexible than the second alternative. Moreover, this user task, with only minimal modification, can also be used to dynamically load one server into an already running user-mode server task, improving the performance of multi-server systems, even when the multiple servers are run in a separate address space from the kernel.

In the normal, in-kernel operation of a single-server system, this loader task will be the bootstrap task initiated by Mach (i.e. the separately linked bootstrap code that, in recent versions of the OSF/1 MK kernel, is used to load /mach_servers/startup). It will by default load the OS server found in /mach_servers/startup. The bootstrap task will obtain the kernel task port to the main kernel task via the usual bootstrap mechanism. It will then read in the header of the executable file containing the code and data of the soon-to-be loaded server, to determine what address the server has been linked and relocated to, along with the size of text and data needed by the server. The bootstrap program will use a new Mach service, kernel_task_create(), to create and initialize a separate kernel task, with an address (sub)map large enough to hold the text, data, stacks and all dynamically allocated memory the server will ever require, including thread stacks. Since the proper size for the submap depends on system configuration and intended loading, the bootstrap program will use a default (probably in the area of 16 Mb, given current experience) and allow the operator to override the default with a boot-time option, for complete control.

The call to kernel_task_create(), if successful, will create a new kernel task, separate from the main kernel task, whose address space is obtained from the previously mentioned sub-map, returning the task port for this new kernel task. The new task is created sans threads, and with no real data instantiated into its address space, except that the main kernel_map space will be accessible to this new task. The bootstrap program will vm_allocate() areas of appropriate address and

size to hold the text and data regions of the server, using vm_write() to install the bits into the server task. Vm_protect() will be used to render the text area read-only. As an aside, it's worthwhile to note a potential drawback to this scheme. These newly allocated portions of pageable address space will be anonymous memory, backed by the default pager, regardless of where their bits came from. Ideally, we would like to avoid wasting paging space for the text that gets paged out, as the wasted bandwidth and paging space from thousands of nodes paging identical copies of read-only text onto default paging partitions is enormous, especially given that the bits could just as easily be retrieved from the original source. It is likely that use of vm_map() rather than vm_allocate/vm_write() would be the right approach to solve this problem on all non-boot nodes; however, since this problem exists today, and is orthogonal to server co-location, we will not address it during this project.

The bootstrap task will then perform any further setup need to allow the new task to function as an in-kernel server. This will include insertion of some well-known ports (only the bootstrap port should be needed) into its IPC space, setup of an initial kernel thread, to begin executing at the main entry point of the server. At that point the bootstrap task is finished and will exit. Privileged users (i.e., developers) can run the bootstrap task again, with appropriate arguments to load specified server programs into the kernel. In a production environment, these additional servers would presumably be loaded via explicit command from system startup scripts.

If for some reason it is desired to run the OS server in user mode (e.g. for improved debugging), a flag can be specified to the "boot:" prompt. This flag will be passed to the bootstrap task, causing it to load the OS server as a user task in the same way it does today.

In more detail, this bootstrap loader task would be invoked with following argument syntax:

```
bootstrap [ -u | [ -s map_size ] ] [ server-pathname ]
```

With no "-u" flag, it would load the specified server into kernel mode. If the server pathname is omitted, it defaults to /mach_servers/startup.

This enhanced bootstrap loader task will use a set of well defined interfaces to object format dependent routines to parse the server object file. These routines will, in turn, be built upon a set of file system dependent routines to read in the object file using Mach services. This object-oriented approach should facilitate porting the bootstrap loader to different object file formats and file system types. Encumbered code used in porting to a particular architecture could also be easily isolated and removed from a distribution this way.

Though ideally we would have all of the fully linked server binaries consist of position independent code (PIC), that exacts a penalty in both code size and run time, is not necessary, and won't be done. If the relocation information is not discarded, but is kept in the server binaries, it is quite simple to run the loader to relocate server images into separate parts of fixed address ranges. However lazy, it is sufficient to pre-allocate fixed-size, large chunks of address space for each server. This simplifies dealing with system ports whose software must statically size the microkernel's pmap. The bootstrap loader will use the kernel_task port to create a new kernel task, having its own pageable address space, constructed as a submap of the primary kernel map. Since we need to control both the base address and size of this submap, a new Mach interface will be provided. This separate submap both allows the load server to be fully pageable and prevents fragmentation of the server's space from slowing vm operations done by the main kernel_task, while use of a separate kernel task allows us to unload (and re-load) servers without rebooting the machine (NT allows this).

The bootstrap loader can then allocate space in the newly created kernel task, write the server text and data into that area, protect the newly loaded code and data appropriately, and finally set a single kernel-mode thread running at the address of the entry point. Providing arguments (e.g.: boot-time configuration information) is also quite straightforward, though it isn't clear that future releases will support this Unix-flavored notion in pieces of software that are only Mach-aware (ie, lacking knowledge of OSF/1 interfaces).

So, once the kernel_task port has been exposed (via the usual bootstrapping mechanism), it is simple to use the usual Mach interfaces to effect the loading of some number of servers into the kernel's address space, starting one or more threads running in this newly loaded code. This allows us to easily support co-locating multiple user-mode servers (and short-circuiting RPC's between them) to enhance the performance of multiple servers without giving up the separate enforced protection domain.

In addition to a new interface for the creation of kernel-mode tasks, minor enhancements to the Mach kernel will probably be necessary to enable all of this to work. Some code paths that have not been adequately exercised will probably be stretched sufficiently to require repair, and some Mach services may not be sufficiently general, if there are any assumptions of having been called against a user-mode task. In particular, supervisor mode faults on pageable memory, both text and data, when being accessed by a kernel thread must be handled correctly, if they aren't already. The Mach kernel must also know how to handle trap instructions that originate (and thus return to) supervisor mode, since the loaded server may issue traps while running in the kernel.

Some functional enhancements will be required of the kernel. The new interface to create kernel tasks mentioned above must be added. Threads running in a dynamically loaded server must be pre-emptible, as they were when running in user mode, to avoid increasing latencies. The kernel will have to know which threads are immigrants into the server, and are targets for pre-emption. These threads must also avoid using continuations while running in the kernel, performing Mach services on behalf of the server, since necessary thread state would be lost. These threads trapping into the kernel and being vectored off through the RPC short-circuiting mechanisms (elaborated upon below) will need only an extra few bits of state to indicate that the thread can be pre-empted, that use of continuations should be avoided, that rescheduling or termination is required at the nearest convenient clean point, and that certain operations should be deferred or restricted. The kernel will also keep track of the stack of tasks a thread has migrated into or through; this is needed both to support "de-migration" as the calling sequence unwinds, but also to ensure that operations performed against mach_task_self() affect the correct task. In other words, when a client thread has migrated into a server, whose code tries to allocate memory, that memory must wind up in the task of the server, not the client

5.2 Short-Circuiting RPC's.

5.2.1 Thread Model Issues

In the semantics of procedure calls (remote or local) there are two essentially different models of the flow-of-control that are possible, with different ramifications for correctness, robustness and performance.

The first model says that the thread invoking the procedure call (the client thread) does so by branching to the code of the procedure. No server thread directly executes the code of the proce-

dure, although auxilliary server threads may exist to assist or control the client thread in its execution. We call this the "thread-migration" model, because the client thread temporarily emmigrates from the client module to the server module.

In the second model, the client thread never enters the server at all. Rather a separate thread in the server is somehow activated by the client's invocation and this server thread executes the code of the procedure. The client thread passively waits until the server thread finishes the procedure call, at which time it is reactivated and can continue processing the instructions after the procedure call. This is known as the "thread-switching" model.

