The following paper was originally published in the
Proceedings of the 3rd Symposium on Operating Systems Design and Implementation
New Orleans, Louisiana, February, 1999

Self-Paging in the Nemesis Operating System

Steven M. Hand
University of Cambridge Computer Laboratory

Self-Paging in the Nemesis Operating System

Steven M. Hand
University of Cambridge Computer Laboratory
New Museums Site, Pembroke St.,
Cambridge CB2 3QG, ENGLAND
Steven.Hand@cl.cam.ac.uk

Abstract

In contemporary operating systems, continuous media (CM) applications are sensitive to the behaviour of other tasks in the system. This is due to contention in the kernel (or in servers) between these applications. To properly support CM tasks, we require "Quality of Service Firewalling" between different applications.

This paper presents a memory management system supporting Quality of Service (QoS) within the *Nemesis* operating system. It combines application-level paging techniques with isolation, exposure and responsibility in a manner we call *self-paging*. This enables rich virtual memory usage alongside (or even within) continuous media applications.

1 Introduction

Researchers now recognise the importance of providing support for continuous media applications within operating systems. This is evinced by the *Nemesis* [1, 2, 3] and *Rialto* [4, 5, 6] operating systems and, more recently, work on the *Scout* [7] operating system and the SMART scheduler [8]. Meanwhile there has been continued interest in the area of memory management, with a particular focus on *extensibility* [9, 10, 11].

While this work is valid, it is insufficient:

- Work on continuous media support in operating systems tends to focus on CPU scheduling only.
  The area of memory management is either totally ignored (Scout, SMART) or considered in practice to be a protection mechanism (Rialto). In fact, the implementation of the Rialto virtual memory system described in [12] explicitly excludes paging since it "*introduces unpredictable latencies*".
- Work on memory management does not support (or try to support) any concept of Quality of Service.
  While support for extensibility is a laudable goal, the behaviour of user-level pagers or application-provided code is hardly any more predictable or isolated than kernel-level implementations. The "unpredictable latencies" remain.

This paper presents a scheme whereby each application is responsible for all of its own paging (and other virtual memory activities). By providing applications with guarantees for physical memory and disk bandwidth, it is possible to isolate time-sensitive applications from the behaviour of others.

2 Quality of Service in Operating Systems

In recent years, the application mix on general purpose computers has shifted to include "multimedia" applications. Of particular interest are *continuous media* (CM) applications — those which handle audio and/or video — since the presentation (or processing) of the information must be done in a timely manner. Common difficulties encountered include ensuring low latency (especially for real-time data) and minimising *jitter* (viz. the variance in delay).

Clearly not all of today's applications have these temporal constraints. More traditional tasks such as formatting a document, compiling a program, or sending e-mail are unlikely to be banished by emerging continuous media applications. Hence there is a requirement for *multi-service* operating systems which can support both types of application simultaneously.

Unfortunately, most current operating systems conspicuously fail to support this mix of CM and non-CM applications:

- CPU scheduling is usually implemented via some form of priority scheme, which specifies *who* but not *when* or *how much*. This is unfortunate since many continuous media applications do not require a large fraction of the CPU resource (i.e. they are not necessarily more *important* than other applications), but they do need to be scheduled in a *timely* fashion.

- Other resources on the data path, such as the disk or network, are generally not explicitly scheduled at all. Instead, the proportion of each resource granted to an application results from a complex set of unpredictable interactions between the kernel (or user-level servers) and the CPU scheduler.
- The OS performs a large number of (potentially) time-critical tasks on behalf of applications. The performance of any particular application is hence heavily dependent on the execution of other supposedly "independent" applications. A greedy, buggy or even pathological application can effect the degradation of all other tasks in the system.

This means that while most systems can support CM applications in the case of resource over-provisioning, they tend to exhibit poor behaviour when contention is introduced.

A number of operating systems researchers are now attempting to provide support for continuous media applications. The Rialto system, for example, hopes to provide modular real-time resource management [4] by means of arbitrarily composable *resource interfaces* collectively managed by a *resource planner*. A novel real-time CPU scheduler has been presented in [5, 6], while an implementation of a simple virtual memory system for set-top boxes is described in [12].

The Scout operating system uses the *path* abstraction to ensure that continuous media streams can be processed in a timely manner. It is motivated by multimedia network streams, and as such targets itself at sources (media servers) and sinks (set-top boxes) of such traffic. Like Rialto, the area of virtual memory management is not considered a high-priority; instead there is a rudimentary memory management system which focuses upon buffer management and does not support paging.

Most other research addresses the problem of Quality of Service within a specific domain only. This has lead to the recent interest in soft real-time scheduling [13, 8, 14, 15] of the CPU and other resources. The work has yet to be widely applied to multiple resources, or to the area of memory management.

3 Extensible Memory Management

Memory management systems have a not undeserved reputation for being complex. One successful method of simplifying the implementation has been the μ-kernel approach: move some or all of the memory management system out of the kernel into "user-space". This mechanism, pioneered by work on Mach [16] is still prevalent in many modern μ-kernels such as V++ [17], Spring [18] and L4 [19].

Even operating systems which eschew the μ-kernel approach still view the extensibility of the memory management system as important:

- the *SPIN* operating system provides for user-level extension of the memory management code via the registration of an event handler for memory management. events [10].
- the VINO operating system [20, pp 1–6] enables applications to override some or all operations within *MemoryResource* objects, to specialise behaviour.
- the V++ Cache Kernel allows "application kernels" to cache address-space objects and to subsequently handle memory faults on these [21].
- the Aegis experimental exokernel enables "library operating systems" to provide their own page-table structures and TLB miss handlers [9].

This is not surprising. Many tasks are ill-served by default operating system abstractions and policies, including database management (DBMS) [22], garbage collection [23] and multi-media applications [24]. Furthermore, certain optimisations are possible when application-specific knowledge can be brought to bear, including improved page replacement and prefetching [17], better buffer cache management [25], and light-weight signal handling [26]. All of these may be realised by providing user-level control over some of the virtual memory system.

Unfortunately, none of the above-mentioned operating systems provide QoS in their memory management:

- No Isolation: applications which fault repeatedly will still degrade the overall system performance. In particular, they will adversely affect the operation of other applications.
  In μ-kernel systems, for example, a single external pager may be shared among an arbitrary number of processes, but there is no scheduling regarding fault resolution. This indirect contention has been referred to as *QoS crosstalk* [2]. Other extensible systems allow the application to specify, for example, the page replacement *policy*, but similarly fail to arbitrate between multiple faulting applications.
- Insufficient Exposure: most of the above operating systems[1] abstract away from the underlying hardware; memory faults are presented as some abstract form of exception and memory translation as an array of virtual to physical mappings.
  Actual hardware features such as multiple TLB page sizes, additional protection bits, address space numbers, physical address encoding, or cache behaviour tend to be lost as a result of this abstraction.

[1]A notable exception is the Aegis exokernel, which endeavours to expose as much as possible to the application.

