Redirecting System Calls in Mach 3.0

An alternative to the emulator

Simon Patience

Open Software Foundation
Research Institute
2, Avenue de Vignate
38610 Gières
France
Tel: (+33) 76 63 48 72
E-mail: sp@gr.osf.org

Abstract:

In order to implement system calls, current Mach 3 based operating system personality servers all make use of Mach's system call redirection mechanism together with an emulator co-resident in the applications address space. Experience with OSF/1 MK has shown that this is problematic for OSF/1 system call semantics, system security and integrity. This paper looks at what these problems are and why they occur. It also describes new kernel mechanisms to implement OSF/1 system calls which address these problems and our experiences implementing them. Finally the benefits and shortfalls of the new mechanisms are discussed both in relation to conventional computer architectures and also with respect to new architectures, such as multicomputers and clusters.

1. Introduction

For the past two years the OSF Research Institute at Grenoble, France, has been developing a version of OSF/1 based on Mach 3. The architecture for this system was basically copied, for the most part, from the BSD server from CMU. The model used was that the application programming interfaces were implemented in a single server executing in user space. Access to the server from the application was via an emulator that was resident in a fixed place in the address space of every process. As time passed it was found that the advantages of the emulator also brought many disadvantages, some performance, some semantic and some maintenance. It became clear that, for implementing OSF/1 semantics, the emulator was not the best solution. During the past 9 months, new mechanisms and improvements to existing mechanisms have been developed to be able to better support servers implementing Un*x like semantics.

This paper discusses the motivation for the emulator, why it was inadequate for the needs of OSF/1, and what was used to replace it. In addition, the impact of this solution on other architectures currently being used in OSF RI is considered.

2. Why an emulator?

The motivation for the emulator was originally for binary compatibility. Traditional statically bound BSD binaries used hardware traps to communicate their request for a system call to the kernel. With Mach 3 however, the kernel does not know anything about BSD semantics, or even which operating system personality is running. As a consequence, these traps had to be redirected in some way. In addition, there was the apparent contradiction of style between BSD system calls and Mach's message based communication mechanisms.

The emulator is essentially part of the operating system personality. The emulator is a second program that is loaded into an unused part of the address space of every process. When the calling application does a system call trap, the emulator is entered by a simple redirection back into a predefined part of its address space. It can then perform the translation between a trap based interface directed at the kernel to a message based interface directed at the server. Figure 1 shows the code path of a normal system call using the emulator.

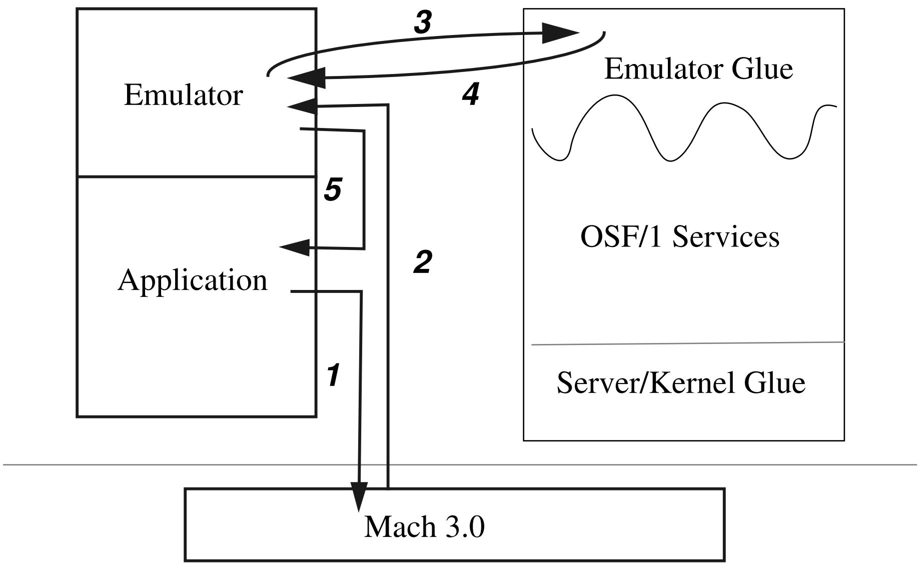

Figure 1: Code Path of an emulated system call

The application makes the system call in the normal way (step 1) by executing a trap instruction into the Mach kernel. The kernel looks up the trap number in a per task redirection table, saves some state on the user stack and returns back to the application (step 2) at the address in the table. The emulator starts to process the system call and normally sends a request Mach IPC message to the server to perform the system call (step 3). Having executed the system call, the server sends a reply message to the emulator (step 4), sending back any resulting data. Finally (step 5), the emulator returns back to the address in the application that the kernel saved on the stack having stored the results of the system call in the appropriate places.

In addition to doing this simple conversion, the emulator soon became the focal point for performance optimizations. Some system calls were able to be entirely implemented in the emulator either due to the emulator caching information or due to introduction of shared data between the emulator and the server. Some other optimizations were made by having an I/O page between the server and the emulator, which eliminated copies into and out of messages.

The other advantage the emulator was seen to have was that in systems which supported shared libraries, the emulator could ultimately replace the system call stubs in libc.

2. Why not an emulator?

The problems introduced by the emulator can be divided into the following categories;

- System security
- System integrity
- System resource management
- Compatibility and maintenance

It should be noted that some of these problems are OSF/1 specific, while others are Un*x like specific and a few are generic to any personality. The root of most of the problems with the emulator is that it is untrusted due to the fact that it is completely unprotected from the application.

2.1 System security

Violations of system security mean that, although the server continues to function normally, the security policy is potentially violated. This means that the operating system personality is fooled into providing services for a process which should not receive them. Using the emulator, all the Mach kernel services are necessarily exposed for direct use by the application. Because Mach services are not mediated by the server, the kernel will perform operations for anyone who presents the appropriate port. The biggest problem areas are task and thread ports which can be passed to other processes via Mach IPC. Using the current emulator based OSF/1 MK or the CMU BSD server, it is relatively easy to become root or force the server to execute system calls for privileged programs which they did not request.

One way of becoming root with the current system would be for a process to modify its emulator code/data such that some innocuous system call will do something to breach security, such as exec() a shell, instead of performing its original service. Then the process exec()s a setuid-root program known to use that system call. Since the emulator is inherited without its text or data being re-initialized, the setuid-root program will use the malicious emulator and security is broken. Although it is possible to close these types of security holes by replacing the task and reloading the emulator for setuid execution, it is at the expense of performance and general increase in complexity.

Removing the emulator does not change these problems in any way but it does allow the possibility of disallowing non-privileged programs from direct access to Mach services, which is what is done. In OSF/1, Mach services are defined for use by system extensions. Consequently in initial versions of OSF/1 MK, a process needs to have the same privilege to use Mach services as it needs to load “kernel” extensions into the server. The use of the services will be re-enabled when the use of Mach services can be brought within the security policy of the operating system personality.

2.2 System integrity

System integrity is breached if a process can get the server to perform incorrectly and at worst, either hang or crash. There are a number of ways in which the emulator exposes the server to attack. OSF/1 MK inherited various mechanisms from the BSD server which cause problems. In addition the direct unmediated use of Mach services causes others.

There are a number of attacks that the server must defend against by using the Mach kernel primitives to manipulate resources that the server believes are under its control. In particular, the problems are processes calling task_terminate() instead of exit and processes calling thread_create() and then these unknown threads making system calls. Processes vm_allocate()ing or vm_deallocate()ing memory causes confusion as the server attempts to manage the processes address space to control the brk area and to flush mapped regions etc.

Processes can also send strange messages to the ports on which the server collects system call requests. These can either be with the correct ID and the wrong contents (for copyin() for example) or with additional out-of-line data that the slowly clogs the servers address/port space.

2.3 System resource management

Attacks of this form can come in two forms. Either the application uses more resources than it is allowed or it forces the server to use resources which could potentially cause denial of service or even to cause the server to crash or panic. Again this is not explicitly enabled by the emulator but is due to the direct use of Mach services which the presence of the emulator requires.

Exposing the external memory manager interfaces means that a malicious program could establish a mapping with an uncooperative pager which could not respond when the server accessed it. This would tie up server threads forcing the server to create more, if it wished to continue to function, until the system resources were exhausted.

The kernel itself has no mechanism for resource control and will blindly satisfying most requests to allocate resources such as threads. ports or vm address space until that resource is exhausted for the whole system.

2.4 Compatibility and maintenance

There are a number of problems with the emulator architecture which are perhaps more specific to OSF/1 than to other personalities as they are caused by some OSF/1 features not normally found in Un*x systems.

OSF/1 allows for kernel loadable modules. One type of subsystem that can be loaded is a module that implements new system calls. New subsystems configure themselves dynamically when loaded, which is normally at boot time. Access to the emulator is set up when loading *init* and only system calls that are known about at this time are redirected to the emulator. This emulator vector is then inherited from task to task. Thus, the system call table is replicated for every process in the system and there is no way to update the vector for every task in the system when new system calls are added.