In general, thread migration is more efficient, because it avoids activation and deactivation of separate threads, but thread switching gives more functionality, because it is possible to distinguish between the waiting client thread and the active server thread, during the call. This allows the suspension or termination of either the client thread or the server thread, without adversely affecting the other.

Consider the case of Mach's `thread_abort` function, which is defined to immediately terminate a blocking Mach service that is in progress by the specified thread, including mach_msg. In the thread-migration model, one has no way to specify the termination of an RPC, because the client thread is not blocked waiting for the server. Instead, the client thread has moved into the server and using thread_abort on it may cause it to corrupt the operation of the server. (It may be in a blocking operation in the server code that is not safe to interrupt.) Clearly there are security and robustness problems with allowing other threads of the client task to apply thread operations in an unrestricted manner to a thread that has migrated into a server.

But in the thread-switching model, one can have thread_abort immediately resume the waiting client thread, while the server thread is still executing the RPC. This raises the issue of whether or not the server should be notified that it is useless to continue processing the RPC (because the client is no longer awaiting a response), but that is mostly just a performance issue, and moreover, happens only in an exceptional case. Besides, it is not hard to arrange that the server is notified, if that is desired. Some would argue that thread_abort should never be used in isolation to abort an RPC. Instead, the server should be sent a separate RPC asking it to abort the first one, at which time it can choose to apply thread_abort to the client thread if it wishes. In any case, the important safety issue is that the thread-switching model does not give the client task any direct ability to affect threads that are in the server — this is entirely up to the server.

It is easy to see that, in their implementation, message-based RPC uses thread switching, and short-circuited RPC (both user-to-kernel and within the same address space) uses thread-migration. Because of the problems of the thread-migration model above, we do not want it to be visible outside of the RPC/Mach layer, but we do want to use it to implement short-circuited RPC, because of its inherent efficiency advantage. Here is how we can achieve both goals.

The implementation of the normal short-circuited RPC path (when no thread operations are performed on either the client or server threads) will be pure thread-migration. The client thread will branch to the code of the server (after a trap to kernel-mode, if necessary) and will directly execute the code of the server procedure. However, whenever any thread or task operation is attempted that would enable any code outside of the Mach kernel to discover that this migration has taken place, Mach will do one of two things to make it appear that thread-switching was used instead:

- If feasible, the operation will be delayed until the RPC is complete. This

will work for thread_assign, thread_priority, thread_suspend, thread_terminate, thread_set_state and others, when applied to the client thread.

- If legal, the operation will be carried out immediately. This will be possible for thread_resume applied to a client thread that is not suspended (a no-op) and for thread_get_state (the state returned will make it appear as if the client thread is waiting at the point of invocation of the RPC) and other informational calls.
- Otherwise, Mach will temporarily split the migrated thread into two threads: an inactive client thread waiting at the point of invocation of the RPC and a new server thread that is executing the server code. This will be necessary for thread_abort applied to the client thread, and for mach_thread_self executed by the client thread that has migrated into the server (it must not see the same answer as it got when it was in the client).

The only case where the above methods will not work is short-circuited RPC within a user task. The Mach kernel is unaware when this type of RPC takes place, since no Mach services are used (and this is necessary to its efficiency). Therefore it cannot know when thread operations are being attempted on a client thread that has migrated into a server. For now, we do not intend to do anything about this defect, except to document it as a restriction. Again, if one believes that clients should not attempt thread operations on threads engaged in RPC's except by notifying (and obtaining permission from) the server, then the restriction is unimportant to the legal activities of clients. The only concern is the case of a user-mode, co-located client and server, where the server calls mach_thread_self or tries to do operations on client threads that are engaged in RPC's to the server. Such servers may exist, but we don't know of any.

5.2.2 Server Port Registration and Client Lookup

The problem to solve here is how the server can announce its presence to clients in the same address space and provide them with the information necessary to call the correct server procedure for each routine in the MIG subsystem. This will then also help solve the problem of how a user mode client can short-circuit an RPC to a server in the kernel: after the client traps into kernel mode, it is in the same address space as the server, and the same technique can then be used.

We are not addressing in this section how the arguments of the RPC are transmitted between client and server. (See *Ports and OOL Memory Parameters* and *Short-Circuiting User-to-Kernel RPC's*, below for that part of the design.)

Also, since we want to allow multiple servers for the same MIG subsystem in a single address space, it is clear that the short-circuiting information has to be specified per port, not per MIG routine. That is, the client cannot simply use the MIG subsystem and message number as inputs to the problem of finding the right server procedure; the answer may also depend on which port is the target of the RPC. So we need to design a method that allows servers to give out this information separately for each port on which they are receiving. It must allow clients that hold a send right to obtain the short-circuiting information about the port, when they are in the same address space with the server, and to discover that they are not, otherwise.

Alternatives

Since a server and client in the same address space always have the same name for the send

right and receive right of a port, a simple way to do the per-port registration and lookup is to have both the server and client map the port name into a pointer to the per-port short-circuiting information. This mapping can be done once and for all by the server, if it renames the receive right at the time of creation to be a pointer to the short-circuting data that it registers. Then the client can look up this information simply by dereferencing the pointer. This method of mapping would make it illegal for servers to do renaming for other purposes. However, this is not a serious restriction on the freedom of such servers if we leave at least one "user" field free in the short-circuiting data structure, so that it can be used to hold whatever renamed value the server wanted. Any existing server that renames its ports can easily be changed to store the desired name in this field, instead of actually renaming the port. However, if this restriction on servers is considered too severe, the short-circuiting data could be accessed by looking up the port name in a hash table instead of depending on renaming. This will be somewhat slower of course.

A problem with this approach is how to enable a client to detect that the port name it holds is not registered (i.e. how does the lookup return "not found") in the case where it expects ports to be renamed by the server to a valid pointer, but this renaming has not been done because the server is not in the same address space? And if we are using hashing instead of port renaming, how can servers and clients that were separately linked and dynamically loaded together find this global hash table in order to use it? This is discussed in more detail in the design, below.

As another approach, the server could register some or all the information about each port it listens to with the Mach kernel. The minimum it would have to register is a pointer to the location of the rest of the information. Then the MIG client code, which holds the send right to the same port, could ask the Mach kernel for this information. To avoid excessive traps into the kernel, it would cache the information in its own address space, so that it only has to ask once. If no server has registered the port, or the server that has registered is in a different address space, the kernel would return an error and the MIG client stub would know that a slower form of RPC must be used. Note that when server and client are both in the kernel, a trap into the kernel is not required to register this information or to look it up, assuming that both client and server know how to directly access the kernel data (a port name in the kernel is a pointer to Mach's per-port data structure, which is where we would store the short-circuiting data).

The fatal problem with this alternative is that the client stub must cache information about each port (if it is in user mode), to avoid excessive traps into the Mach kernel. This requires it to be able to map from a port name to the cached information about the port. Well if it can do that, so can the server that is in the same user task with it (the have the same port name), so why not have the server register the data locally in the user task? The conclusion is that this method has no advantage over the first alternative, and is slower because it requires traps into Mach for registration and the first lookup.

A third possibilility is that the MIG client code could unconditionally use the slower, mach_msg implementation of RPC on the first RPC to each port. When the server finishes the RPC it would send its own task port in the reply message, along with the address where it has registered the short-circuiting information about the target port. The client would then compare the returned task port with mach_task_self, to see if it is in the same task. If so, it would cache this information and use short-circuited RPC's for that port on subsequent calls. Unfortunately, this alternative suffers from the same problem: if per port information has to be cached by the client stub, for efficiency, the server might just as well cache the information for the client to look it up, so this method also has no advantage over the first alternative.