- No Responsibility: while the many of the above operating systems *allow* applications some form of "extensibility" for performance or other reasons, they do not by any means *enforce* its use. Indeed, they provide a "default" or "system" pager to deal with satisfying faults in the general case. The use of this means that most applications fail to pay for their own faults; instead the system pager spends its time and resources processing them.

What is required is a system whereby applications benefit from the ability to control their own memory management, but do not gain at the expense of others.

4 Nemesis

The Nemesis operating system has been designed and implemented at the University of Cambridge Computer Laboratory in recent years. Nemesis is a *multi-service* operating system — that is, it strives to support a mix of conventional and time-sensitive applications. One important problem it addresses is that of preventing *QoS crosstalk*. This can occur when the operating system kernel (or a shared server) performs a significant amount of work on behalf of a number of applications. For example, an application which plays a motion-JPEG video from disk should not be adversely affected by a compilation started in the background.

One key way in which Nemesis supports this isolation is by having applications execute as many of their own tasks as possible. This is achieved by placing much traditional operating system functionality into user-space modules, resulting in a *vertically integrated* system (as shown in Figure 1). This vertical structure is similar to that of the Cache Kernel [21] and the Aegis Exokernel [27], although the motivation is different.

The user-space part of the operating system is comprised of a number of distinct *modules*, each of which exports one or more strongly-typed *interfaces*. An interface definition language called *MIDDL* is used to specify the types, exceptions and procedures of an interface, and a run-time typesystem allows the narrowing of types and the marshaling of parameters for non-local procedure invocations.

A name-space scheme (based on Plan-9 contexts) allows implementations of interfaces to be published and applications to pick and choose between them. This may be termed "plug and play extensibility"; we note that it is implemented *above* the protection boundary.

Given that applications are responsible for executing traditional operating system functions themselves, the must be sufficiently empowered to perform them. Nemesis handles this by providing explicit low-level resource guarantees or reservations to applications. This is not limited simply to the CPU: all resources — including disks [14], network interfaces [28] and physical memory — are treated in the same way. Hence any given application has a set of guarantees for all the resources it requires. Other applications cannot interfere.

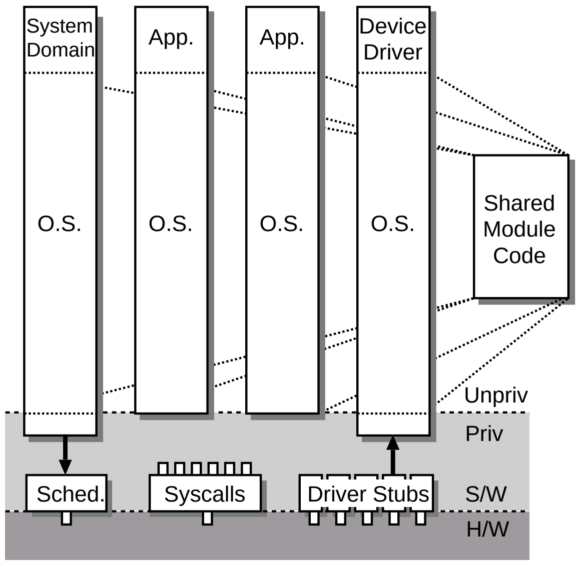


Figure 1: Vertical Structuring in Nemesis

5 Self-Paging

Self-paging provides a simple solution to memory system crosstalk: *require every application to deal with all its own memory faults using its own concrete resources*. All paging operations are removed from the kernel; instead the kernel is simply responsible for dispatching fault notifications.

More specifically, self-paging involves three principles:

1. *Control*: resource access is multiplexed in both space and time. Resources are guaranteed over medium-term time-scales.
2. *Power*: interfaces are sufficiently expressive to allow applications the flexibility they require. High-level abstractions are not imposed.
3. *Responsibility*: each application is directly responsible for carrying out its own virtual memory operations. The system does not provide a "safety net".

The idea of performing virtual memory tasks at application-level may at first sound similar to the ideas pioneered in Mach [16] and subsequently used in μ-kernel systems such as L4 [19]. However while μ-kernel systems *allow* the use of one or more external (i.e. non-kernel) pagers in order to provide extensibility and to simplify the kernel, several applications will typically still share an external pager, and hence the problem of QoS crosstalk remains.

In Nemesis we *require* that every application is *self-paging*. It *must* deal with any faults that it incurs. This, along with the use of the single address space and widespread sharing of text, ensures that the execution of each domain[2] is completely independent to that of other domains save when interaction is desired.

The difference between μ-kernel approaches and Nemesis' is illustrated in Figure 2.

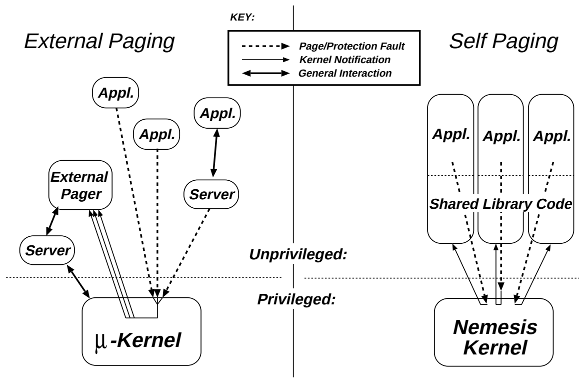


Figure 2: External Paging versus Self Paging

The left-hand side on the figure shows the μ-kernel approach, with an external pager. Three applications are shown, with two of them causing page faults (or, more generally, memory faults). The third application has a server performing work on its behalf, and this server is also causing a memory fault. The kernel notifies the external pager and it must then deal with the three faults.

This causes two problems:

1. Firstly, the process which caused the fault does not use any of its own resources (in particular, CPU time) in order to satisfy the fault. There is no sensible way in which the external pager (or the μ-kernel itself) can account for this. A process which faults repeatedly thus degrades the overall system performance but bears only a fraction of the cost.
2. Secondly, multiplexing happens in the server — i.e. the external pager needs some way to decide how to 'schedule' the handling of the faults. However it will generally not be aware of any absolute (or even relative) timeliness constraints on the faulting clients. A first-come first-served approach is probably the best it can do.

On the right-hand side we once again have three applications, but no servers. Each application is causing a memory fault of some sort, which is trapped by the kernel. However rather than sending a notification to some external pager, the kernel simply notifies the faulting domain. Each domain will itself be responsible for handling the fault. Furthermore, the latency with which the fault will be resolved (assuming it is resolvable) is dependent on the guarantees held by that domain.

[2]A *domain* in Nemesis is the analog of a process or task.

Rather closer to the self-paging ideal are "vertically structured" systems such as the Aegis & Xok exokernels [9, 29]. Like Nemesis, these systems dispatch memory faults to user-space and expect unprivileged library operating system code to handle them. In addition, exokernels expose sufficient low-level detail to allow applications access to hardware-specific resources.