For almost the entire time that OSF/1 MK was being worked on in Grenoble, there were always outstanding signal bugs. Even after almost two years of development, the signal semantics of OSF/1 MK were not totally compatible with the integrated kernel, and indeed there were still some bugs remaining that would cause the server to hang. This was caused primarily by splitting the signal functionality between the server and the emulator, and then not trusting the emulator to do the job. The signals code in OSF/1 MK deviated from its integrated kernel equivalent more than any other piece of code in the system. It changed significantly in virtually every release and there were still corner case bugs after more than 12 releases. Part of the problem of delivering signals from the server is that, if the application is in the emulator, it is hard to find the real user stack to build the signal frame, as the emulator changes stacks when it is entered. When or if you do find it however, it has data the emulator has saved there and so cannot be used anyway. This is also a problem when trying to dump core.

In order to improve the performance of the emulator, a page of state is shared between the server and the emulator. This page holds various information normally found in the utask structure. This means that structure offsets are different and thus kernel extensions are not binary compatible. It also means that the server cannot trust this information and must check that it is sane before it can be used.

In addition, the kernel-to-application interface is well defined in the integrated kernel, it is copyin() and copyout(). When any functionality of a system call is implemented in the emulator, copyin() is implemented by the emulator knowing, in advance, what is to be copied and sending the data along with the message containing the system call request. Similarly, copyout() is implemented by extending the reply message that is sent back at the end of the system call with the data to be copied and then the emulator does the copy to the correct place in the user's address space. The problem with this is it means that copyin() only works if arranged in advance and any copyout()s are deferred until the end of the system call. As the size of the return message was limited, this also limited the amount of data the server could copyout(). This all means that any kernel extensions expecting, for example, that copyout() followed by copyin() of the same address would see the data written would not see the expected result. As a consequence, OSF/1 integrated kernel extensions are not even source compatible with OSF/1 MK.

Upgrading becomes more expensive due to the amount of code that is changed to support the emulator that has to be repeatedly applied to each new version of OSF/1. This also means that OSF/1 MK is not using OSF/1 IK code for this functionality which increases the chances of the semantics differing between the systems and introducing bugs into OSF/1 MK. The issue of kernel extensions becomes more important as the server is upgraded to OSF/1 1.2 and beyond as virtually all subsystems (NFS, System V FS, LVM, network protocols) are dynamically loaded during booting and it is unreasonable and expensive to expect people to re-code their kernel extensions for the sake of changes introduced by the emulator.

Even in itself, the emulator is a maintenance nightmare for OSF/1 MK. It is a complex piece of code involving stack switching code, port and memory management, code for signal delivery (which incidentally is duplicated in the server) and address fault recovery. There is communication between the emulator and server via shared memory, which the server cannot trust and must therefore defend against, thus forcing the complexity into the server.

Finally, there were a number of little problems. Errant programs would fail in unexpected ways if they scribbled over the emulator. Debuggers do not understand the emulator and so follow the trap redirection into a part of the address space that the programmer may not have realized (and probably does not want to know) was mapped. In OSF/1 AD, the emulator even has extra threads which are not part of the application to further confuse and add failure modes. The mapped file implementation also involved passing tokens between emulators to indicate ownership which might not be given up, so a complex time-out and revocation scheme was invented. There seemed to be excessive data copying from the servers buffers into the shared I/O page and then into the user's buffer for read()s and write()s. The stack allocation and switching at the start of every system call seemed like another expense that was not needed. Ultimately it became too much and it was felt as though more effort was being put into maintaining the emulator than moving the system forward functionally.

3. What instead?

The solution was to solve two problems: The communication of a system call request to the server and correcting the implementation of copyin() and copyout() in the server.

The request from the application had to reach the server in an efficient form and as this is Mach, this should be a message. The application request for OSF/1 was in the form of a trap instruction, or equivalent, in the system call stub in libc. This meant that it had to be a kernel mechanism. The obvious choice was to send an exception to the appropriate exception port. There were numerous problems with this solution however.

The exception message Mach currently sends contains send rights to the task and thread ports of the thread that generated the exception. Sending port rights in messages is relatively slow compared to sending data alone and for system call delivery, speed is of the essence. In addition, it is certain that a system call will need to modify the state of the calling thread. With the current message format, this means the server having to call thread_get_state() and thread_set_state() for every system call. The added complication of debuggers interposing the exception port in order to intercept exception messages also means that these messages could not be sent on the regular exception port. So what with slow, port laden messages being sent to the wrong port and two additional Mach system calls being needed for every OSF/1 system call, it was clear that this would not work. However the general idea was right and this led to the exception mechanism being generalized.

In order to solve the limitations of the emulator based server implementation of copyin() and copyout(), the functions had to be made semantically identical to the integrated kernel. With the current Mach interfaces, the only available way of reading or writing to another task's address space is to use vm_read() or vm_write() respectively. Neither of these calls in their current forms are capable of doing the job for copyin() and copyout().

The problem of vm_write() is simply that it only writes page sized and aligned quantities. This makes it impossible to atomically copyout() data that is not page aligned in both size and address in both the server and the target process. For example, to copy 40 bytes would involve the server reading the page in which the addresses it wants to modify are, modify the 40 bytes and then write the entire page back. If that page was shared by multiple threads, either in the same task or in different tasks, then there is no guarantee that one of these threads will not touch the part of the page that the server does not wish to change and if it does, the server's page write would overwrite these modifications.

The problem for vm_read() is simply that it returns the data out-of-line. (It also page aligned in both size and address, but that is not a semantic problem). As a consequence of this, the data is never where it needs to be and so it must be copied. The original data returned out-of-line must then vm_deallocate()'d. copyin() is not a function that can tolerate making two Mach system calls as it is called too frequently.

The result of making these two changes, adding the new exception mechanism and fixing copyin()/copyout(), is that the architecture looks remarkably like that of the integrated kernel, with exception that the OSF/1 services are in user space rather than being bound in with the Mach kernel. The interface between the application and operating system personality, OSF/1 in our case, is back to being trap from the application to the personality and copyin()/copyout() from the personality to the application.

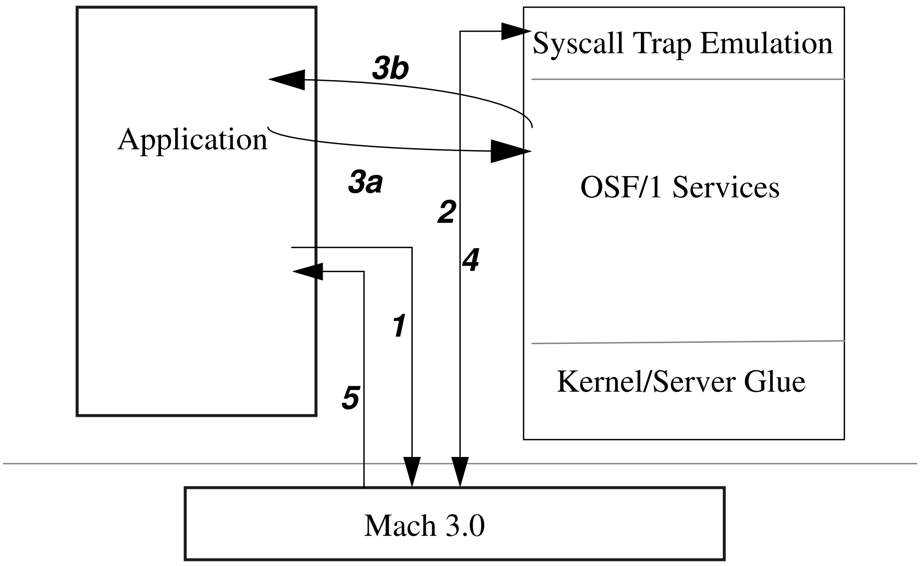

Figure 3: Code path for a system call without an emulator

The application makes the system call request in the normal way (step 1) by executing a trap instruction into the kernel. The kernel generates an exception message which it sends to a port that the server has set up especially for handling system call requests (step 2). The server then processes the system call in the same way as in the integrated kernel, calling copyin() (step 3a) or copyout() (step 3b) whenever it needs to get or return data to the application. Finally the server returns any modified register state to the kernel (step 4) in the reply to the exception message, and the kernel resumes the thread having restored it (step 5).

4. A generalized exception mechanism

Having initially tried having the kernel send specialized messages when traps occurred, it was found that the code used in the kernel was remarkably similar to that to send exceptions. The motivation for considering a separate interface was simply performance, i.e. to avoid sending ports. From the implementation however, it was clear that what was trying to be achieved was to simply raise an exception on the occurrence of a trap instruction.

The current Mach exception mechanism treats all exceptions alike, in that the kernel generates a message and sends it to a thread's exception port, if it has one, or the task's exception port, if not. The message that is sent is the same in all cases and it is intended simply to identify the task and thread that caused the exception. The new mechanism changes this in two fundamental ways.