Both of these last two alternatives suffer from the problem that they do not respond well in the face of migrating receive rights. After the first RPC to a port, the client assumes that the server remains stationary, and will never change the method that it uses to do the RPC. But servers can pass receive rights among one another, and a port that used to be targeted at a remote server can later be targeted at a server in the same address space. The first alternative, since it does a full port lookup on every RPC, will notice immediately if a receive right migrates into the same address space. When this happens, the server that obtained the receive right will immediately register the port and subsequent lookups by clients in that task will find it.

Design

For the reasons stated above, we have chosen the first alternative. We also choose to use port renaming to map the port name into a pointer to the short-circuiting data, because it is faster than hashing, and we see no problem modifying existing OSF servers that rename their ports. In most of the existing cases, changing two or three lines of code in the server, for each kind of port that it renames, is all that is required, and this will not result in any noticeable performance loss, since it only adds one pointer dereference to each use of the server's desired name.

Another reason to use port renaming is that this will make the access to the short-circuiting data be the same from kernel mode or user mode, as will be seen below.

There are three new data structures involved in port registration and lookup: the **RPC subsystem** data structure, the **RPC port** data structure and, for user-mode tasks only, an array of pointers to the RPC port data structures known as the **RPC port vector**. The reason these are being called RPC data structures rather than "short-circuiting" data structures is that they have broad use outside of the specific task of short-circuiting message-based RPCs into a procedure call. A previous note by Faramarz Rabii and Michael Condict detailed several other uses for this data. It was proposed that a general-purpose RPC library be created for the purpose of providing functions to register and look up ports in this data structure, and perhaps also provide generic server functionality, such as that provided by mach_msg_server and mach_msg_destroy (i.e. perhaps these functions belong in the proposed RPC library). A complete copy of this proposal appears as *Appendix A* and *Appendix B* in this document.

Block diagrams of the RPC data structures and their connectivity are shown below for the case of a user-mode server (Figure 8) and a kernel-mode server (Figure 9). The only difference is that, in kernel mode, Mach ipc_port structure takes the place of the RPC port vector, so that structure is not needed. In both cases, servers and client can assume that any registered port is a pointer to a pointer to an RPC port structure (in the kernel case, the RPC port structure pointer is the first field of the ipc_port structure), that the RPC port structure contains a pointer to an RPC subsystem structure, and that the subsystem structure contains a jump table indexable by message number which points to the server functions that implement that subsystem. Obviously, the subsystem structures will be shared by all the ports belonging to that subsystem, to save space.

It remains to describe how the client can know that a port is not registered, given that the lookup scheme involves dereferencing a port name that is interpreted as a pointer. This is not a safe operation on a send right that has not been renamed (because no server registered it in the current task). Note that this is not a problem for servers running in the main kernel task. There, every port name is a pointer to an ipc_port data structure, and thus can safely be dereferenced.

In user tasks, the client code will do a range check on the port name, to ensure that it points

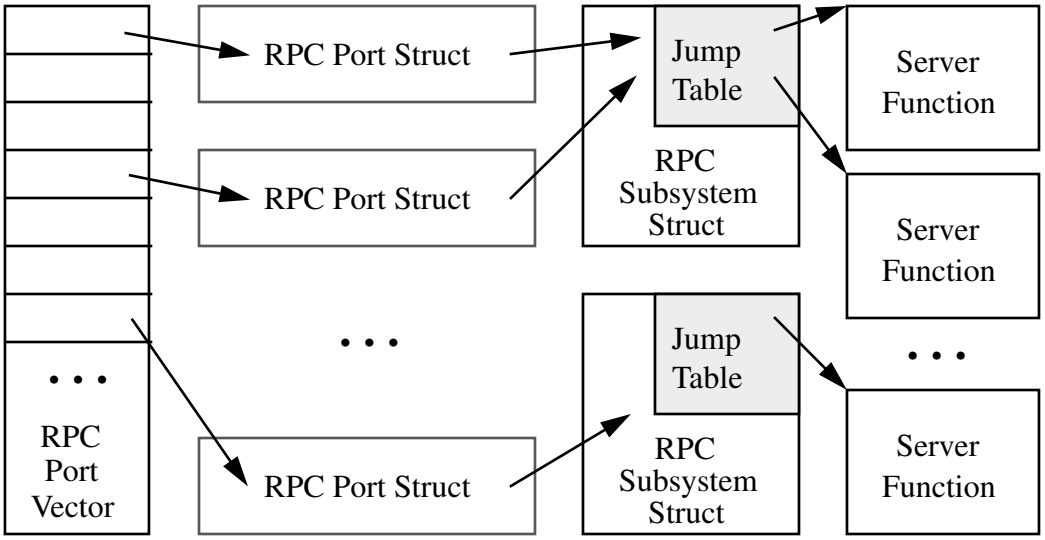

Figure 8: User-mode RPC Data Structures

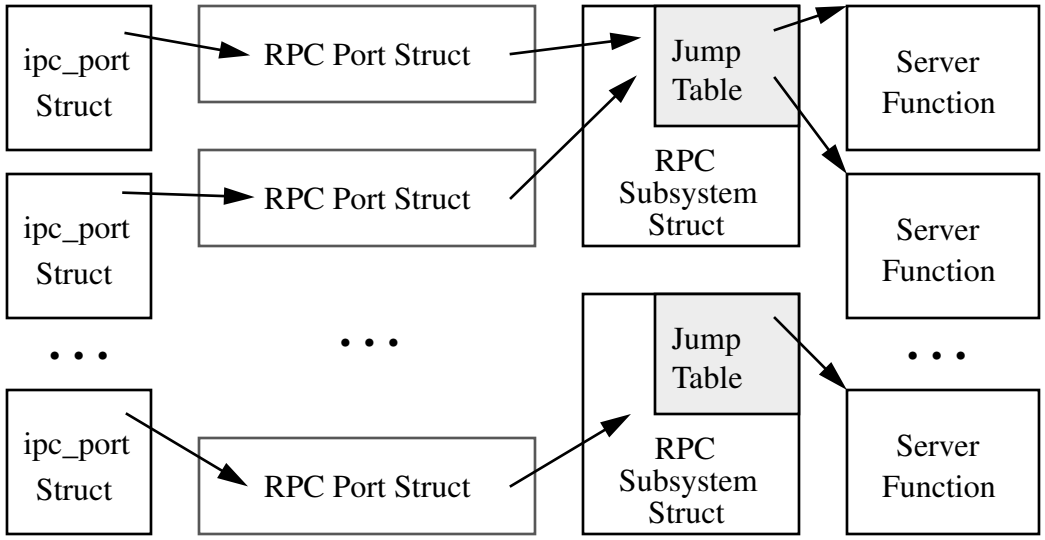

Figure 9: Kernel-mode RPC Data Structures

into the port vector. This will add a few instructions to the short-circuited RPC, but seems unavoidable. Since we want to use the same MIG-generated client code in the kernel, we will do the range check there as well. But the range will be set to the entire kernel address space, so that the check always succeeds.