However exokernels do not fully cope with the aspect of *control*: resources are multiplexed in space (i.e. there is protection), but not in time. For example, the Xok exokernel allows library filing systems to download untrusted metadata translation functions. Using these in a novel way, the exokernel can protect disk blocks without understanding file systems [29]. Yet there is no consideration given to partitioning access in terms of *time*: library filing systems are not guaranteed a proportion of disk bandwidth.

A second problem arises with crosstalk within the exokernel itself. Various device drivers coexist within the kernel execution environment and hence an application (or library operating system) which is paging heavily will impact others who are using orthogonal resources such as the network. This problem is most readily averted by pushing device driver functionality outside the kernel, as is done with μ-kernel architectures.

6 System Design

A general overview of the virtual memory architecture is shown in Figure 3. This is necessarily simplified, but does illustrate the basic abstractions and their relationships.

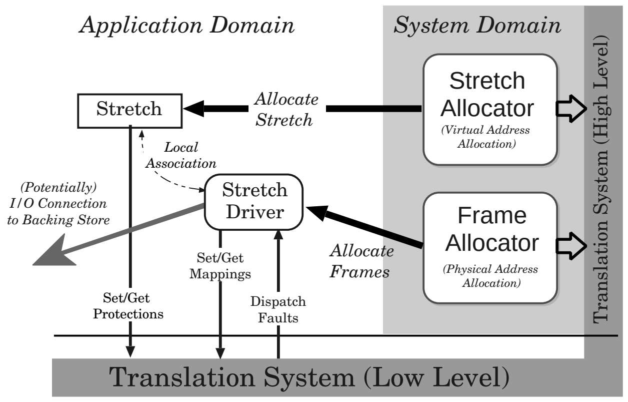


Figure 3: VM System Architecture

The basic virtual memory abstractions shown are the *stretch* and the *stretch driver*. A stretch merely represents a range of virtual addresses with a certain accessibility. It does not own — nor is it guaranteed — any physical resources. A stretch driver is responsible for providing the backing for the stretch; more generally, a stretch driver is responsible for dealing with any faults on a stretch. Hence it is only via its association with a stretch driver that it becomes possible to talk meaningfully about the "contents" of a stretch. Stretch drivers are unprivileged, application-level objects, instantiated by user-level creator modules and making use only of the resources owned by the application.

The stretch driver is shown as potentially having a connection to a backing store. This is necessarily vague: there are many different sorts of stretch driver, some of which do not deal with non-volatile storage at all. There are also potentially many different kinds of backing store. The most important of these is the *User-Safe Backing Store* (USBS). This draws on the work described in [14] to provide per-application guarantees on paging bandwidth, along with isolation between paging and file-system clients.

Allocation is performed in a centralised way by the system domain, for both virtual and physical memory resources. The high-level part of the translation system is also in the system domain: this is machine-dependent code responsible for the construction of page tables, and the setting up of "NULL" mappings for freshly allocated virtual addresses. These mappings are used to hold the initial protection information, and by default are set up to cause a page fault on the first access. Placing this functionality within the system domain means that the low-level translation system does not need to be concerned with the allocation of page-table memory. It also allows protection faults, page faults and "unallocated address" faults to be distinguished and dispatched to the faulting application.

Memory protection operations are carried out by the application through the stretch interface. This talks directly to the low-level translation system via simple system calls; it is not necessary to communicate with the system domain. Protection can be carried out in this way due to the protection model chosen which includes explicit rights for "change permissions" operations. A light-weight validation process checks if the caller is authorised to perform an operation.

The following subsections explain relevant parts of this architecture in more detail.

6.1 Virtual Address Allocation

Any domain may request a stretch from a stretch allocator, specifying the desired size and (optionally) a starting address and attributes. Should the request be successful, a new stretch will be created and returned to the caller. The caller is now the *owner* of the stretch. The starting address and length of the returned stretch may then be queried; these will always be a multiple of the machine's page size[3].

Protection is carried out at stretch granularity — every *protection domain* provides a mapping from the set of valid stretches to a subset of { *read*, *write*, *execute*, *meta* }. A domain which holds the *meta* right is authorised to modify protections and mappings on the relevant stretch.

When allocated, a stretch need not in general be backed by physical resources. Before the virtual address may be referred to the stretch must be associated with a *stretch driver* — we say that a stretch must be *bound* to a stretch driver. The stretch driver is the object responsible for providing any backing (physical memory, disk space, etc.) for the stretch. Stretch drivers are covered in Section 6.6.

6.2 Physical Memory Management

As with virtual memory, the allocation of physical memory is performed centrally, in this case by the *frames allocator*. The frames allocator allows fine-grained control over the allocation of physical memory, including I/O space if appropriate. A domain may request specific physical frames, or frames within a "special" region[4]. This allows an application with platform knowledge to make use of page colouring [30], or to take advantages of superpage TLB mappings, etc. A default allocation policy is also provided for domains with no special requirements.

Unlike virtual memory, physical memory is generally a scarce resource. General purpose operating systems tend to deal with contention for physical memory by performing *system-wide load balancing*. The operating system attempts to (dynamically) share physical memory between competing processes. Frames are *revoked* from one process and granted to another. The main motivation is global system performance, although some systems may consider other factors (such as the estimated working set size or process class).

Since in Nemesis we strive to devolve control to applications, we use an alternative scheme. Each application has a contract with the frames allocator for a certain number of *guaranteed* physical frames. These are immune from revocation in the short term (on the order of tens of seconds). In addition to these, an application may have some number of *optimistic* frames, which may be revoked at much shorter notice. This distinction only applies to frames of main memory, not to regions of I/O space.

When a domain is created, the frames allocator is requested to admit it as a client with a service contract $\{g, x\}$. This represents a pair of quotas for guaranteed and optimistic

[3]Where multiple page sizes are supported, "page size" refers to the size of the smallest page.

[4]Such as DMA-accessible memory on certain architectures.

frames respectively. Admission control is based on the requested guarantee g — the sum of all guaranteed frames contracted by the allocator must be less than the total amount of main memory. This is to ensure that the guarantees of all clients can be met simultaneously.

The frames allocator maintains the tuple $\{n, g, x\}$ for each client domain, where n is the number of physical frames allocated so far. As long as $g > n$, a request for a single physical frame is guaranteed to succeed[5]. If $g \leq n < x$ and there is available memory, frames will be *optimistically* allocated to the caller.

The allocation of optimistic frames improves global performance by allowing applications to use the available memory when it is not otherwise required. If, however, a domain wishes to use some more of the frames guaranteed to it, it may be necessary to *revoke* some optimistically allocated frames from another domain. In this case, the frames allocator chooses a candidate application[6], but the selection of the frames to release (and potentially write to the backing store) is under the control of the application.

By convention, each application maintains a *frame stack*. This is a system-allocated data structure which is writable by the application domain. It contains a list of physical frame numbers (PFNs) owned by that application ordered by 'importance' — the top of the stack holds the PFN of the frame which that domain is most prepared to have revoked.

This allows revocation to be performed *transparently* in the case that the candidate application has *unused* frames at the top of its stack. In this case, the frames allocator can simply reclaim these frames and update the application's frame stack. Transparent revocation is illustrated on the left-hand side of Figure 4.