Firstly, to obviate the need for a server to use thread_get_state() and thread_set_state() when exceptions occur, new versions of the exception message were defined in which the state is sent and the modified state returned with the reply, which the kernel will set before resuming the thread. In addition, one of the new interfaces does not send the task and thread port of the thread causing the exception, making the assumption that the thread can be determined by the port that the incoming message appears on. The MiG exc_server() routine calls either catch_exception_-raise(), catch_exception_raise_state() or catch_exception_raise_state_identity(), depending on the ID of the exception message the sent by the kernel. The new exception message interfaces are listed in Appendix A.

Clearly, the state needed for dealing with system call exception messages would not necessarily be the same state as that for dealing with arithmetic or address faults. Consequently, a way of associating what flavor state is required to be sent in response to a given exception type was introduced.

Secondly, to avoid confusion for debuggers that wish to interpose exception ports for the applications that they are controlling, you would not want to send system call exceptions, which the debugger is not interested in and probably couldn't handle, to the same port as arithmetic faults, which the debugger is interested in and can handle. Due to this, and since it is likely that some servers will want to use different threads to handle the different types of exception, a new interface makes it possible to associate different ports to different exception types for a given thread/task. The behavior specified determines the type of message the kernel will send and the flavor indicates the flavor of thread state the kernel sends with that message. Obviously, if the behavior is EXC_DEFAULT indicating that catch_exception_raise() is to be used, then the flavor is ignored. The new exception port manipulation interfaces are listed in Appendix B.

The types of exception supported are those returned currently by catch_exception_raise() in the exception parameter, plus the additional types EXC_TYPE_SYSCALL and EXC_MACH_-SYSCALL. However it is expected that certain architectures will wish to define more.

In addition to the current thread state flavors, a new flavor was added, THREAD_STATE_SY-SCALL. The definition the contents of this state is machine dependent but the intention is that it contains a small subset of registers adequate for handling most system calls for most personalities. For the i386, this register set is currently 5 registers (as opposed to 17 for the full set).

Note that

```
task_set_special_port(task, THREAD_EXCEPTION_PORT, new_port);
```

is equivalent to

```
task_set_exception_ports(task, EXC_MASK_ALL & ~EXC_MASK_SYSCALL,
                                     new_port, EXCEPTION_DEFAULT, 0);
```

The same will apply to the other task/thread get/set exception calls. However a backwards compatibility problem has been introduced as the format of the catch_exception_raise() interface has changed (for a completely unrelated reason). So, in our kernels, setting the exception ports with task_set_special_port() will generate the old catch_exception_raise() format message but doing the equivalent operation with task_set_exception_ports() described above, the new catch_exception_raise() format is used, thus it was possible to achieve both source and binary compatibility.

Delivery of exceptions in the kernel uses exactly the same algorithm. When an exception occurs, the kernel tries to deliver the exception message to the thread exception port for that exception type. If that does not exist or returns a value other than KERN_SUCCESS then the task exception port for that exception type is tried. If the exception is not handled after both possibilities have been tried, the kernel terminates the task, as before. Note it is perfectly possible and reasonable that the task and thread ports for a given exception type have different behaviors and flavors set.

5. Solving the copyin()/copyout() problem

Two solutions to this problem were explored. One was relatively conventional in Mach terms and the other rather controversial. The first was to 'fix' vm_read() and vm_write() such that they could be used in such performance critical code as copyin() and copyout(). The second involved the server mapping arbitrary addresses in the target process.

5.1 Using vm_read() and vm_write()

The change to vm_write() was simply to allow the specification of addresses in both the source and the target tasks that are not page aligned and to allow quantities of data that are not multiples of a page in size. In fact a large number of people had seen this as a bug in the specification of vm_write() (there was even a bug report against OSF/1 1.1 for this) so this seemed like an uncontroversial change. It had the added advantage that it did not effect existing callers of vm_write(). Clearly if unaligned data was specified, there was no possibility for the kernel to perform any VM manipulations which would avoid physical copying of data; however, the implementation was careful to ensure that if correctly aligned data was specified then the appropriate VM magic would occur.

The change for vm_read() was slightly more awkward. The problem was caused by the fact that vm_read() returns the data out-of-line. For copyin() semantics this has two ramifications; the first being that the data is not where you want it thus causing a copy, and the second being that the returned data needs to be vm_deallocate()'d after use thus incurring the extra cost of another Mach service. The solution was to introduce a new class of Mach interface; one which returned out-of-line data in a buffer that the receiver already had mapped.

This was done by introducing a new interface, mach_msg_overwrite(). The semantics are exactly the same as mach_msg() with out-of-line data with the exception of final delivery to the receiver. The receiver specifies an already mapped address and range in which the kernel is to put any out-of-line data that the message may contain. If the specified buffer is too small, then the call to mach_msg_overwrite() fails. Once this call existed, it was simply a case of changing MiG to use it. The resulting interface was vm_read_overwrite() which is semantically identical to vm_read() with the exception that the call provides the buffer. Using this interface, there was no need for the bcopy() (as the kernel did it or the equivalent) but more importantly, the vm_deallocate() was removed from the copyin() path.

6. Using vm_remap()

There was still concern that one Mach system call per copyin() or copyout() was still one too many, especially considering that performance of the system would be compared against the integrated kernel, whose implementation of these functions normally amounted to the equivalent of a bcopy(). To solve this problem, a new interface was added to Mach which seemed to break one of the rules of Mach interfaces, that of being location transparent. The interface, called vm_remap(), maps arbitrary regions of one task's address space into the caller's address space.

The main issue is that the interface is not location transparent in that if the target task was not on the same host as the caller, then the call would fail. The reason that this will fail is not an inherent problem with the interface as it is specified, but it is more due to the fact that most pagers do not know how to manage multiple kernels (the netmemoryserver being the obvious exception). There is the same problem with default_pager_object_create being used by two clients of vm_map on different hosts. Also, there is no mechanism to allow the kernel managing the source task to inform the kernel that is managing the target task of the memory objects that represent the address range in the source task. This problems could probably be fixed but it would involve some inter-kernel communication. Vm_remap() will work on multiple kernels within a NORMA domain even across different nodes as the NORMA XMM performs basically the same job as the netmemoryserver within the NORMA domain and the NORMA kernels cooperate with task address space management, as this is needed to support norma_task_create().

The call to vm_remap(), listed in Appendix C, is similar to vm_map() except that instead of a memory_object port being supplied to the kernel, a task port and a virtual address range is specified. The kernel then maps the underlying objects of the source address range into the target task's address space. The implementation of vm_remap() simply replicates the kernel data structures in exactly the same way as would have happened if one of the tasks was specified as the parent task to task_create() and this portion of memory was marked as VM_INHERIT_SHARE. The main reason for this choice was to avoid any new kernel data structures or new vm relationships which would increase the complexity of the VM generally.

The disadvantage of mapping the underlying objects instead of somehow sharing address maps, is that deallocating the addresses in the application does not invalidate the mapping in the server. Consequently the server has to be very careful when it deallocates memory on behalf of the application to ensure that the equivalent server mapping is deallocated also. Even worse is that if an application used Mach primitives directly to deallocate this address range, the server would never get to know about the change and any subsequent system call specifying this address range would use the now stale mapping in the server. However to really share address ranges would involve a considerable change to the kernel management of address maps which was not consider either desirable or necessary.

Two different caching schemes were implemented in the server to take advantage of existing mappings. One was a system wide cache and the other was a task local cache. Both were prototypes and need further work in one area or another.

The system cache consisted of a hash table of mappings which were indexed by application virtual address and PID. A LRU (Least Recently Used) list of mappings is maintained for garbage collection when the hash table gets full. This had the advantage that on a quiet system, a single application could use as much of the cache as it liked but on a busy system, an application that used buffers distributed throughout its address space would use an excessive number of cache entries and thus cause better behaved applications to garbage collect the cache to find a free slot.

The second caching scheme was to maintain a simple linked list of mappings per task. Limits on the number of mappings and the amount of server virtual memory were applied to the task to prevent a task from using up server resources excessively. Two hints were maintained into the linked list based on address to speed the lookup.

Both caching schemes seemed to perform similarly with cache hit rates in excess of 90%. The system wide cache was marginally slower but this was caused by two deficiencies, one in each cache implementation. The system wide cache did not implement hints (which in the per task scheme had a very high hit rate, thus obviating the cache lookup completely) and the per task cache was not thread safe so avoiding lock overhead. I doubt if there would be a significant difference in performance if these two problems were fixed.