This exposes another problem: how do multiple servers and clients loaded into the same address space obtain shared access to the RPC port vector (in the kernel, this is a phantom port vector that appears to span the entire kernel address space, but this phantom address range still has to be specified to any dynamically loaded MIG client code). The port vector represents the only shared variable between the separately compiled and linked servers. Therefore, it does not seem

necessary or advisable to devise an elaborate scheme for dynamically linking references to shared data. Since we want to allow the loading of stripped servers, we cannot assume that the dynamic loading process fills in the value of some specified symbol (say RPC_port_vector). This pointer will have to be set up at run time using the program arguments (argv), the environment (envp) or a new Mach service.This address is filled in with a pointer to a descriptor of the port vector, consisting of its size in bytes, together with a pointer to the port vector itself. (In the kernel, the vector would not actually exist, but that's okay, since any actual pointer to an ipc_port structure looks like it is in-range, and can safely be derefernced.) The first server in any user task, which is loaded by the OS server itself (using exec) could of course have this pointer defined at link time in the conventional fasion, or could allocate the port vector dynamically and set up the descriptor at that time.

Another problem is how a server can deregister a port, when it wants to migrate the receive right to a different server, or destroy the object (but in the latter case, it should probably have arranged to get rid of all outstanding send rights first). This is complicated by the fact that migrating the receive right to a different server may leave clients holding a send right in the current task. This send right has been renamed to point into the port vector, but it no longer points at a valid entry, since the receive right is now in another task. To indicate this, we will set the pointer to the RPC port structure to null in the port vector. Thus a second level of validity testing is needed when the client does its lookup.

Finally, some other subsystem information is needed besides the jump table. This includes a pointer to a threads-package immigration function, to allow a foreign thread to enter the server without corrupting its threads-package data structures (see Threads Library Changes, below). It also includes a description of the number and type of arguments for each routine in the subsystem, so that the Mach kernel can implement argument transmission in user-to-kernel RPC's (see User to Kernel RPC's, below).

5.2.3 Sequentialization and Throttling of RPCs

Several existing Mach-based servers need to be able to ensure that certain RPC's directed to a particular port are processed by the server in the same order they arrived. The most obvious example is that a "read" RPC on a port that represents an object with state must wait for all preceding "write" RPC's to complete, or it will return the wrong results.

Also, some servers may wish to limit the total number of simultaneous RPC's that can be in progress at once. A per-port limit, per-subsystem limit, or per-server limit may be desired. We call this "throttling".

At first glance, it might appear that the short-circuiting scheme proposed here has to take into account these needs and solve them in a manner that is integrated with the solution of the same problems for message-based RPC. At least for the case of sequentialization of messages, this is not true. No out-of-order message processing can happen with RPC's, in which by definition the client does not proceed until the server finishes the processing of the RPC. And even with multiple threads doing RPC's to the same port in parallel, there is no way for two threads to interact with one another in such a way as to prove that the server has executed their RPC's out of order. This is because neither thread can initiate an RPC, and then inform the other thread that it has done so, before the RPC completes. (And a third thread looking at either of the two cannot tell when and whether their RPC has reached the server.)

So we conclude that we do not need to worry at all about sequentialization of short-circuited RPC's. However, if and when we arrive at a design for short-circuiting one-way messages, it will have to take into account this need.

The case is not so clear for throttling of messages. While it is clear that one-way messages allow the throttling problem to be much more severe, it is not clear whether the throttling problem is eliminated by requiring clients to wait for an RPC to complete. Some servers may not be able to efficiently handle simultaneous RPCs from the largest number of client threads that can exist. For now, however, we consider this problem of low priority, since it has not been addressed in any current server. Our design leaves open the possibility of implementing throttling by having client threads block in the MIG client stubs, to be resumed automatically by other client threads finishing their RPC's. Other than keeping this possibility open, we will not address the design of throttling any further, for now.

5.2.4 Short-Circuiting User-to-Kernel RPCs

There are two issues to be worked out here. The first is how a user-mode client can know that the server is in the local kernel, so that it can replace the use of RPC with a more efficient form. The second issue is what the more efficient form might be.

We said above that servers will register in a per-task table, and that clients can look in this table to see if the server is local. This will not work in the case of a server in the kernel, because the client cannot access the kernel's memory.

One solution, which is how several Mach services work today (e.g. vm_allocate), is to always assume that the server is in the kernel, if it cannot be found by the look-up in the user-mode table. This is okay, as long as the client is prepared to fall back to a full RPC if the request is rejected by the kernel whenever the server is not in the kernel. The performance impact of a failed attempt is not a concern because the cost of this will be much less than the cost of message-based RPC.

So there would be three possible levels in the efficiency of request processing by user-mode clients (see :

- If the server is in the same user task, a direct procedure call is used.
- If not, a direct kernel request is made for the service (using the new user-to-kernel short circuiting Mach interface).
- If that is rejected with the appropriate error, a fully general RPC is initiated, using mach_msg.

What form should a direct kernel request take? First note that it has to work for dynamically loaded services as well as Mach services, so it cannot be based on compile-time assignment of numeric trap operands.

On the other hand, there has to be some way that the different requests are distinguished by the kernel. The obvious way to identifiy a MIG-based request is by the message number that was assigned to it by MIG.

One solution is to use the same trap instruction with a fixed numeric operand in all short-circuited kernel calls (i.e. define a new, trap-based Mach system call). The MIG client stub would call the function containing this trap instruction with the MIG message number as its first argument. The other arguments would be the target port and the remaining MIG arguments. Upon entry to the kernel, a special trap handler could find the message number and target port on the

user stack, and use this to determine whether the server for the request is in the kernel. If so, it could transfer control directly to the correct server procedure using the server-provided jump table.

(Note that Franco Travos (travos@osf.org) and Franklin have already talked about a plan to invoke all kernel services via traps, avoiding the message overhead. This proposal avoids the messages in a different way, but both proposals have in common that they require the implementations of the Mach services to be reorganized so that they do not assume throughout that they were accessed via a message.)

The RPC short-circuiting proposed here, when applied to user-mode requests for Mach services, has several advantages over converting all message-based services into hard-coded trap numbers in the Mach library (as others have proposed):

- If an OS server that uses traps to request Mach services is loaded into kernel mode, it will continue to use traps, which is not as good as direct procedure calls. Using our method, the OS server will automatically short-circuit to direct procedure calls after discovering it is in the same address space with the Mach servers.
- It avoids the name-space pollution problem (number-space, actually) associated with assigning trap numbers to services. Which trap numbers are guaranteed not to conflict with any OS that we may wish to layer on top of Mach? We avoid this by collapsing all trap-based access to kernel services into one new trap number, so we minimize conflict with existing OS trap numbers.
- When new traps are added for existing Mach services, programs compiled to take advantage of the new traps will not run on older Mach kernels that don't support those traps. We don't have this problem because the information about which address to branch to for each Mach service is obtained at run-time from the kernel.
- Finally, our method is simpler because it uses much of the same code to speed up user-to-user RPC's as it does for user-to-kernel RPC's.

Having described the code path from a user to a kernel-loaded server, we now discuss the transmission of arguments between user and server. This is not a simple matter of copying them, since there are machine-dependent precautions and set-up that needs to be done when kernel-mode code accesses user data. Moreover, for MIG arguments that are ports, someone has to translate the user's port name to a kernel port name (the latter is required to be a pointer to the kernel's ipc_port struct). Note that this work will not be done immediately by this project, since it is not needed by the emulator-free server, where the user process traps into the kernel with an ordinary Unix system call trap, causing the kernel to generate a syscall exception message to the kernel-loaded server (i.e., a kernel-to-kernel RPC). Only emulator-based servers have a need for this user-to-kernel short-circuiting, described here for completeness.