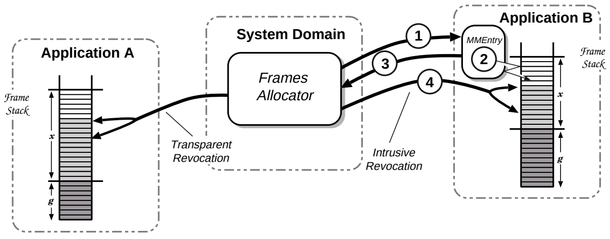
If there are no unused frames available, *intrusive* revocation is required. In this case, the frames allocator sends a revocation notification to the application requesting that it release k frames by time T. The application then must arrange for the top k frames of its frame stack to contain unmapped frames. This can require that it first clean some dirty pages; for this reason, T may be relatively far in the future (e.g. 100ms).

After the application has completed the necessary operations, it informs the frames allocator that the top k frames may now be reclaimed from its stack. If these are not all unused, or if the application fails to reply by time T, the domain is killed and all of its frames reclaimed. This protocol is illustrated on the right-hand side of Figure 4.

Notice that since the frames allocator *always* revokes from the top of an application's frame stack, it makes sense for the application to maintain its preferred revocation order.

[5]Due to fragmentation, a single request for up to $(g - n)$ frames may or may not succeed.

[6]i.e. one which currently has optimistically allocated frames.



① The frames allocator sends a revocation notification to Application B.
② Application B fields this notification, and arranges for the top k frames on the stack to be unused.
③ Application B replies that all is now ready.
④ The frames allocator reclaims the top k frames from the stack.

Figure 4: Revocation of Physical Memory

The frame stack also provides a useful place for stretch drivers to store local information about mappings, and enables the internal revocation interface to be simpler.

Due to the need for relatively large timeouts, a client domain requesting some more guaranteed frames may have to wait for a non-trivial amount of time before its request succeeds. Hence time-sensitive applications generally request *all* their guaranteed frames during initialisation and do not use optimistically allocated frames at all. This is not mandated, however: use of physical memory is considered orthogonal to use of other resources. The only requirement is that any domain which uses optimistically allocated frames should be able to handle a revocation notification.

6.3 Translation System

The translation system deals with inserting, retrieving or deleting mappings between virtual and physical addresses. As such it may be considered an interface to a table of information held about these mappings; the actual mapping will typically be performed as necessary by whatever memory management hardware or software is present.

The translation system is divided into two parts: a high-level management module, and the low-level trap handlers and system calls. The high-level part is private to the system domain, and handles the following:

- Bootstrapping the 'MMU' (in hardware or software), and setting up initial mappings.
- Adding, modifying or deleting ranges of virtual addresses, and performing the associated page table management.
- Creating and deleting protection domains.

- Initialising and partly maintaining the *RamTab*; this is a simple data structure maintaining information about the current use of frames of main memory.

The high-level translation system is used by both the stretch allocator and the frames allocator. The former uses it to setup initial entries in the page table for stretches it has created, or to remove such entries when a stretch is destroyed. These entries contain protection information but are by default *invalid*: i.e. addresses within the range will cause a page fault if accessed. The frames allocator, on the other hand, uses the *RamTab* to record the owner and logical frame width of allocated frames of main memory.

Recall that each domain is expected to deal with mapping its own stretches. The low-level translation system provides direct support for this to happen efficiently and securely. It does this via the following three operations:

1. `map(va, pa, attr)` : arrange that the virtual address `va` maps onto the physical address `pa` with the (machine-dependent) PTE attributes `attr`.
2. `unmap(va)` : remove the mapping of the virtual address `va`. Any further access to the address should cause some form of memory fault.
3. `trans(va)` $\rightarrow$ `(pa, attr)` : retrieve the current mapping of the virtual address `va`, if any.

Either mapping or unmapping a virtual address `va` requires that the calling domain is executing in a protection domain which holds a *meta* right for the stretch containing `va`. A consequence of this is that it is not possible to map a virtual address which is not part of some stretch[7].

It is also necessary that the frame which is being used for mapping (or which is being unmapped) is validated. This involves ensuring that the calling domain owns the frame, and that the frame is not currently mapped or nailed. These conditions are checked by using the *RamTab*, which is a simple enough structure to be used by low-level code.

6.4 Fault Dispatching

Apart from TLB misses which are handled by the low-level translation system, all other faults are dispatched directly to the faulting application in order to prevent QoS crosstalk. To prevent the need to block in the kernel for a user-level entity, the kernel-part of fault handling is complete once the dispatch has occurred. The application must perform any additional operations, including the resumption (or termination) of the faulting thread.

The actual dispatch is achieved by using an *event channel*. Events are an extremely lightweight primitive provided by the kernel — an event "transmission" involves a few sanity checks followed by the increment of a 64-bit value. A full description of the Nemesis event mechanism is given in [2].

On a memory fault, then, the kernel saves the current context in the domain's *activation context* and sends an event to the faulting domain. At some point in the future the domain will be selected for activation and can then deal with the fault. Sufficient information (e.g. faulting address, cause, etc.) is made available to the application to facilitate this. Once the fault has been resolved, the application can resume execution from the saved activation context.

6.5 Application-Level Fault Handling

At some point after an application has caused a memory fault, it will be *activated* by the scheduler. The application then needs to handle all the events it has received since it was last activated. This is achieved by invoking a *notification handler* for each endpoint containing a new value; if there is no notification handler registered for a given endpoint, no action is taken. Following this the user-level thread scheduler (ULTS) is entered which will select a thread to run.

Up until the point where a thread is run, the application is said to be running within an *activation handler*. This is a limited execution environment where further activations are disallowed. One important restriction is that inter-domain communication (IDC) is not possible within an activation handler. Hence if the handling of an event requires communication with another domain, the relevant notification handler simply unblocks a worker thread. When this is scheduled, it will carry out the rest of the operations required to handle the event.

The combination of notification handler and worker threads is called an *entry* (after ANSAware/RT [31]). Entries encapsulate a scheduling policy on event handling, and may be used for a variety of IDC services. An entry called the *MMEntry* is used to handle memory management events.

The notification handler of the *MMEntry* is attached to the endpoint used by the kernel for fault dispatching. Hence it gets an upcall every time the domain causes a memory fault. It is also entered when the frames allocator performs a revocation notification (as described in Section 6.2). The 'top' part of the *MMEntry* consists of one or more worker threads which can be unblocked by the notification handler.