7. The resulting system

As a result of using per thread exception ports, a large amount of restructuring of internal data structures occurred in the final implementation. In fact, to simplify other things, the original OSF/1 kernel internal data structures were reintroduced, that is the Mach 2.5 task and thread structures. The system was very stable remarkably quickly, passing VSX and VSE to the same level as OSF/1 MK V4.1 within a few weeks of getting to multi-user mode and ran AIMIII to 250 users. This is compared to the emulator based server which took much of a year to get to the same level of robustness and functional completeness. The main reason for this was the significant increase of reuse of mature OSF/1 Integrated Kernel code rather than writing new code to implement the same functionality, as was done for the emulator. The code reuse was so great that even the integrated kernel's function syscall(), which handles system call setup, could be virtually entirely reused, with 15 extra lines of preamble, 2 postamble and a name change to catch_exception_raise_state().

As the internal environment became indistinguishable from the integrated OSF/1 system, virtually all the subsystems, e.g. ufs, vfs, net, netinet, etc., were reused totally unchanged. This in turn has made the upgrade of the 1.0.4 based version of the server to be based on OSF/1 1.2 to be achievable in a far shorter time.

The table below shows performance ratios of various versions of the no-emulator server compared to OSF/1 MK V4.1, which is the latest emulator based version of OSF/1 MK. The first 6 benchmarks are micro-benchmarks consisting of a 10 iterations of 1000 loop tests of either one, or a small number of system calls. The choice of system calls was based on the number of copyin()/copyout() operations they performed rather than frequency of use and thus are not representative of the overall performance that could be expected of the system, but more indicate the behavior of the mechanisms that are the subject of this paper. The AIMIII tests at 24 and 32 users give a better idea of overall system behavior. All the tests were performed using the same kernel which was an

	OSF/1 MK 4.1	vm_remap (system cache)	vm_remap (task cache)	vm_read_over write
getuid	1.00	1.04	1.04	1.05
read/lseek	1.00	0.73	0.71	1.06
File copy	1.00	0.81	0.88	0.88
uname	1.00	1.00	0.94	2.29
exec	1.00	1.06	1.04	1.37
sigaction	1.00	12.99	11.95	23.74
AIMIII 24 Users	1.00	1.04	1.01	1.42
AIMIII 32 Users	1.00	1.03	1.00	1.48

Performance Ratios of OSF/1 MK V4.1 using the emulator Vs. No-emulator servers

extended NMK13.3 with support for the no-emulator servers.

From the performance perspective, the new architecture was about performance neutral with OSF/1 MK V4.1, the latest version which used the emulator. This is encouraging as V4.1 is a product of many performance optimizations implemented over a year whereas there has been no effort put into optimizing any part of the no-emulator server yet. As there are known performance problems in the no-emulator server, this augurs well for the final performance of this server.

The vm_read_overwrite()/vm_write() still seems to have some performance problems. Admittedly there are some known kernel optimizations that can be made to these calls, but it is still clear that the server can afford the Mach call for every copyin() and copyout() operation. The high cache hit rate seems to be helping considerably for the vm_remap() versions of the server but neither caching strategy seems to be a clear winner judged simply on performance. Also, as neither caching scheme was fully implemented, it seems premature to make a judgement between them until they have been.

8. Suitability for other architectures

The main architectures that OSF has to consider, beyond traditional uniprocessors and symmetric shared memory multiprocessors, are multicomputers, otherwise known as NORMA (NO Remote Memory Access) machines, and Clusters of cooperating machines sharing a single system image. Both systems have a set of similar problems when it comes to distributing services, even though the details of the solutions may differ slightly. It was important that any change of server/kernel architecture was usable by systems such as these.

One of the main differences between this architecture and that using the emulator is that, as with the integrated kernel, the applications system call request enters the TCB (Trusted Computing Base) immediately (the emulator is not a trusted piece of code). As the only way of getting to the system call service point is via a single Mach port, the system call exception port, this implies that all system calls are, at least initially, handled by the same server. At first glance, this may not look as if it matches well with the idea of distributed operating system services, however having a protected local entity has many benefits that the emulator solutions could not properly provide.

On each node of these systems there can be a local agent, which acts as the trusted router of system call requests for all the tasks on this node. This allows the agent to play its part in distributed database management (for things like credentials, process management, etc.) like the untrusted emulator never could. In addition, the local agent can also play the part of a trusted cache manager by performing read ahead on file data not only to bring the data off the disk, but also to bring it local to the task so that a subsequent read would not have to leave the node to be satisfied. The agent will also at least have to participate in, if not initiate, process migration between nodes. None of this could be satisfied if the agent was not a trusted piece of code.

This does not mean however that an application may never communicate directly with a remote server. Once the problems of exposing Mach interfaces to applications (briefly described in Section 2) have been solved, a new libc could be implemented which used the services of both a local agent and remote servers, using the agent as the distributor of access means (i.e. ports) to the remote services.

Both the next version of OSF/1 AD (for multicomputers) and the version being designed for clusters are being based on the no-emulator technology.

9. Conclusion

The work is still not fully implemented in a final system. The 1.0.4 prototype was missing a number of key optimizations and small pieces of functionality were not implemented as they were seen as not to be relevant to this proof of concept. However, it is being used as the basis for the OSF/1 1.2 based version of the server, which will be the first version of OSF/1 MK that can be seriously considered for productization and is the first version that is expected to be able to attain C2 and B1 security levels. This server is also being used as the base for all OSF RI operating systems technology, including for multicomputers and clusters.

There is still more work to do examining the vm_remap() vs. vm_read_overwrite() versions of the server to determine if the difference in performance between the two systems is inherent in the design or could be solved with a better implementation. The new servers are very robust considering how long they have been running which is undoubtedly due to the increased use of mature OSF/1 integrated kernel code. In addition, the system's semantics are now as close to those of the integrated kernel that they can be, given that the system is now in two parts, the kernel and the server.

The performance data is promising but also implies that there will probably not be any performance revolutions using this architecture, however it is believed that we can exceed the performance of the emulator based servers once certain optimizations have been implemented.

10. Acknowledgments

I would like to thank all the members of the OSF/1 MK team in the OSF Research Institute, Grenoble for their contributions to this work including their ideas and the efforts implementing the final design. In particular I would like to thank Michael Condict and Philippe Bernadat for many passionate design discussions, François Barbou des Places, Éamonn McManus, David George, Christian Bruel and Rod MacDonald for their vital help in implementing the system. I would also like to thank all the OSF Research engineers whose scepticism kept us honest.

Appendix A

Exception Message Interfaces.

```
kern_return_t
catch_exception_raise(
        mach_port_t                     exception_port;
        mach_port_t                     thread;
        mach_port_t                     task;
        exception_type_t                exception_type;
        exception_data_t                code;
        mach_msg_type_number_t          code_count);

kern_return_t
catch_exception_raise_state(
        mach_port_t                     exception_port;
        exception_type_t                exception_type;
        exception_data_t                code;
        mach_msg_type_number_t          code_count;
        thread_state_flavor_t           *flavor;
        thread_state_t                  *in_state;
        mach_msg_type_number_t          in_state_count;
        thread_state_t                  *out_state;
        mach_msg_type_number_t          *out_state_count);

kern_return_t
catch_exception_raise_state_identity(
        mach_port_t                     exception_port;
        mach_port_t                     thread;
        mach_port_t                     task;
        exception_type_t                exception_type;
        exception_data_t                code;
        mach_msg_type_number_t          code_count;
        thread_state_flavor_t           *flavor;
        thread_state_t                  *in_state;
        mach_msg_type_number_t          in_state_count;
        thread_state_t                  *out_state;
        mach_msg_type_number_t          *out_state_count);
```

Appendix B

Exception Port Manipulation Interfaces.

```
kern_return_t
thread_set_exception_ports(
         mach_port_t                    thread;
         exception_mask_t               exception_types;
         mach_port_t                    exception_port;
         exception_behavior_t           behavior;
         thread_state_flavor_t          flavor);

kern_return_t
thread_get_exception_ports(
         mach_port_t                    thread;
         exception_mask_t               exception_types;
         exception_mask_array_t         old_exception_types;
         mach_msg_type_number_t         *old_exception_count;
         mach_port_array_t              old_exception_ports;
         exception_behaviorarray_t      behavior;
         thread_state_flavor_array_t    flavor);

kern_return_t
thread_swap_exception_ports(
         mach_port_t                    thread;
         exception_mask_t               exception_types;
         mach_port_t                    exception_port;
         exception_behavior_t           behavior;
         thread_state_flavor_t          flavor;
         exception_mask_array_t         old_exception_types;
         mach_msg_type_number_t         *old_exception_count;
         mach_port_array_t              old_exception_ports;
         exception_behaviorarray_t      behavior;
         thread_state_flavor_array_t    flavor);
```

Appendix C

Arbitrary Memory Mapping Interface.

```
kern_return_t
vm_remap(
         mach_port_t                    target_task,
         vm_address_t                   *target_addr,
         vm_size_t                      size,
         vm_address_t                   mask,
         boolean_t                      anywhere,
         mach_port_t                    source_task,
         vm_address_t                   source_addr,
         boolean_t                      copy,
         vm_prot_t                      prot,
         vm_prot_t                      *max_protection,
         vm_inherit_t                   inheritance);
```

Redirecting System Calls in Mach 3.0

An alternative to the emulator

Simon Patience

Open Software Foundation
Research Institute
2, Avenue de Vignate
38610 Gières
France
Tel: (+33) 76 63 48 72
E-mail: sp@gr.osf.org

Abstract:

In order to implement system calls, current Mach 3 based operating system personality servers all make use of Mach's system call redirection mechanism together with an emulator co-resident in the applications address space. Experience with OSF/1 MK has shown that this is problematic for OSF/1 system call semantics, system security and integrity. This paper looks at what these problems are and why they occur. It also describes new kernel mechanisms to implement OSF/1 system calls which address these problems and our experiences implementing them. Finally the benefits and shortfalls of the new mechanisms are discussed both in relation to conventional computer architectures and also with respect to new architectures, such as multicomputers and clusters.