Alternatives

We must first decide which entity is responsible for copying the arguments in and out. We might allow the client to "push" the arguments, but that would continue to allow the current behavior, where an errant or malicious client can kill a server by sending an arbitrary amount of

trash in its request. We might also allow the server (-side stub) to be responsible, since it could do the argument transmission using existing Mach services such as mach_port_extract_right, vm_read_overwrite, vm_write. The problem with this approach is that the server would of necessity need to be given the requesting client's task port, a potential security risk, and in addition would need to call back into Mach multiple times to handle the multiple parameters. Hence the approach which seems to be the cleanest and most efficient is to require the Mach kernel to process the arguments. Since the server will need to register its subsystem (ports, msh_id range, worker functions) in any case, it is only a small extension to require it to fully describe the arguments to those functions. The kernel can then efficiently prepare the arguments for the its own or the server's use, avoid any possibility of subterfuge by a malicious client, and still allow MIG to own the description of the arguments for each short-circuited RPC.

Design

When MIG compiles the client and server-side stubs for a given subsystem, it also builds a table describing the arguments for each message (msgh_id) supported by that subsystem. The entries in this table are organized as string of small descriptors, to allow the description of an arbitrary number of parameters with various properties. Each argument may be in, out, in/out, a port, in-line, or out-of-line (OOL). Since MIG logically owns the description of the arguments though the kernel must perform the preparation and transmission of the arguments, we will avoid intertwining the knowledge of the argument descriptor format by using macros. MIG will generate the tables describing the arguments for each short-circuited message, and also generate macros used to parse these descriptors. The kernel will simply use these macros to iterate over the arguments as it prepares them for server use (or for return to the client); with this separation of responsibilities, MIG can easily alter the format of the descriptors without requiring matching changes to any kernel source, merely re-compilation with newer MIG-supplied macros.

The following pseudo-code outlines the algorithm the kernel follows to handle the client's arguments in the process of carrying out a user-to-kernel RPC.

```
copyin msg_number and port_name from the client;
ipc_port_t p = translate(port_name);
subsys_pointer subsys = kernel_RPC_lookup(p, msg_number;
copyin(client, &user_args[0], RPC_ARG_COUNT(subsys->arg_desc[msg_id]));

FOREACH_RPC_ARG(user_args, subsys->arg_desc[msg_id], &arg, &argflag,
                                                        &argsize) {
        switch(argflag) {
                case ARRAY | IN :
                case ARRAY | IN | OUT:
                        if (argsize >= PAGE_SIZE) {
                                map it into kernel address space;
                                change local arg to kernel pointer;
                        } else {
                                copy it into a kernel local variable;
                        }
                break;

                case PORT | IN:
                case PORT | IN | OUT:
```

```
                        local arg = translate(arg); /* takes a ref */
                break;

                case SIMPLE | IN:
                case SIMPLE | IN | OUT:
                default:
                        /* Nothing to do, use copied in arg as is */
                        ; /* break; */
        }
}

/* Migrate into the server to do the RPC: */
IMMIGRATE(current_thread(), subsys->func[msg_id], p, local_args, n_args);

FOREACH_RPC_ARG(user_args, subsys->arg_desc[msg_id], &arg, &argflag,
                                                        &argsize) {
        switch(argflag) {
                case ARRAY | OUT:
                case ARRAY | OUT | IN:
                        if (size >= PAGE_SIZE) {
                                free up kernel mapping;
                        } else if (argflag & OUT) {
                                copy out from local kernel variable;
                        }
                break;

                case PORT | OUT:
                        copy out the port;
                break;
                case PORT | OUT | IN:
                        copy out the port;
                        /* fall through */
                case PORT | IN:
                        deallocate the reference taken above;
                break;

                case SIMPLE | OUT | IN:
                case SIMPLE | OUT:
                        copy out the arg;
                break;

                default:
                        /* nothing to do */
                ; /* break; */
        }
}
```

The IMMIGRATE macro, explained in more detail elsewhere, does very little in the case of user-to-kernel RPC:

- sets flags in the thread structure indicating it's a migrant
- calls the function with local args
- clears flags in preparation for returning to user mode

In the case of user-to-kernel-loaded-server RPCs it does more:

```
        - sets flags indicating migrant status, pre-emptibility
        - indicates that thread is in new, not home, task
        - switches to an empty server stack
        - calls the function with local args
        - switches back from server stack
        - cleans up notion of current task
        - clears flags in preparation for return to user mode.
```

5.2.5 MIG Client-Side Stub Changes

The necessary changes to the client stub function generated by MIG are pretty well defined by the data structures chosen in the previous sections. We describe them algorithmically here, for clarity.

Currently, MIG generates a client stub function that has roughly the same arguments as specified in the MIG routine declaration (but expressed in C instead of the MIG language). It copies the in-line arguments into a message buffer, sets up the message header to deal properly with the out-of-line memory and port parameters and then calls mach_msg to do the RPC.

We will keep the same interface to the client stub function, and the mach_msg-based code just described will remain unchanged, but it will be only be used if both types of short-circuited RPC's are not possible. An outline of the algorithm for a MIG stub "xyz" follows:

```
mig_routine_t xyz(target_port, other args . . .) {
if (target_port in range of port vector) {
        spin_lock(target_port->lock);
        if (target_port->port_info == NULL) {
                /* This port has been deleted, so: */
                spin_unlock(target_port->lock);
                goto try_kern;
        }
        /* Server is in my address space, so use procedure call: */
        if (this stub's message_id in range for port's subsystem) {
                svr_fn = RPC_sc_fn_lookup(*target_port, message_id);
                set up OOL memory and port args . . .
                spin_unlock(target_port->lock);
                IMMIGRATE(svr_fn, target_port);
                spin_lock(target_port->lock);
                tear down OOL memory and port args . . .
                spin_unlock(target_port->lock);
        } else{
                spin_unlock(target_port->lock);
                return INVALID_PORT_FOR_THIS_ROUTINE;
        }
} else {
        try_kern:
        /* The server might be in kernel, so try the short-circuiting
         * RPC into the kernel:
         */
        if ((err = kernel_RPC(msg_id, target_port, other args . . .)) !=
                                MACH_RPC_NO_SERVER)
                return err;
}
```

```
/* Server is not in kernel, fall back to mach_msg: */
marshall the in parameters . . .
set up message header for OOL and ports . . .
mach_msg( . . . );
unmarshall the out parameters . . .
```

The IMMIGRATE line, above, is a call to a programmer-supplied macro or function that will cause the client thread to do whatever is necessary to immigrate into the threads world of the server and then branch to the specified server function with the same arguments as were supplied to the MIG client stub. The definition of this Macro is both compiler/machine dependent and threads-package dependent. Moreover, the Mach kernel may have special requirements for immigration that need to be satisfied by a kernel-loaded server that wants to do short-circuited RPC into Mach. Therefore, it will often be necessary for the programmer of the server to supply the definition of this macro at the time the client stubs are compiled. MIG will have default versions for each machine-type supported, which will simply use the immigration function supplied by the server with the subsystem information (see Server Port Registration and Client Lookup, above, and Threads Library Changes, below).

5.2.6 Ports and OOL Memory Parameters

Care must be taken to preserve the semantics of passing port rights and OOL memory in RPC's, when they are short-circuited to a direct procedure call. Clearly, when a send right is passed as a parameter, the client-side code must allocate a new send right, and must allocate a fresh copy of any OOL memory, passing the copy instead of the original memory. (No action need be taken when a receive right or send-once right is passed.) Otherwise, when the server is finished with the port or data, it will deallocate it, wreaking havoc with the client.