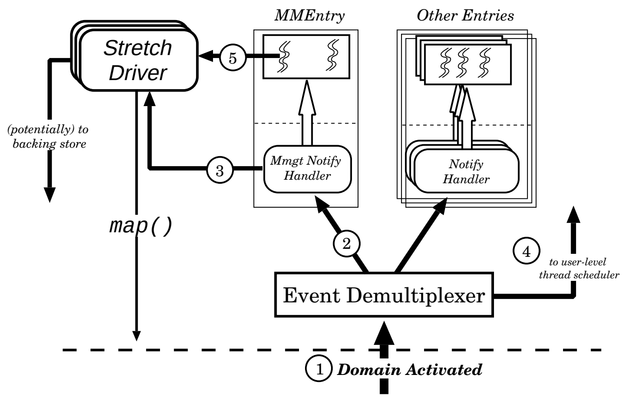
The *MMEntry* does not directly handle memory faults or revocation requests: rather it coordinates the set of stretch drivers used by the domain. It does this in one of two ways:

- If handling a memory fault, it uses the faulting stretch to lookup the stretch driver bound to that stretch and then invokes it.

[7] Bootstrapping code clearly does this, but it uses the high-level translation system and not this interface.

- If handling a revocation notification, it cycles through each stretch driver requesting that it relinquish frames until enough have been freed.

Figure 5 illustrates this in the case of a page fault.



①	The domain receives an event. At some point, the kernel decides to schedule it, and it is *activated*. It is informed that the reason for the activation was the receipt of an event.
②	The user-level event demultiplexer notifies interested parties of any events which have been received on their end-point(s).
③	The memory fault notification handler demultiplexes the stretch to the stretch driver, and invokes this in an initial attempt to satisfy the fault.
④	If the attempt fails, the handler blocks the faulting thread, unblocks a worker thread, and returns. After all events have been handled, the user-level thread scheduler is entered.
⑤	The worker thread in the memory management entry is scheduled and once more invokes the stretch driver to map the fault, which may potentially involve communication with another domain.

Figure 5: Memory Management Event Handling

Note that the initial attempt to resolve the fault (arrow labelled ③) is merely a “fast path” optimisation. If it succeeds, the faulting thread will be able to continue once the ULTS is entered. On the other hand, if the initial attempt fails, the *MMEntry* must block the faulting thead pending the resolution of the fault by a worker thread.

6.6 Stretch Drivers

As has been described, the actual resolution of a fault is the province of a *stretch driver*. A stretch driver is something which provides physical resources to back the virtual addresses of the stretches it is responsible for. Stretch drivers acquire and manage their own physical frames, and are responsible for setting up virtual to physical mappings by invoking the translation system.

The current implementation includes three stretch drivers which may be used to handle faults. The simplest is the *nailed* stretch driver; this provides physical frames to back a stretch at bind time, and hence never deals with page faults. The second is the *physical* stretch driver. This provides no backing frames for any virtual addresses within a stretch initially. The first authorised attempt to access any virtual address within a stretch will cause a page fault which is dispatched in the manner described in Section 6.4. The physical stretch driver is invoked from within the *notification handler*: this is a limited execution environment where certain operations may occur but others cannot. Most importantly, one cannot perform any inter-domain communication (IDC) within a notification handler.

When the stretch driver is invoked, the following occurs:

- After performing basic sanity checks, the stretch driver looks for an unused (i.e. unmapped) frame. If this fails, it cannot proceed further now — but may be able to request more physical frames when activations are on. Hence it returns `Retry`.

- Otherwise, it can proceed now. In this case, the stretch driver sets up the new mapping with a call to `map(va,pa,attr)`, and returns `Success`.

In the case where `Retry` is returned, a memory management entry worker thread will invoke the physical stretch driver for a second time once activations are on. In this case, IDC operations are possible, and hence the stretch driver may attempt to gain additional physical frames by invoking the frames allocator. If this succeeds, the stretch driver sets up a mapping from the faulting virtual address to a newly allocated physical frame. Otherwise the stretch driver returns `Failure`.

The third stretch driver implemented is the *paged* stretch driver. This may be considered an extension of the physical stretch driver; indeed, the bulk of its operation is precisely the same as that described above. However the paged stretch driver also has a binding to the USBS and hence may swap pages in and out to disk. It keeps track of swap space as a bitmap of *bloks* — a blok is a contiguous set of disk blocks which is a multiple of the size of a page. A (singly) linked list of bitmap structures is maintained, and bloks are allocated first fit — a hint pointer is maintained to the earliest structure which is known to have free bloks.

Currently we implement a fairly pure demand paged scheme — when a page fault occurs which cannot be satisfied from the pool of free frames, disk activity of some form will ensue. Clearly this can be improved; however it will suffice for the demonstration of “Quality of Service Firewalling” in Section 7.2.

6.7 User-Safe Backing Store

The user-safe backing store (USBS) is comprised of two parts: the swap filesystem (SFS) and the user-safe disk (USD) [32]. The SFS is responsible for *control* operations such as allocation of an extent (a contiguous range of blocks) for use as a swap file, and the negotiation of Quality of Service parameters to the USD, which is responsible for scheduling *data* operations. This is illustrated in Figure 6.

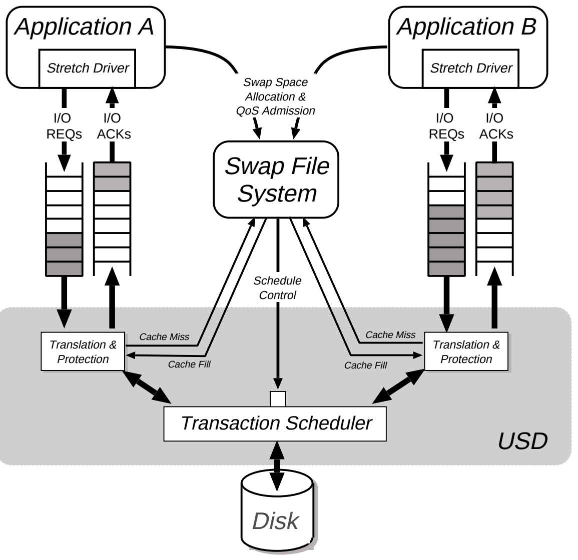


Figure 6: The User-Safe Backing Store

Clients communicate with the USD via a FIFO buffering scheme called *IO channels*; these are similar in operation to the 'rbufs' scheme described in [33].

The type of QoS specification used by the USD is in the form (p, s, x, l) where p is the *period* and s the *slice*; both of these are typically of the order of tens of milliseconds. Such a guarantee represents that the relevant client will be allowed to perform disk transactions totalling at most s ms within every p ms period. The x flag determines whether or not the client is eligible for any slack time which arises in the schedule — for the purposes of this paper it will always be *False*, and so may be ignored.

The actual scheduling is based on the *Atropos* algorithm [2]: this is a based on the earliest-deadline first (EDF) algorithm [34], although the deadlines are implicit, and there is support for *optimistic* scheduling.

Each client is periodically allocated s ms and a deadline of *now* + p ms, and placed on a *runnable* queue. A thread in the USD domain is awoken whenever there are pending requests and, if there is work to be done for multiple clients, chooses the one with the earliest deadline and performs a single transaction.

Once the transaction completes, the time taken is computed and deducted from that client's remaining time. If the remaining time is ≤ 0, the client is moved onto a *wait* queue; once its deadline is reached, it will receive a new allocation and be returned to the runnable queue. Until that time, however, it cannot perform any more operations.