1. Introduction

For the past two years the OSF Research Institute at Grenoble, France, has been developing a version of OSF/1 based on Mach 3. The architecture for this system was basically copied, for the most part, from the BSD server from CMU. The model used was that the application programming interfaces were implemented in a single server executing in user space. Access to the server from the application was via an emulator that was resident in a fixed place in the address space of every process. As time passed it was found that the advantages of the emulator also brought many disadvantages, some performance, some semantic and some maintenance. It became clear that, for implementing OSF/1 semantics, the emulator was not the best solution. During the past 9 months, new mechanisms and improvements to existing mechanisms have been developed to be able to better support servers implementing Un*x like semantics.

This paper discusses the motivation for the emulator, why it was inadequate for the needs of OSF/1, and what was used to replace it. In addition, the impact of this solution on other architectures currently being used in OSF RI is considered.

2. Why an emulator?

The motivation for the emulator was originally for binary compatibility. Traditional statically bound BSD binaries used hardware traps to communicate their request for a system call to the kernel. With Mach 3 however, the kernel does not know anything about BSD semantics, or even which operating system personality is running. As a consequence, these traps had to be redirected in some way. In addition, there was the apparent contradiction of style between BSD system calls and Mach's message based communication mechanisms.

The emulator is essentially part of the operating system personality. The emulator is a second program that is loaded into an unused part of the address space of every process. When the calling application does a system call trap, the emulator is entered by a simple redirection back into a predefined part of its address space. It can then perform the translation between a trap based interface directed at the kernel to a message based interface directed at the server. Figure 1 shows the code path of a normal system call using the emulator.

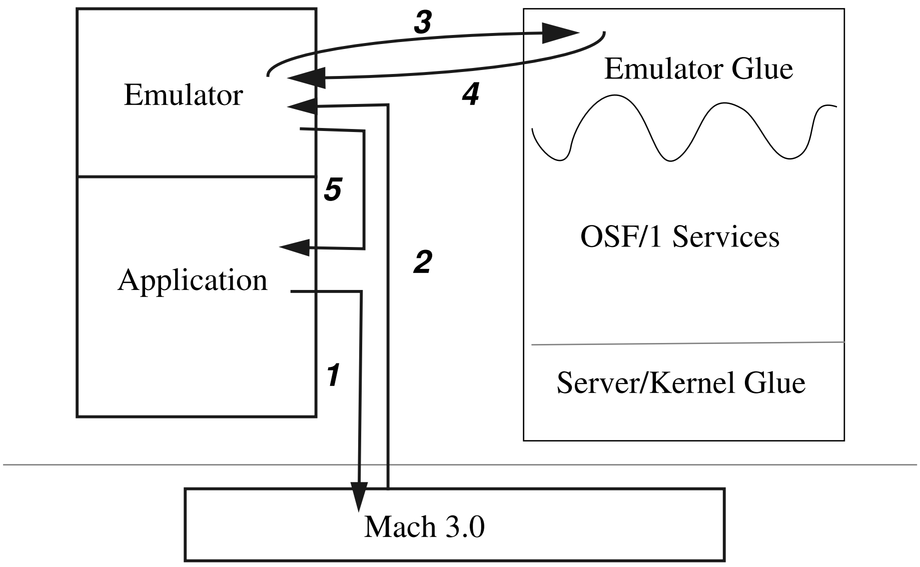

Figure 1: Code Path of an emulated system call

The application makes the system call in the normal way (step 1) by executing a trap instruction into the Mach kernel. The kernel looks up the trap number in a per task redirection table, saves some state on the user stack and returns back to the application (step 2) at the address in the table. The emulator starts to process the system call and normally sends a request Mach IPC message to the server to perform the system call (step 3). Having executed the system call, the server sends a reply message to the emulator (step 4), sending back any resulting data. Finally (step 5), the emulator returns back to the address in the application that the kernel saved on the stack having stored the results of the system call in the appropriate places.

In addition to doing this simple conversion, the emulator soon became the focal point for performance optimizations. Some system calls were able to be entirely implemented in the emulator either due to the emulator caching information or due to introduction of shared data between the emulator and the server. Some other optimizations were made by having an I/O page between the server and the emulator, which eliminated copies into and out of messages.

The other advantage the emulator was seen to have was that in systems which supported shared libraries, the emulator could ultimately replace the system call stubs in libc.

2. Why not an emulator?

The problems introduced by the emulator can be divided into the following categories;

- System security
- System integrity
- System resource management
- Compatibility and maintenance

It should be noted that some of these problems are OSF/1 specific, while others are Un*x like specific and a few are generic to any personality. The root of most of the problems with the emulator is that it is untrusted due to the fact that it is completely unprotected from the application.

2.1 System security

Violations of system security mean that, although the server continues to function normally, the security policy is potentially violated. This means that the operating system personality is fooled into providing services for a process which should not receive them. Using the emulator, all the Mach kernel services are necessarily exposed for direct use by the application. Because Mach services are not mediated by the server, the kernel will perform operations for anyone who presents the appropriate port. The biggest problem areas are task and thread ports which can be passed to other processes via Mach IPC. Using the current emulator based OSF/1 MK or the CMU BSD server, it is relatively easy to become root or force the server to execute system calls for privileged programs which they did not request.

One way of becoming root with the current system would be for a process to modify its emulator code/data such that some innocuous system call will do something to breach security, such as exec() a shell, instead of performing its original service. Then the process exec()s a setuid-root program known to use that system call. Since the emulator is inherited without its text or data being re-initialized, the setuid-root program will use the malicious emulator and security is broken. Although it is possible to close these types of security holes by replacing the task and reloading the emulator for setuid execution, it is at the expense of performance and general increase in complexity.

Removing the emulator does not change these problems in any way but it does allow the possibility of disallowing non-privileged programs from direct access to Mach services, which is what is done. In OSF/1, Mach services are defined for use by system extensions. Consequently in initial versions of OSF/1 MK, a process needs to have the same privilege to use Mach services as it needs to load "kernel" extensions into the server. The use of the services will be re-enabled when the use of Mach services can be brought within the security policy of the operating system personality.

2.2 System integrity

System integrity is breached if a process can get the server to perform incorrectly and at worst, either hang or crash. There are a number of ways in which the emulator exposes the server to attack. OSF/1 MK inherited various mechanisms from the BSD server which cause problems. In addition the direct unmediated use of Mach services causes others.

There are a number of attacks that the server must defend against by using the Mach kernel primitives to manipulate resources that the server believes are under its control. In particular, the problems are processes calling task_terminate() instead of exit and processes calling thread_create() and then these unknown threads making system calls. Processes vm_allocate()ing or vm_deallocate()ing memory causes confusion as the server attempts to manage the processes address space to control the brk area and to flush mapped regions etc.

Processes can also send strange messages to the ports on which the server collects system call requests. These can either be with the correct ID and the wrong contents (for copyin() for example) or with additional out-of-line data that the slowly clogs the servers address/port space.

2.3 System resource management

Attacks of this form can come in two forms. Either the application uses more resources than it is allowed or it forces the server to use resources which could potentially cause denial of service or even to cause the server to crash or panic. Again this is not explicitly enabled by the emulator but is due to the direct use of Mach services which the presence of the emulator requires.

Exposing the external memory manager interfaces means that a malicious program could establish a mapping with an uncooperative pager which could not respond when the server accessed it. This would tie up server threads forcing the server to create more, if it wished to continue to function, until the system resources were exhausted.

The kernel itself has no mechanism for resource control and will blindly satisfying most requests to allocate resources such as threads. ports or vm address space until that resource is exhausted for the whole system.

2.4 Compatibility and maintenance

There are a number of problems with the emulator architecture which are perhaps more specific to OSF/1 than to other personalities as they are caused by some OSF/1 features not normally found in Un*x systems.

OSF/1 allows for kernel loadable modules. One type of subsystem that can be loaded is a module that implements new system calls. New subsystems configure themselves dynamically when loaded, which is normally at boot time. Access to the emulator is set up when loading *init* and only system calls that are known about at this time are redirected to the emulator. This emulator vector is then inherited from task to task. Thus, the system call table is replicated for every process in the system and there is no way to update the vector for every task in the system when new system calls are added.