In order to avoid unnecessary and costly allocation and deallocation of port rights and OOL memory in short-circuited RPC's, we add two new attributes to MIG parameters, which can only be used with "in" parameters. The first is "read-only", with the obvious meaning that the user code on the server side is agreeing not to modify the parameter. The second is "temporary", which declares that the server will not access the parameter after the return of the RPC. The MIG code in the server-side stub will automatically deallocate "temporary" port or OOL parameters at the end of the RPC (using `vm_deallocate` or `mach_port_deallocate`, as appropriate). The user-written server code is not expected or permitted to do the allocation of temporary parameters itself.

Now observe how we can implement the passing of port rights and OOL memory that are declared "read-only, temporary" in the case where we are short- circuiting the RPC and doing a direct call to the server procedure. Instead of vm_copying the OOL memory or making a new send right, we simply pass the address of the memory or the name of the port right to the server procedure. Because it is in the same address space (and port name space) it can access the memory or port directly. And because the argument was declared read-only, we don't need to worry about the server modifying the memory, which would be visible to the client and would be a violation of RPC semantics. Then note that since we called the server procedure directly from the client-side stub, we automatically bypass the code in the server-side stub that would have deallocated the memory or port right, which is exactly what we need to do in the short-circuited case.

In order to make these optimizations in an existing server, all that is necessary is to add the temporary and read-only attributes to the appropriate MIG routine declarations, and for each one, delete the server code that deallocates the port or out-of-line memory.

5.2.7 Server Changes and Constraints for Short-Circuiting

As alluded to previously, servers need to adhere to certain rules and provide a small amount of functionality to the MIG stubs, if short-circuiting of RPCs is to work correctly. The main issue has do with the semantics of the threads used to carry out server requests, but there may be other limitations as well.

Normally, a Mach server will be multiply threaded and will be using some library, such as C-threads, to implement efficient user-mode threads on top of Mach's threads. There may be assumptions made by either the threads library or by the server itself about the threads executing server code. It is not clear that a foreign client thread can successfully come barging into the server code unannounced without violating some of these assumptions. The solution to this is to have the top-level server function in a short-circuited RPC call a new C-threads function that implements immigration of foreign threads into C-threads. See *Threads Library Changes*, below for details.

But this should only be done by the top-level server function when it has been entered via short-circuited RPC. It must be avoided if a message-based RPC is in progress, because in that case the current thread is the server thread that received the message and it is not foreign.

The simplest solution to this is to declare two server functions for each entry in the jump table of the RPC subsystem structure, one to be used in message-based RPC's and the other for short-circuited RPC's (the latter will call the former, after the foreign thread immigrates into the C-threads domain). We may also need a third entry in the jump table, for user-to-kernel short-circuited RPC's, but this design has not yet been completed.

In addition to the threads library issue, some versions of the OSF/1 servers expect each server C-thread to contain (at a certain offset from the base of its stack) a "uthread" structure containing thread-specific data needed throughout the server. This requirement has been eliminated in the newer, emulator-free architectures, so we will not worry about it.

A different problem with stacks occurs when a server that has been loaded into the Mach kernel accesses a Mach service by direct procedure call. Many Mach services are implemented using so-called "continuations", whereby when a thread needs to block, it saves a bit of state in a per-thread data structure and gives up its stack to a list of free stacks. Upon resuming, it is provided with a new stack from the free list and the saved state is restored in the new stack. Clearly, this allocation and reallocation of stacks will wreak havoc upon a server that is using a threads package unknown to the Mach kernel, with unknown constraints on the size, alignment and contents of the stack.

So there are two choices when a server thread accesses Mach services by direct procedure call: either continuations have to be disabled for that thread, or it has to switch to a Mach-kernel-provided stack upon entry to the Mach code. The latter is easiest to implement, and may be just as efficient as disabling continuations.

5.2.8 Threads Library Changes

The problem here is how to allow a foreign thread to enter a server via short-circuited RPC and become a legitimate citizen of the server's threads package. This is not as trivial as it sounds.

For instance, the C-threads library expects that each thread has a stack that is a predefined power of two in size, and is aligned to a multiple of that size. Furthermore, it expects the base of the stack to contain a pointer to the C-threads data about that thread..

[In order to avoid confusion, we will henceforth distinguish between user-library threads (such as P-threads and C-threads), Mach threads, as seen by the Mach kernel, and foreign threads, which are Mach threads that are not yet user-library threads, but may wish to become so.]

Alternatives

One way to solve the problem (as the Utah project did) is to have the server manage a pool of permanently dormant user-library threads. They are all blocked in some Mach service (probably mach_msg) unbeknownst to the user threads package, which considers them to be runnable threads. Since the threads are not executing, their stacks, and indeed their entire identity can be "borrowed" by a foreign thread entering the server for an RPC, then returned by that thread as it leaves. This method will only work if the threads package doesn't require that a particular user-library thread always run in the same Mach thread. This is not a problem for C-threads, but it might be for P-threads, the OSF/1 threads package.

Another problem with this alternative is that it requires rather a lot of code to be added to each server to fool the threads package in this way. Moreover, the code is machine-dependent (it involves stack-switching) and is prone to breaking when the threads package changes.

Finally, it has the disadvantage that the dormant Mach threads, which serve no purpose, are a waste of kernel resources. It may be possible to destroy them without the threads package noticing or caring, but then again, it may not.

A better way to solve the problem is to add an explicit new facility to the threads package, whose purpose is to allow a foreign thread to become a new user-library thread for the duration of execution of a specified function call. We describe the design of such a facility for C-threads, below.

Design

For concreteness, assume the new C-threads function is called cthread_immigrate. Its syntax and semantics are:

```
cthread_immigrate(function, &arg1, args_size);
```

The current Mach thread becomes a new wired C-thread that is executing the specified function with the specified list of arguments. When the function returns, the Mach thread resumes executing after the call to cthread_immigrate, but it is no longer a C-thread. While it is a C-thread, it has a different stack, so it must not attempt a longjump out of the function call.

The implementation of cthread_immigrate is simple. It is a small assembler function that pulls a C-thread identity (C-thread stack + thread_descriptor) off of a free list of dead C-threads, marks the C-thread runnable, switches to the C-thread stack and calls the function. It reverses all these actions when the function returns to it. It is imperative for good performance that it not be much more expensive than a null procedure call. Of course if it finds the free list empty, it will call a

high-level C-threads function to create a new dormant C-thread (one that is marked as “wired”, but has no associated Mach thread and is placed on the free list).

The reason for specifying the arguments as the address of the first argument together with the total number of argument bytes, is to avoid copying them more than once from the foreign thread’s stack, where they are at the time of call to cthread_immigrate. We pass a descriptor of the arguments to the assembler function, which first allocates a C-thread stack from the free list, then copies the argument bytes from the foreign stack to a newly pushed frame of the C-thread stack. It is hard to see how to avoid this one copy, but the overhead amounts to little more than one extra procedure call of the same complexity as the call to the client stub.

5.2.9 Short-Circuiting of One-Way Messages

So far we have discussed the replacement of message passing by direct procedure calls only in RPCs, which by definition consist of a message sent to a server by a client, which then passively waits for a reply message. Because the semantics are (by design) almost identical, it is straightforward to replace the RPC with a more efficient procedure call.

We would like to optimize one-way messages in a similar fashion, when the sender and recipient are in the same address space. At first glance, a one-way message can be viewed as a call to a “void” procedure, i.e. one that does not return a result, implying that we can replace the message with a direct call to a void procedure.