Note that this algorithm will tend to perform requests from a single client consecutively. This is a very desirable property since it minimises the impact of clients upon each other — the first transaction after a "context switch" to a new client will often incur a considerable seek/rotation penalty over which it has no control. However this cost can be amortised over the number of transactions which the client subsequently carries out, and hence has a smaller impact on its overall performance.

Unfortunately, many clients (and most particularly clients using the USD as a swap device) cannot pipeline a large number of transactions since they do not know in advance to/from where they will wish to write/read. Early versions of the USD scheduler suffered from this so-called "short-block" problem: if the client with the earliest deadline has (instantaneously) no further work to be done, the USD scheduler would mark it idle, and ignore it until its next periodic allocation.

To avoid this problem, the idea of "laxity" is used, as given by the l parameter of the tuple mentioned above. This is a time value (typically a small number of milliseconds) for which a client should be allowed to remain on the runnable queue, even if it currently has no transactions pending. This does not break the scheduling algorithm since the additional time spent — the *lax time* — is accounted to the client just as if it were time spent performing disk transactions. Section 7.2 will show the beneficial impact of laxity in the case of paging operations.

7 Experiments

7.1 Micro-Benchmarks

In order to evaulutate the combination of low-level and application-level memory system functions, a set of micro-benchmarks based on those proposed in [23] were performed on Nemesis and compared with Digital OSF1 V4.0 on the same hardware (PC164) and basic page table structure (linear). The results are shown in Table 1.

The first benchmark shown is `dirty`. After [9] this measures the time to determine whether a page is dirty or not. On Nemesis this simply involves looking up a random page table entry and examining its 'dirty' bit[8]. We use a *linear* page table implementation (i.e. the main page table is an 8Gb array in the virtual address space with a secondary

[8]We implement 'dirty' and 'referenced' using the FOR/FOW bits; these are set by software and cleared by the PALCODE DFault routine.

OS	**dirty**	**(un)prot1**	**(un)prot100**
OSF1 V4.0	n/a	3.36	5.14
Nemesis	0.15	0.42 [0.40]	10.78 [0.30]
	trap	**appel1**	**appel2**
OSF1 V4.0	10.33	24.08	19.12
Nemesis	4.20	5.33	9.75†

†Non-standard — see main text.

Table 1: Comparative Micro-Benchmarks; the units are μs.

page table used to map it on "double faults") which provides efficient translation; an earlier implementation using *guarded* page tables was about three times slower.

The second benchmark measures the time taken to protect or unprotect a random page. Since our protection model requires that all pages of a stretch have the same access permissions, this amounts to measuring the time required to change the permissions on small stretches. There are two ways to achieve this under Nemesis: by modifying the page tables, or by modifying a *protection domain* — the times for this latter procedure are shown in square brackets.

The third benchmark measures the time taken to (un)protect a range of 100 pages. Nemesis does not have code optimised for the page table mechanism (e.g. it looks up each page in the range individually) and so it takes considerably longer than (un)protecting a single page. OSF1, by contrast, shows only a modest increase in time when protecting more than one page at a time. Nemesis does perform well when using the protection domain scheme.

This benchmark is repeated a number of times with the same range of pages and the average taken. Since on Nemesis the protection scheme detects idempotent changes, we alternately protect and unprotect the range; otherwise the operation takes an average of only 0.15μs. If OSF1 is benchmarked using the Nemesis semantics of alternate protections, the cost increases to $\sim$75μs.

The `trap` benchmark measures the time to handle a page fault in user-space. On Nemesis this requires that the kernel send an event ($<$50ns), do a full context save ($\sim$750ns), and then activate the faulting domain ($<$200ns). Hence approximately 3μs are spent in the unoptimised user-level notification handlers, stretch drivers and thread-scheduler. This could clearly be improved.

The next benchmark, `appel1` (this is "prot1+trap+unprot" in [23]), measures the time taken to access a random protected page and, in the fault handler, to unprotect that page and protect another. This uses a standard (physical) stretch driver with the access violation fault type overridden by a custom fault-handler; a more efficient implementation would use a specialised stretch driver.

The final benchmark is `appel2`, which is called "protN+trap+unprot" in [23]. This measures the time taken to protect 100 contiguous pages, to access each in a random order and, in the fault handler, unprotect the relevant page. It is not possible to do this precisely on Nemesis due to the protection model — all pages of a stretch must have the same accessibility. Hence we unmap all pages rather than protecting them, and map than rather than unprotecting them. An alternative solution would have been use the Alpha FOW bit, but this is reserved in the current implementation.

7.2 Paging Experiments

A number of simple experiments have been carried out to illustrate the operation of the system so far described. The host platform was a Digital EB164 (with a 21164 CPU running at 266Mhz) equipped with a NCR53c810 Fast SCSI-2 controller with a single disk attached. The disk was a 5400 rpm Quantum (model VP3221), 2.1Gb in size (4,304,536 blocks with 512 bytes per block). Read caching was enabled, but write caching was disabled (the default configuration).

The purpose of these experiments is to show the behaviour of the system under heavy load. This is simulated by the following:

- Each application has a tiny amount of physical memory (16Kb, or 2 physical frames), but a reasonable amount of virtual memory (4Mb).
- A trivial amount of computation is performed per page — in the tests, each byte is read/written but no other substantial work is performed.
- No pre-paging is performed, despite the (artificially) predictable reference pattern.

A test application was written which created a paged stretch driver with 16Kb of physical memory and 16Mb of swap space, and then allocated a 4Mb stretch and bound it to the stretch driver. The application then proceeded to sequentially read every byte in the stretch, causing every page to be demand zeroed.

7.2.1 Paging In

The first experiment is designed to illustrate the overall performance and isolation achieved when multiple domains are paging in data from different parts of the same disk. The test application continues from the initialisation described above by writing to every byte in the stretch, and then forking a "watch thread". The main thread continues sequentially accessing every byte from the start of the 4Mb stretch, incrementing a counter for each byte 'processed' and looping around to the start when it reaches the top.

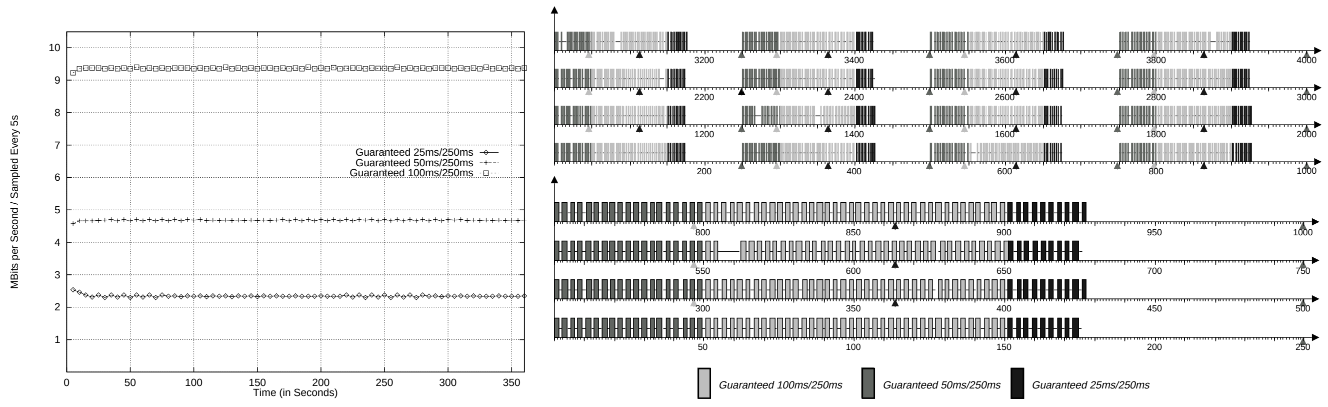


Figure 7: Paging In (*lhs* shows sustained bandwidth, *rhs* shows a USD Scheduler Trace)

The watch thread wakes up every 5 seconds and logs the number of bytes processed.