For almost the entire time that OSF/1 MK was being worked on in Grenoble, there were always outstanding signal bugs. Even after almost two years of development, the signal semantics of OSF/1 MK were not totally compatible with the integrated kernel, and indeed there were still some bugs remaining that would cause the server to hang. This was caused primarily by splitting the signal functionality between the server and the emulator, and then not trusting the emulator to do the job. The signals code in OSF/1 MK deviated from its integrated kernel equivalent more than any other piece of code in the system. It changed significantly in virtually every release and there were still corner case bugs after more than 12 releases. Part of the problem of delivering signals from the server is that, if the application is in the emulator, it is hard to find the real user stack to build the signal frame, as the emulator changes stacks when it is entered. When or if you do find it however, it has data the emulator has saved there and so cannot be used anyway. This is also a problem when trying to dump core.

In order to improve the performance of the emulator, a page of state is shared between the server and the emulator. This page holds various information normally found in the utask structure. This means that structure offsets are different and thus kernel extensions are not binary compatible. It also means that the server cannot trust this information and must check that it is sane before it can be used.

In addition, the kernel-to-application interface is well defined in the integrated kernel, it is copyin() and copyout(). When any functionality of a system call is implemented in the emulator, copyin() is implemented by the emulator knowing, in advance, what is to be copied and sending the data along with the message containing the system call request. Similarly, copyout() is implemented by extending the reply message that is sent back at the end of the system call with the data to be copied and then the emulator does the copy to the correct place in the user's address space. The problem with this is it means that copyin() only works if arranged in advance and any copyout()s are deferred until the end of the system call. As the size of the return message was limited, this also limited the amount of data the server could copyout(). This all means that any kernel extensions expecting, for example, that copyout() followed by copyin() of the same address would see the data written would not see the expected result. As a consequence, OSF/1 integrated kernel extensions are not even source compatible with OSF/1 MK.

Upgrading becomes more expensive due to the amount of code that is changed to support the emulator that has to be repeatedly applied to each new version of OSF/1. This also means that OSF/1 MK is not using OSF/1 IK code for this functionality which increases the chances of the semantics differing between the systems and introducing bugs into OSF/1 MK. The issue of kernel extensions becomes more important as the server is upgraded to OSF/1 1.2 and beyond as virtually all subsystems (NFS, System V FS, LVM, network protocols) are dynamically loaded during booting and it is unreasonable and expensive to expect people to re-code their kernel extensions for the sake of changes introduced by the emulator.

Even in itself, the emulator is a maintenance nightmare for OSF/1 MK. It is a complex piece of code involving stack switching code, port and memory management, code for signal delivery (which incidentally is duplicated in the server) and address fault recovery. There is communication between the emulator and server via shared memory, which the server cannot trust and must therefore defend against, thus forcing the complexity into the server.

Finally, there were a number of little problems. Errant programs would fail in unexpected ways if they scribbled over the emulator. Debuggers do not understand the emulator and so follow the trap redirection into a part of the address space that the programmer may not have realized (and probably does not want to know) was mapped. In OSF/1 AD, the emulator even has extra threads which are not part of the application to further confuse and add failure modes. The mapped file implementation also involved passing tokens between emulators to indicate ownership which might not be given up, so a complex time-out and revocation scheme was invented. There seemed to be excessive data copying from the servers buffers into the shared I/O page and then into the user's buffer for read()s and write()s. The stack allocation and switching at the start of every system call seemed like another expense that was not needed. Ultimately it became too much and it was felt as though more effort was being put into maintaining the emulator than moving the system forward functionally.

3. What instead?

The solution was to solve two problems: The communication of a system call request to the server and correcting the implementation of copyin() and copyout() in the server.

The request from the application had to reach the server in an efficient form and as this is Mach, this should be a message. The application request for OSF/1 was in the form of a trap instruction, or equivalent, in the system call stub in libc. This meant that it had to be a kernel mechanism. The obvious choice was to send an exception to the appropriate exception port. There were numerous problems with this solution however.

The exception message Mach currently sends contains send rights to the task and thread ports of the thread that generated the exception. Sending port rights in messages is relatively slow compared to sending data alone and for system call delivery, speed is of the essence. In addition, it is certain that a system call will need to modify the state of the calling thread. With the current message format, this means the server having to call thread_get_state() and thread_set_state() for every system call. The added complication of debuggers interposing the exception port in order to intercept exception messages also means that these messages could not be sent on the regular exception port. So what with slow, port laden messages being sent to the wrong port and two additional Mach system calls being needed for every OSF/1 system call, it was clear that this would not work. However the general idea was right and this led to the exception mechanism being generalized.

In order to solve the limitations of the emulator based server implementation of copyin() and copyout(), the functions had to be made semantically identical to the integrated kernel. With the current Mach interfaces, the only available way of reading or writing to another task's address space is to use vm_read() or vm_write() respectively. Neither of these calls in their current forms are capable of doing the job for copyin() and copyout().

The problem of vm_write() is simply that it only writes page sized and aligned quantities. This makes it impossible to atomically copyout() data that is not page aligned in both size and address in both the server and the target process. For example, to copy 40 bytes would involve the server reading the page in which the addresses it wants to modify are, modify the 40 bytes and then write the entire page back. If that page was shared by multiple threads, either in the same task or in different tasks, then there is no guarantee that one of these threads will not touch the part of the page that the server does not wish to change and if it does, the server's page write would overwrite these modifications.

The problem for vm_read() is simply that it returns the data out-of-line. (It also page aligned in both size and address, but that is not a semantic problem). As a consequence of this, the data is never where it needs to be and so it must be copied. The original data returned out-of-line must then vm_deallocate()'d. copyin() is not a function that can tolerate making two Mach system calls as it is called too frequently.

The result of making these two changes, adding the new exception mechanism and fixing copyin()/copyout(), is that the architecture looks remarkably like that of the integrated kernel, with exception that the OSF/1 services are in user space rather than being bound in with the Mach kernel. The interface between the application and operating system personality, OSF/1 in our case, is back to being trap from the application to the personality and copyin()/copyout() from the personality to the application.

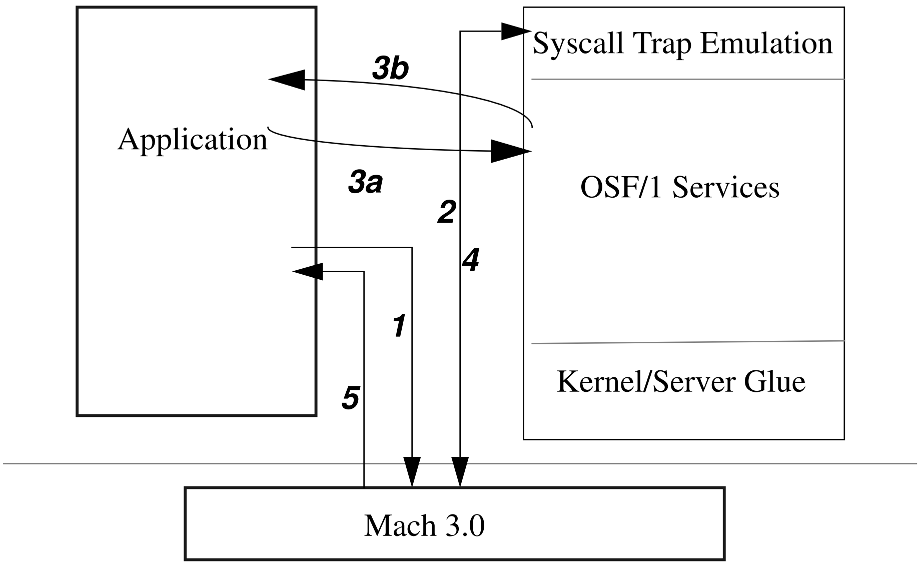

Figure 3: Code path for a system call without an emulator

The application makes the system call request in the normal way (step 1) by executing a trap instruction into the kernel. The kernel generates an exception message which it sends to a port that the server has set up especially for handling system call requests (step 2). The server then processes the system call in the same way as in the integrated kernel, calling copyin() (step 3a) or copyout() (step 3b) whenever it needs to get or return data to the application. Finally the server returns any modified register state to the kernel (step 4) in the reply to the exception message, and the kernel resumes the thread having restored it (step 5).

4. A generalized exception mechanism

Having initially tried having the kernel send specialized messages when traps occurred, it was found that the code used in the kernel was remarkably similar to that to send exceptions. The motivation for considering a separate interface was simply performance, i.e. to avoid sending ports. From the implementation however, it was clear that what was trying to be achieved was to simply raise an exception on the occurrence of a trap instruction.

The current Mach exception mechanism treats all exceptions alike, in that the kernel generates a message and sends it to a thread's exception port, if it has one, or the task's exception port, if not. The message that is sent is the same in all cases and it is intended simply to identify the task and thread that caused the exception. The new mechanism changes this in two fundamental ways.