The problem with this is that it ignores the parallelism of message passing. When a one-way message is sent, both sender and recipient can proceed immediately as two parallel threads of control. If one were to short-circuit the message into a call to the server procedure that processes it, the effect is to make the sender of the message block until the recipient has finished processing it. Not only does this fail to exploit the parallelism that is available on a multi-processor, but it may be an unacceptable change in semantics, even on a uni-processor. For example, the page-out daemon in the Mach kernel sends one-way page-out messages to pagers, and it must not block when doing so or the system will quickly run out of memory.

Alternatives

All the same, there are more efficient ways to process a one-way message that is within an address space. We can avoid the marshalling of arguments into a message buffer, the overhead of port-name translation and the VM manipulations involved in OOL memory.

One way to do thisis to use the same short-circuiting algorithm in the MIG client stub that we use for RPCs. Assume that the server procedure called by the client stub knows whether it was reached by a real message or a short-circuited call. In the latter case, the server procedure, which is running in the context of the client thread, will do whatever may be necessary to activate a different server thread to process the request. The procedure will then immediately return control to the client, without blocking. To optimize this, the server could carry out non-blocking requests in the context of the client thread, only activating a different server thread for blocking requests. There are potential problems, however, in altering the order of execution between sender and recipient in this fashion. The most important case of sending one-way messages are the messages sent by the pageout daemon to pagers. It is crucial that the pageout daemon thread control any memory allocation it does during its pageout scans very carefully, to avoid running out of its pool of reserved pages. It is not safe, in general, for the pageout daemon thread to enter an unknown

pager (via short-circuited paging message) to do an arbitrary amount of work, even non-blocking work.

Given that the above issue can be resolved, there are at least two plausible ways that the server could activate another thread to process the request. One is to simply insert a description of the request in a queue, and arrange that one or more server threads are always processing requests from this queue. This sounds suspiciously close in complexity and performance to the operation of mach_msg.

Another approach, which may be more efficient, is to have a pool of blocked server threads (e.g. in C-thread condition_wait), and awaken one of them after modifying its saved state and stack so that when resumed, it will call the desired server function with the desired list of arguments, then go back to sleep. Note that if the thread to be awakened is an unwired C-thread, no Mach services are required to do so. We can simply change the state saved by C-threads and move it to the C-threads run queue. In this approach, there would be no message queue, but it is not clear whether the activation of threads by the client stub is too heavy-weight an operation to impose on the client. For example, if a large enough flurry of messages arrived in a short period of time, the pool of threads could be exhausted and it would be necessary to actually create a new server thread before the client could be allowed to resume. This might not be a blocking operation, but it could be rather expensive. A solution to this problem could be to revert to sending real messages when the pool of inactive server threads is too low.

Since there are serious open issues involved in all of the above alternatives, we will postpone any design for short-circuiting one-way messages until they can be resolved.

Appendix A. Proposal for an Improved Port Management Data Structure

By: Michael Condict and Faramarz Rabii

Abstract. The current design of the OSF/1 AD server has a number of deficiencies in the handling of requests. These include problems with renaming of ports to data structures, inefficiencies in dispatching of requests, inefficiencies in client/server communication, processing of messages to a particular port in a sequential order, limiting the number of server threads existing at any given time. We propose a mechanism involving the introduction of a new global data structure which will address all of these deficiencies. This structure, which can be used by all Mach-IPC-based servers, is a table that maps port right names to information about the object represented by the port and about the MIG subsystem associated with that object type.

The five problems mentioned in the abstract are discussed in some detail below, in order:

1) The problem of port renaming:

There are two distinct problems which involve port renaming.

- If the same client holds send rights for objects of different types, and both kinds of ports are renamed in the server, the client can crash the server by sending a message that is intended for one type, using a target port that is of a different type. Or worse yet, the client might hold a send right for a port that is not renamed to point to anything, and may use that in a message type that is supposed to go to a renamed port.

  Previous solutions to this problem required scattering "magic" type identifier fields across all the different structures (and they don't solve the problem of ports that don't point at any data structure).

- When server and client are in the same address space and the server migrates away an object, together with its receive right, which was renamed to a data structure (this happens in AD), the client is left with a send right that points to a deallocated data structure. This wreaks havoc when the server attempts to reallocate that data structure for a different object.

2) The problem of inefficient dispatching of requests:

Currently, each incoming message is sent to the MIG-dispatcher function for each subsystem, one at a time, until a dispatcher is found that will accept it. In AD there are twenty different subsystems in this sequential search, implying up to nineteen failed procedure calls to find the right subsystem, a very inefficient dispatching method.

3) The problem of unnecessary Mach messages:

In the current implementation, all client/server messages are sent through Mach messages, even when the two exist in the same address space. This involves a great deal of unnecessary over-

head which could be avoided by detecting the fact that the client and the server are in the same address space and short circuiting the RPC via a procedure call (see proposal by Reynolds, Condict and Mitchell) . The problem is that an efficient way is required to tell when the client/server are in the same address space and, if so, which server procedure to call.

4) The problem of sequentializing processing of messages to a port:

In the current implementation, multiple threads are listening on the same port set. Therefore, it is possible that more than one message to a given port can be received in a small time interval. Since scheduling of server threads is random it is possible that a later message (one with a higher sequence number) can be processed before an earlier one. Subsystems requiring sequential processing of requests have been forced to implement their own schemes involving suspension of the thread handling the later message until earlier messages are processed. These methods have two major problems. They waste system resources (threads) and are also subsystem specific. A general method which avoids thread suspension is desired.

5) The problem of too many server threads:

In the current implementation, there is no way to control the number of existing server threads. Each request is handled by a thread and when the number of free threads drops new ones are generated. The issue manifests itself in two ways:

> 5a) One problem is that a single port can have multiple threads handling requests for it at the same time. A user can generate a very large number of server threads by simply sending one way requests as fast as it can without waiting for the answer, and thus running the system out of resources.
>
> 5b) Even if problem "A" is solved it is still possible to have as many threads as ports which can also run the system out of resources.

It is not possible to simply enforce a maximum on the number of server threads without the danger of deadlocks. Therefore a method to control the number of server threads without the danger of deadlocks is desired.

The solutions:

We propose solutions to the above problems all of which involve the use of a new data structure mentioned earlier.

The first-level structure is a global, per-task table with one entry per port being listened to by the server. This table, called the "RPC port vector", is accessed very efficiently. It is stored as an array of pointers to the structures that contain the information about the object associated with a port. These per port objects are called "the RPC port structure" (see Figure 8).

Every port that can be the target of an RPC must be renamed by the creator to be a pointer to an unused position in this array. The associated structure is then filled in with the appropriate information. The RPC port structure holds information which is common for all ports as follows:

- object type: The type of object represented (e.g. vnode, file, credentials), if any
- object pointer: A pointer to the object, if any
- sequence info pointer: A pointer to a data structure used for sequenced objects or NULL otherwise. This structure is called "message_queue_-data"

- subsystem info pointer: A pointer to a per-subsystem struct called the "RPC subsystem structure "

The RPC subsystem structure contains the following fields:

- subsystem minimum number: Lowest message id number in the subsystem
- subsystem maximum number: Highest message id number in the subsystem
- jump table: A table containing addresses of routines for that subsystem, indexed by message id number.
- dispatcher: A pointer to the MIG generated dispatch routine for the subsystem

And a third (optional) structure "message_queue_data" used for objects which require sequencing. This structure, pointed to by the RPC port structure contains the following:

- lock: The protective lock
- current sequence number: Sequence number of the next message to be processed
- message queue count: Count of messages queued for this object
- message queue: A queue of messages for this object

A crucial fact is that the RPC port vector can never be moved, so it must be allocated large enough to handle as many ports as the program will ever hold receive rights to. 10,000 entries seems safe, without wasting too much address space.