The experiment uses three applications: one is allocated 25ms per 250ms, the second allocated 50ms per 250ms, and the third allocated 100ms per 250ms — the same period is used in each case to make the results easier to understand. No domain is eligible for slack time, and all domains have a *laxity* value of 10ms. The resulting measured progress (in terms of Mbits/second) is plotted on the left hand side of Figure 7.

Observe that the ratio between the three domains is very close to $4 : 2 : 1$, which is what one would expect if each domain were receiving all of its guaranteed time. In order to see what was happening in detail, a log was taken inside the USD scheduler which recorded, among other things:

- each time a given client domain was scheduled to perform a transaction,
- the amount of lax time each client was charged, and
- the period boundaries upon which a new allocation was granted.

The right hand side of Figure 7 shows these events on two different time-scales: the top plot shows a four second sample, while the bottom plot shows the first second of the sample in more detail. The darkest gray squares represent transactions from the application with the smallest guarantee (10%), while the lightest gray show those from the application with the highest (40%). The small arrows in each colour represent the point at which the relevant client received a new allocation.

Each filled box shows a transaction carried out by a given client — the width of the box represents the amount of time the transaction took. All transactions in the sample given take roughly the same time; this is most likely due to the fact that the sequential reads are working well with the cache.

The solid lines between the transactions (most visible in the detailed plot) illustrate the effect of *laxity* on the scheduler: since there is only one thread *causing* page faults, and one thread *satisfying* them, no client ever has more than one transaction outstanding. In this case the EDF algorithm unmodified by laxity would allow each client exactly one transaction per period.

Notice further that the length of any laxity line never exceeds 10ms, the value specified above, and that the use of laxity does not impact the deadlines of clients.

7.2.2 Paging Out

The second experiment is designed to illustrate the overall performance and isolation achieved when multiple domains are paging out data to different parts of the same disk. The test application operates with a slightly modified stretch driver in order to achieve this effect — it “forgets” that pages have a copy on disk and hence never pages in during a page fault. The other parameters are as for the previous experiment. The resulting progress is plotted on the left hand side of Figure 8.

As can been seen, the domains once again proceed roughly in proportion, although overall throughput is much reduced. The reason for this is in the detailed USD scheduler trace on the right hand side of Figure 8. This clearly shows that almost every transaction is taking on the order of 10ms, with some clearly taking an additional rotational delay. This is most likely due to the fact that individual transactions are separated by a small amount of time, hence preventing the driver from performing any transaction coalescing.

One may also observe the fact that the client with the smallest slice (which is 25ms) tends to complete three transactions (totalling more than 25ms) in some periods, but then will obtain less time in the following period. This is since we employ a roll-over accounting scheme: clients are al-

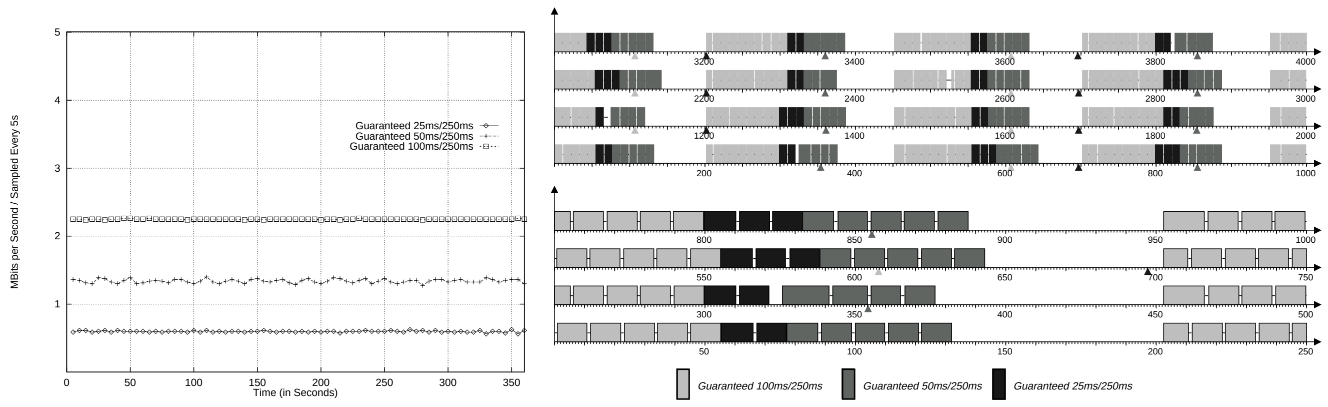


Figure 8: Paging Out (*lhs* shows sustained bandwidth, *rhs* shows a USD Scheduler Trace)

lowed to complete a transaction if they have a reasonable amount of time remaining in the current period. Should their transaction take more than this amount of time, the client will end with a negative amount of remaining time which will count against its next allocation.

Using this technique prevents an application deterministically exceeding its guarantee. It is not perfect — since it allows jitter to be introduced into the schedule — but it is not clear that there is a better way to proceed without intimate knowledge of the disk caching and scheduling policies.

7.3 File-System Isolation

The final experiment presented here adds another factor to the equation: a client domain reading data from another partition on the same disk. This client performs significant *pipelining* of its transaction requests (i.e. it trades off additional buffer space against disk latency), and so is expected to perform well. For homogeneity, its transactions are each the same size as a page.

The file-system client is guaranteed 50% of the disk (i.e. 125ms per 250ms). It is first run on its own (i.e. with no other disk activity occuring) and achieves the sustained bandwidth shown in the left hand side of Figure 9. Subsequently it was run again, this time concurrently with two paging applications having 10% and 20% guarantees respectively. The resulting sustained bandwidth is shown in the right hand side of Figure 9.

As can be seen, the throughput observed by the file-system client remains almost exactly the same despite the addition of two heavily paging applications.

8 Conclusion

This paper has presented the idea of *self-paging* as a technique to provide Quality of Service to applications. Experiments have shown that it is possible to accurately isolate the effects of application paging, which allows the coexistence of paging along with time-sensitive applications. Most of the VM system is provided by unprivileged user-level modules which are explicitly and dynamically linked, thus supporting extensibility.