Firstly, to obviate the need for a server to use thread_get_state() and thread_set_state() when exceptions occur, new versions of the exception message were defined in which the state is sent and the modified state returned with the reply, which the kernel will set before resuming the thread. In addition, one of the new interfaces does not send the task and thread port of the thread causing the exception, making the assumption that the thread can be determined by the port that the incoming message appears on. The MiG exc_server() routine calls either catch_exception_raise(), catch_exception_raise_state() or catch_exception_raise_state_identity(), depending on the ID of the exception message the sent by the kernel. The new exception message interfaces are listed in Appendix A.

Clearly, the state needed for dealing with system call exception messages would not necessarily be the same state as that for dealing with arithmetic or address faults. Consequently, a way of associating what flavor state is required to be sent in response to a given exception type was introduced.

Secondly, to avoid confusion for debuggers that wish to interpose exception ports for the applications that they are controlling, you would not want to send system call exceptions, which the debugger is not interested in and probably couldn't handle, to the same port as arithmetic faults, which the debugger is interested in and can handle. Due to this, and since it is likely that some servers will want to use different threads to handle the different types of exception, a new interface makes it possible to associate different ports to different exception types for a given thread/task. The behavior specified determines the type of message the kernel will send and the flavor indicates the flavor of thread state the kernel sends with that message. Obviously, if the behavior is EXC_DEFAULT indicating that catch_exception_raise() is to be used, then the flavor is ignored. The new exception port manipulation interfaces are listed in Appendix B.

The types of exception supported are those returned currently by catch_exception_raise() in the exception parameter, plus the additional types EXC_TYPE_SYSCALL and EXC_MACH_SYSCALL. However it is expected that certain architectures will wish to define more.

In addition to the current thread state flavors, a new flavor was added, THREAD_STATE_SYSCALL. The definition the contents of this state is machine dependent but the intention is that it contains a small subset of registers adequate for handling most system calls for most personalities. For the i386, this register set is currently 5 registers (as opposed to 17 for the full set).

Note that

```
task_set_special_port(task, THREAD_EXCEPTION_PORT, new_port);
```

is equivalent to

```
task_set_exception_ports(task, EXC_MASK_ALL & ~EXC_MASK_SYSCALL,
                                           new_port, EXCEPTION_DEFAULT, 0);
```

The same will apply to the other task/thread get/set exception calls. However a backwards compatibility problem has been introduced as the format of the catch_exception_raise() interface has changed (for a completely unrelated reason). So, in our kernels, setting the exception ports with task_set_special_port() will generate the old catch_exception_raise() format message but doing the equivalent operation with task_set_exception_ports() described above, the new catch_exception_raise() format is used, thus it was possible to achieve both source and binary compatibility.

Delivery of exceptions in the kernel uses exactly the same algorithm. When an exception occurs, the kernel tries to deliver the exception message to the thread exception port for that exception type. If that does not exist or returns a value other than KERN_SUCCESS then the task exception port for that exception type is tried. If the exception is not handled after both possibilities have been tried, the kernel terminates the task, as before. Note it is perfectly possible and reasonable that the task and thread ports for a given exception type have different behaviors and flavors set.

5. Solving the copyin()/copyout() problem

Two solutions to this problem were explored. One was relatively conventional in Mach terms and the other rather controversial. The first was to 'fix' vm_read() and vm_write() such that they could be used in such performance critical code as copyin() and copyout(). The second involved the server mapping arbitrary addresses in the target process.

5.1 Using vm_read() and vm_write()

The change to vm_write() was simply to allow the specification of addresses in both the source and the target tasks that are not page aligned and to allow quantities of data that are not multiples of a page in size. In fact a large number of people had seen this as a bug in the specification of vm_write() (there was even a bug report against OSF/1 1.1 for this) so this seemed like an uncontroversial change. It had the added advantage that it did not effect existing callers of vm_write(). Clearly if unaligned data was specified, there was no possibility for the kernel to perform any VM manipulations which would avoid physical copying of data; however, the implementation was careful to ensure that if correctly aligned data was specified then the appropriate VM magic would occur.

The change for vm_read() was slightly more awkward. The problem was caused by the fact that vm_read() returns the data out-of-line. For copyin() semantics this has two ramifications; the first being that the data is not where you want it thus causing a copy, and the second being that the returned data needs to be vm_deallocate()'d after use thus incurring the extra cost of another Mach service. The solution was to introduce a new class of Mach interface; one which returned out-of-line data in a buffer that the receiver already had mapped.

This was done by introducing a new interface, mach_msg_overwrite(). The semantics are exactly the same as mach_msg() with out-of-line data with the exception of final delivery to the receiver. The receiver specifies an already mapped address and range in which the kernel is to put any out-of-line data that the message may contain. If the specified buffer is too small, then the call to mach_msg_overwrite() fails. Once this call existed, it was simply a case of changing MiG to use it. The resulting interface was vm_read_overwrite() which is semantically identical to vm_read() with the exception that the call provides the buffer. Using this interface, there was no need for the bcopy() (as the kernel did it or the equivalent) but more importantly, the vm_deallocate() was removed from the copyin() path.

6. Using vm_remap()

There was still concern that one Mach system call per copyin() or copyout() was still one too many, especially considering that performance of the system would be compared against the integrated kernel, whose implementation of these functions normally amounted to the equivalent of a bcopy(). To solve this problem, a new interface was added to Mach which seemed to break one of the rules of Mach interfaces, that of being location transparent. The interface, called vm_remap(), maps arbitrary regions of one task's address space into the caller's address space.

The main issue is that the interface is not location transparent in that if the target task was not on the same host as the caller, then the call would fail. The reason that this will fail is not an inherent problem with the interface as it is specified, but it is more due to the fact that most pagers do not know how to manage multiple kernels (the netmemoryserver being the obvious exception). There is the same problem with default_pager_object_create being used by two clients of vm_map on different hosts. Also, there is no mechanism to allow the kernel managing the source task to inform the kernel that is managing the target task of the memory objects that represent the address range in the source task. This problems could probably be fixed but it would involve some inter-kernel communication. Vm_remap() will work on multiple kernels within a NORMA domain even across different nodes as the NORMA XMM performs basically the same job as the netmemoryserver within the NORMA domain and the NORMA kernels cooperate with task address space management, as this is needed to support norma_task_create().

The call to vm_remap(), listed in Appendix C, is similar to vm_map() except that instead of a memory_object port being supplied to the kernel, a task port and a virtual address range is specified. The kernel then maps the underlying objects of the source address range into the target task's address space. The implementation of vm_remap() simply replicates the kernel data structures in exactly the same way as would have happened if one of the tasks was specified as the parent task to task_create() and this portion of memory was marked as VM_INHERIT_SHARE. The main reason for this choice was to avoid any new kernel data structures or new vm relationships which would increase the complexity of the VM generally.

The disadvantage of mapping the underlying objects instead of somehow sharing address maps, is that deallocating the addresses in the application does not invalidate the mapping in the server. Consequently the server has to be very careful when it deallocates memory on behalf of the application to ensure that the equivalent server mapping is deallocated also. Even worse is that if an application used Mach primitives directly to deallocate this address range, the server would never get to know about the change and any subsequent system call specifying this address range would use the now stale mapping in the server. However to really share address ranges would involve a considerable change to the kernel management of address maps which was not consider either desirable or necessary.

Two different caching schemes were implemented in the server to take advantage of existing mappings. One was a system wide cache and the other was a task local cache. Both were prototypes and need further work in one area or another.

The system cache consisted of a hash table of mappings which were indexed by application virtual address and PID. A LRU (Least Recently Used) list of mappings is maintained for garbage collection when the hash table gets full. This had the advantage that on a quiet system, a single application could use as much of the cache as it liked but on a busy system, an application that used buffers distributed throughout its address space would use an excessive number of cache entries and thus cause better behaved applications to garbage collect the cache to find a free slot.

The second caching scheme was to maintain a simple linked list of mappings per task. Limits on the number of mappings and the amount of server virtual memory were applied to the task to prevent a task from using up server resources excessively. Two hints were maintained into the linked list based on address to speed the lookup.

Both caching schemes seemed to perform similarly with cache hit rates in excess of 90%. The system wide cache was marginally slower but this was caused by two deficiencies, one in each cache implementation. The system wide cache did not implement hints (which in the per task scheme had a very high hit rate, thus obviating the cache lookup completely) and the per task cache was not thread safe so avoiding lock overhead. I doubt if there would be a significant difference in performance if these two problems were fixed.