We now discuss how the above structures help solve each of the problems stated in the abstract one by one.

1) The problem of port renaming:

Port renaming was used as an efficient method to access an object associated with a port. In the above scheme when a port is associated with an object the port is renamed to an unallocated entry in the port vector, which is set to point to a newly allocated port structure. A pointer to the object and an indication of its type are then inserted into the port structure.

This allows finding the object through two indirections: one to get to the port structure, and the other to get to the object. This is slightly less efficient than casting the port as the object but still considerably more efficient than a hash queue search.

The method solves both problems mentioned before.

- Since the port structure has an explicit field indicating the object type, there is no danger of incorrectly referencing a field of the object and thus crashing the system. Furthermore the subsystem number in the message can be compared with the subsystem range stored in the subsystem structure and invalid messages can be detected and rejected (in fact the object type is not even needed, if the subsystem only handles one type of object).
- In this scheme a receive right is renamed to an entry in the port vector instead of an object. When a server and client are in same address space and the server migrates away an object together with its receive right, the

pointer in the vector is marked invalid but unavailable. It becomes temporarily unusable until the client gives up the send right. However all other data structures previously associated with the port can be re-used. This results in having to make the table a bit larger but since the table entries are only four bytes it is not expected to be a major issue.

2) The problem of inefficient dispatching of requests:

The subsystem structure now holds a pointer to the appropriate dispatch routine, which is inserted at the time of port creation. The server will use this pointer to call the correct dispatcher on the first try, thus avoiding the current inefficient linear search method. 3) The problem of unnecessary Mach messages:

The new method provides a very efficient way for clients to discover that they are in the same address space as the server, so that they can short-circuit the RPC to a procedure call. If a send right name has an in-range value, the client looks it up in the port vector. If a valid entry is present there, the server must be in this same task (because entries are only made by servers and the receive right name is always the same as the send right name). Also, in this case, the associated subsystem structure contains a pointer to the jump table that can be used to short circuit the RPC.

4) The problem of sequentializing processing of messages to a port:

The new method provides a centralized solution to the sequentializing of requests (for those subsystems that want to do this). When a sequenced object is created, the message_queue_data mentioned earlier is created and is pointed to by the port structure with its fields initialized as follows:

FIELD	INITIAL VALUE
lock	unlocked
current sequence number	zero
message queue count	zero
message queue	empty

When the server receives a message for such an object, it will examine the message_queue_-data structure and act as explained in appendix B.

This method requires that message buffers be dynamically allocated in the server loop, which will involve some small changes to the current code.

This will fully solve the sequencing problem while also avoiding wasting threads by suspending them pending processing of lower sequence numbered messages. It also provides for additional deadlock-free throttling control of messages to a port via a modification to the above algorithm which will be discussed next.

5) The problem of too many server threads:

The above problem manifests itself in two ways as discussed earlier and the solutions to each are discussed separately.

5a) Limiting the number of threads per port can be achieved by treating all messages in a similar way as those which are sequenced with a slight modification to the message_queue_data

structure. In this case a thread servicing the object will have to mark the message_queue_data structure as BUSY, thus preventing other threads from servicing it at the same time. Instead other threads will simply queue the message buffers, and let the one thread service them all. The message_queue_data structure used for sequenced objects can be modified as follows to accommodate the new requirement.

A new field indicating status information for the object will be added and will contain the following flags:

- SEQUENCED: If this bit is set, the object is of sequenced type
- BUSY: Only valid for non sequenced objects, indicates a thread is currently servicing this object
- PORT_IN_SET: Indicates the server is currently receiving messages on this port (it is in the server's port set(s)).

The algorithm given in appendix B accomplishes this task.

The length of the message queue in both the above case and that of sequenced objects can be kept reasonable by simply removing the port from the port set when the length gets above a high water mark and re-inserting it into the port set when the length drops below a low water mark.

This gives us per port throttling

5b) Limiting the total number of threads in the server can be achieved by adding an adjustable delay before each server thread returns to the top of the server loop to listen on the port set. Whenever the number of server threads gets above a high water mark a delay of "n" seconds is introduced before the server thread listens on the port set. The delay can be adjusted by monitoring the change in the number of server threads. If they are increasing despite the delay, it can be increased. If they are decreasing (but still above the low water mark), it can be decreased. The exact nature of this adjustment must be determined through experimentation with different types of algorithms, and delay values. This will give us system wide throttling

Conclusion:

We have identified five problems with the OSF/1 AD server's handling of requests. Each of these problems has been addressed and solutions have been proposed. A common requirement in all of the solutions is the for a global table which associates ports with information about the objects they represent. An efficient way of implementing this table has been proposed which will help solve the mentioned problems.

Appendix B. Use of Port Management Data in Message-Based RPC's

[Note: this section is a continuation of the proposal made in *Appendix A*.]

Below is a combined algorithm for processing of MIG messages in the main server loop, including per port throttling, and sequencing.

Description of Variables

```
my_message:       pointer to message being processed

throttling:       boolean indicating if throttling is active

sequential:       boolean indicating if messages are to be
                  processed in sequential order.

port_in_set:      boolean indicating if port is in port set
                  being listened on
```

NOTE: low and high water marks may be per object, per subsystem or system wide.

Algorithm

```
receive message in ux_server_loop
my_message = NULL
lock RPC port structure
if ((sequential && current_sequence_number == message just received) ||
                          (!sequential && port structure ! BUSY)) {
         my_message = message buffer just received
         if (!sequential)
                  set message_queue_data BUSY
} else {
         if (sequential) {
                  queue message on the message queue in sequential order
         } else {
                  queue message on the message queue
         }
         increment message queue count
}
unlock port structure
while (my_message != NULL) {
         process my_message
         deallocate my_message buffer
         lock port structure
         if (throttling) {
                  if (message queue length > high_water_mark &&
                                                    port_in_set) {
                           remove port from port set
                           port_in_set = FALSE
                  } else {
                           if (message queue length < low_watermark &&
                                                    !port_in_set) {
                           insert port into port set
```

```
                        port_in_set = TRUE
                }
        }
        if (sequential) {
                increment current sequence number in port structure
        }
        if message queue is not empty {
                if ((sequential) && current_sequence_number matches
                        first message on the queue) ||
                                        !sequential) {
                        dequeue message
                        set my_message = message just deqeued
                }
        }
        unlock port structure
}
allocate message buffer
listen for more messages in ux_server_loop
```

Acknowledgments

The ideas expressed in this design have benefited greatly from the free and vigorous communications that frequently take place among members of the OSF Research Institute. In particular, discussions with Dave Bernard, Alan Langerman, David Black and Franco Travistino have provided valuable input.

References

[1] Brian Bershad, "The Increasing Irrelevance of IPC Performance for Microkernel-based Operating Systems", *Proceedings of the USENIX Workshop on microkernels and Other Kernel Architectures*, USENIX, April 1992, pp 205-212

[2] Peter Druschel, Larry L. Peterson, Norman C. Hutchinson, "Beyond Microkernel Design: Decoupling Modularity and Protection in Lipto", *Proceedings of the Twelfth International Conference on Distributed Computing Systems*, Yokohama, Japan, June 1992, pp 512-520.

[3] J. Lepreau, M. Hibler, B. Ford, J. Law, D. Orr, "Experience with Flexible Operating System Structuring", *USENIX Mach Symposium*, Santa Fe, April 1993.

[4] Graham Hamilton, Panos Kougiouris, "The Spring Nucleus: A Microkernel for Objects", *Sun Microsystem Laboratories*, SMLI TR-93-14, April 1993.