Performance can definitely be improved. For example, the 3μs overhead in user-space trap-handling could probably be cut in half. Additionally the current stretch driver implementation is immature and could be extended to handle additional pipe-lining via a “stream-paging” scheme such as that described in [24].

A more difficult problem with the self-paging approach, however, is that of *global* performance. The strategy of allocating resources directly to applications certainly gives them more control, but means that optimisations for global benefit are not directly enforced. Ongoing work is looking at both centralised and devolved solutions to this issue.

Nonetheless, the result is promising: virtual memory techniques such as demand-paging and memory mapped files have proved useful in the commodity systems of the past. Failing to support them in the continuous media operating systems of the future would detract value, yet supporting them is widely perceived to add unacceptable unpredictability. Self-paging offers a solution to this dilemma.

Acknowledgments

I should like to express my extreme gratitude to Paul Barham who encouraged me to write this paper, and wrote the tools to log and post-process the USD scheduler traces. Without his help, this paper would not have been possible.

I would also like to thank the anonymous reviewers and my shepherd, Paul Leach, for their constructive comments and suggestions.

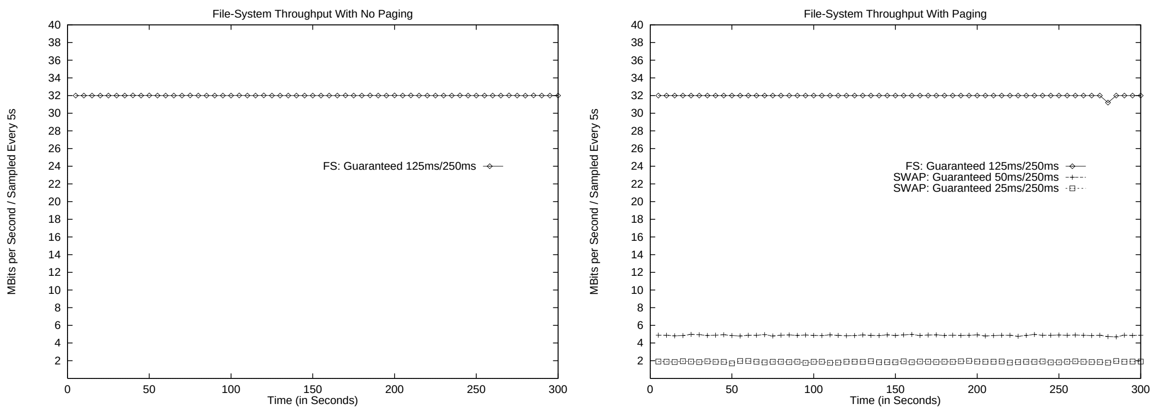


Figure 9: File-System Isolation

Availability

The Nemesis Operating System has been developed as part of the Pegasus II project, supported by the European Communities' ESPRIT programme. A public release of the source code will be made in 1999.

References

[1] E. Hyden. *Operating System Support for Quality of Service*. PhD thesis, University of Cambridge Computer Laboratory, February 1994.

[2] T. Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, University of Cambridge Computer Laboratory, April 1995.

[3] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas In Communications*, 14(7):1280–1297, September 1996. Article describes state in May 1995.

[4] M. B. Jones, P. J. Leach, R. P. Draves, and III J. S. Barrera. Modular Real-Time Resource Management in the Rialto Operating System. In *Processings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 12–17, May 1995.

[5] M. B. Jones, III J. S. Barrera, A. Forin, P. J. Leach, D. Rosu, and M. Rosu. An Overview of the Rialto Real-Time Architecture. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 249–256, September 1996.

[6] M. B. Jones, D. Rosu, and M. Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles*, pages 198–211, October 1997.

[7] D. Mosberger. *Scout: A Path-Based Operating System*. PhD thesis, University of Arizona, Department of Computer Science, 1997.

[8] J. Nieh and M. S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles*, pages 184–197, October 1997.

[9] D. Engler, S. K. Gupta, and F. Kaashoek. AVM: Application-Level Virtual Memory. In *Processings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, May 1995.

[10] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles*, December 1995.

[11] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, October 1996.

[12] R. P. Draves, G. Odinak, and S. M. Cutshall. The Rialto Virtual Memory System. Technical Report MSR-TR-97-04, Microsoft Research, Advanced Technology Division, February 1997.

[13] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[14] P. R. Barham. A Fresh Approach to Filesystem Quality of Service. In *7th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 119–128, St. Louis, Missouri, USA, May 1997.

[15] P. Shenoy and H. M. Vin. Cello: A Disk Scheduling Framework for Next-Generation Operating Systems. In *Proceedings of ACM SIGMETRICS'98, the International Conference on Measurement and Modeling of Computer Systems*, June 1998.

[16] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th ACM SIGOPS Symposium on Operating Systems Principles*, pages 63–76, November 1987.

[17] K. Harty and D. R. Cheriton. Application-Controlled Physical Memory using External Page-Cache Management. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 187–197, October 1992.

[18] Y. A. Khalidi and M. N. Nelson. The Spring Virtual Memory System. Technical Report SMLI TR-93-9, Sun Microsystems Laboratories Inc., February 1993.

[19] J. Liedtke. On μ-Kernel Construction. In *Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles*, pages 237–250, December 1995.

[20] Y. Endo, J. Gwertzman, M. Seltzer, C. Small, K. A. Small, and D. Tang. VINO: the 1994 Fall Harvest. Technical Report TR-34-94, Center for Research in Computing Technology, Harvard University, December 1994. Compilation of six short papers.

[21] D. R. Cheriton and K. J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 179–194, November 1994.

[22] M. Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981.

[23] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 96–107, April 1991.

[24] G. E. Mapp. *An Object-Oriented Approach to Virtual Memory Management*. PhD thesis, University of Cambridge Computer Laboratory, January 1992.

[25] P. Cao. *Application-Controlled File Caching and Prefetching*. PhD thesis, Princeton University, January 1996.

[26] C. A. Thekkath and H. M. Levy. Hardware and Software Support for Efficient Exception Handling. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 110–121, October 1994.

[27] D. Engler, F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles*, December 1995.

[28] R. Black, P. Barham, A. Donnelly, and N. Stratford. Protocol Implementation in a Vertically Structured Operating System. In *Proceedings of the 22nd Conference on Local Computer Networks*, pages 179–188, November 1997.

[29] M. F. Kaashoek, D. R. Engler, G. R. Granger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles*, pages 52–65, October 1997.

[30] B. N. Bershad, D. Lee, T. H. Romer, and J. Bradley Chen. Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 158–170, October 1994.

[31] Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, UK. *ANSAware/RT 1.0 Manual*, March 1995.

[32] P. Barham. *Devices in a Multi-Service Operating System*. PhD thesis, University of Cambridge Computer Laboratory, July 1996.

[33] R. J. Black. *Explicit Network Scheduling*. PhD thesis, University of Cambridge Computer Laboratory, April 1995.

[34] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.