7. The resulting system

As a result of using per thread exception ports, a large amount of restructuring of internal data structures occurred in the final implementation. In fact, to simplify other things, the original OSF/1 kernel internal data structures were reintroduced, that is the Mach 2.5 task and thread structures. The system was very stable remarkably quickly, passing VSX and VSE to the same level as OSF/1 MK V4.1 within a few weeks of getting to multi-user mode and ran AIMIII to 250 users. This is compared to the emulator based server which took much of a year to get to the same level of robustness and functional completeness. The main reason for this was the significant increase of reuse of mature OSF/1 Integrated Kernel code rather than writing new code to implement the same functionality, as was done for the emulator. The code reuse was so great that even the integrated kernel's function syscall(), which handles system call setup, could be virtually entirely reused, with 15 extra lines of preamble, 2 postamble and a name change to catch_exception_raise_state().

As the internal environment became indistinguishable from the integrated OSF/1 system, virtually all the subsystems, e.g. ufs, vfs, net, netinet, etc., were reused totally unchanged. This in turn has made the upgrade of the 1.0.4 based version of the server to be based on OSF/1 1.2 to be achievable in a far shorter time.

The table below shows performance ratios of various versions of the no-emulator server compared to OSF/1 MK V4.1, which is the latest emulator based version of OSF/1 MK. The first 6 benchmarks are micro-benchmarks consisting of a 10 iterations of 1000 loop tests of either one, or a small number of system calls. The choice of system calls was based on the number of copyin()/copyout() operations they performed rather than frequency of use and thus are not representative of the overall performance that could be expected of the system, but more indicate the behavior of the mechanisms that are the subject of this paper. The AIMIII tests at 24 and 32 users give a better idea of overall system behavior. All the tests were performed using the same kernel which was an

	OSF/1 MK 4.1	vm_remap (system cache)	vm_remap (task cache)	vm_read_overwrite
getuid	1.00	1.04	1.04	1.05
read/lseek	1.00	0.73	0.71	1.06
File copy	1.00	0.81	0.88	0.88
uname	1.00	1.00	0.94	2.29
exec	1.00	1.06	1.04	1.37
sigaction	1.00	12.99	11.95	23.74
AIMIII 24 Users	1.00	1.04	1.01	1.42
AIMIII 32 Users	1.00	1.03	1.00	1.48

Performance Ratios of OSF/1 MK V4.1 using the emulator Vs. No-emulator servers

extended NMK13.3 with support for the no-emulator servers.

From the performance perspective, the new architecture was about performance neutral with OSF/1 MK V4.1, the latest version which used the emulator. This is encouraging as V4.1 is a product of many performance optimizations implemented over a year whereas there has been no effort put into optimizing any part of the no-emulator server yet. As there are known performance problems in the no-emulator server, this augurs well for the final performance of this server.

The vm_read_overwrite()/vm_write() still seems to have some performance problems. Admittedly there are some known kernel optimizations that can be made to these calls, but it is still clear that the server can afford the Mach call for every copyin() and copyout() operation. The high cache hit rate seems to be helping considerably for the vm_remap() versions of the server but neither caching strategy seems to be a clear winner judged simply on performance. Also, as neither caching scheme was fully implemented, it seems premature to make a judgement between them until they have been.

8. Suitability for other architectures

The main architectures that OSF has to consider, beyond traditional uniprocessors and symmetric shared memory multiprocessors, are multicomputers, otherwise known as NORMA (NO Remote Memory Access) machines, and Clusters of cooperating machines sharing a single system image. Both systems have a set of similar problems when it comes to distributing services, even though the details of the solutions may differ slightly. It was important that any change of server/kernel architecture was usable by systems such as these.

One of the main differences between this architecture and that using the emulator is that, as with the integrated kernel, the applications system call request enters the TCB (Trusted Computing Base) immediately (the emulator is not a trusted piece of code). As the only way of getting to the system call service point is via a single Mach port, the system call exception port, this implies that all system calls are, at least initially, handled by the same server. At first glance, this may not look as if it matches well with the idea of distributed operating system services, however having a protected local entity has many benefits that the emulator solutions could not properly provide.

On each node of these systems there can be a local agent, which acts as the trusted router of system call requests for all the tasks on this node. This allows the agent to play its part in distributed database management (for things like credentials, process management, etc.) like the untrusted emulator never could. In addition, the local agent can also play the part of a trusted cache manager by performing read ahead on file data not only to bring the data off the disk, but also to bring it local to the task so that a subsequent read would not have to leave the node to be satisfied. The agent will also at least have to participate in, if not initiate, process migration between nodes. None of this could be satisfied if the agent was not a trusted piece of code.

This does not mean however that an application may never communicate directly with a remote server. Once the problems of exposing Mach interfaces to applications (briefly described in Section 2) have been solved, a new libc could be implemented which used the services of both a local agent and remote servers, using the agent as the distributor of access means (i.e. ports) to the remote services.

Both the next version of OSF/1 AD (for multicomputers) and the version being designed for clusters are being based on the no-emulator technology.

9. Conclusion

The work is still not fully implemented in a final system. The 1.0.4 prototype was missing a number of key optimizations and small pieces of functionality were not implemented as they were seen as not to be relevant to this proof of concept. However, it is being used as the basis for the OSF/1 1.2 based version of the server, which will be the first version of OSF/1 MK that can be seriously considered for productization and is the first version that is expected to be able to attain C2 and B1 security levels. This server is also being used as the base for all OSF RI operating systems technology, including for multicomputers and clusters.

There is still more work to do examining the vm_remap() vs. vm_read_overwrite() versions of the server to determine if the difference in performance between the two systems is inherent in the design or could be solved with a better implementation. The new servers are very robust considering how long they have been running which is undoubtedly due to the increased use of mature OSF/1 integrated kernel code. In addition, the system's semantics are now as close to those of the integrated kernel that they can be, given that the system is now in two parts, the kernel and the server.

The performance data is promising but also implies that there will probably not be any performance revolutions using this architecture, however it is believed that we can exceed the performance of the emulator based servers once certain optimizations have been implemented.

10. Acknowledgments

I would like to thank all the members of the OSF/1 MK team in the OSF Research Institute, Grenoble for their contributions to this work including their ideas and the efforts implementing the final design. In particular I would like to thank Michael Condict and Philippe Bernadat for many passionate design discussions, François Barbou des Places, Éamonn McManus, David George, Christian Bruel and Rod MacDonald for their vital help in implementing the system. I would also like to thank all the OSF Research engineers whose scepticism kept us honest.

Appendix A

Exception Message Interfaces.

```
kern_return_t
catch_exception_raise(
          mach_port_t                    exception_port;
          mach_port_t                    thread;
          mach_port_t                    task;
          exception_type_t               exception_type;
          exception_data_t               code;
          mach_msg_type_number_t         code_count);

kern_return_t
catch_exception_raise_state(
          mach_port_t                    exception_port;
          exception_type_t               exception_type;
          exception_data_t               code;
          mach_msg_type_number_t         code_count;
          thread_state_flavor_t          *flavor;
          thread_state_t                 *in_state;
          mach_msg_type_number_t         in_state_count;
          thread_state_t                 *out_state;
          mach_msg_type_number_t         *out_state_count);

kern_return_t
catch_exception_raise_state_identity(
          mach_port_t                    exception_port;
          mach_port_t                    thread;
          mach_port_t                    task;
          exception_type_t               exception_type;
          exception_data_t               code;
          mach_msg_type_number_t         code_count;
          thread_state_flavor_t          *flavor;
          thread_state_t                 *in_state;
          mach_msg_type_number_t         in_state_count;
          thread_state_t                 *out_state;
          mach_msg_type_number_t         *out_state_count);
```

Appendix B

Exception Port Manipulation Interfaces.

```
kern_return_t
thread_set_exception_ports(
         mach_port_t                  thread;
         exception_mask_t             exception_types;
         mach_port_t                  exception_port;
         exception_behavior_t         behavior;
         thread_state_flavor_t        flavor);

kern_return_t
thread_get_exception_ports(
         mach_port_t                  thread;
         exception_mask_t             exception_types;
         exception_mask_array_t       old_exception_types;
         mach_msg_type_number_t       *old_exception_count;
         mach_port_array_t            old_exception_ports;
         exception_behaviorarray_t    behavior;
         thread_state_flavor_array_t  flavor);

kern_return_t
thread_swap_exception_ports(
         mach_port_t                  thread;
         exception_mask_t             exception_types;
         mach_port_t                  exception_port;
         exception_behavior_t         behavior;
         thread_state_flavor_t        flavor;
         exception_mask_array_t       old_exception_types;
         mach_msg_type_number_t       *old_exception_count;
         mach_port_array_t            old_exception_ports;
         exception_behaviorarray_t    behavior;
         thread_state_flavor_array_t  flavor);
```

Appendix C

Arbitrary Memory Mapping Interface.

```
kern_return_t
vm_remap(
         mach_port_t                    target_task,
         vm_address_t                   *target_addr,
         vm_size_t                      size,
         vm_address_t                   mask,
         boolean_t                      anywhere,
         mach_port_t                    source_task,
         vm_address_t                   source_addr,
         boolean_t                      copy,
         vm_prot_t                      prot,
         vm_prot_t                      *max_protection,
         vm_inherit_t                   inheritance);
```