ESPRIT LTR 21917 (Pegasus II)

Deliverable 2.3.1: Virtual Address Management
Deliverable 2.3.2: Virtual Memory Management

Steven Hand
University of Cambridge
July 1997

1 Introduction

This document describes the current status of *virtual address management* in the Nemesis operating system. As the project is also actively pursuing the broader topic of *virtual memory management*, it has been decided to describe this work here in the hope that a clearer overall framework may be presented.

Some background is initially given regarding the general goals and philosophy of the Nemesis operating system, followed by an overview of the general architecture of the virtual memory system. Section 4 presents the design of the system in detail. This is followed by a description of a prototype implementation of the virtual memory system, a brief look at related work, and a summary.

2 Goals

Nemesis is an operating system designed to support a mixture of time-sensitive (e.g. multimedia) and standard applications. It achieves this by:

1. providing applications with explicit guarantees for *real* resources,
2. enabling applications to execute the vast majority of of operations themselves, and then
3. making applications responsible for their behaviour.

In terms of CPU allocation, for example, this strategy is realised by means of a soft real-time scheduling algorithm and the use of scheduler activations [**Anderson92**]. Once an application has sufficient knowledge to observe exactly what occurs in the present, and to predict what will occur in the future, it may choose its operational parameters with confidence. This fact has been successfully demonstrated in the past [**Hyden94**, **Roscoe95**, **Leslie96**].

Nemesis has been constructed as a *single address space operating system* (SASOS); i.e. translations from virtual to physical addresses are shared between all executing domains[1]. This is in contrast to typical operating systems such as Unix, VMS and Windows NT, where each executing process (or task) sees an individual virtual address space. Note that global translations do not in any way imply any lack of inter-domain protection: all domains execute in

[1]A Nemesis *domain* may be considered the analog of a process or task in other operating systems.

some *protection domain* which contains a set of access rights for any given page. Thus there is a clear separation between the concepts of *protection* and of *translation*.

The decision to make Nemesis a SASOS was motivated by a number of clear benefits:

- Lower Context Switch Times.
  The support of applications which have temporal constraints requires fine-grained multiplexing of the CPU. This leads to more context switches. In a SASOS, a single virtual address translation structure can be used for all domains, and hence need not be context switched. While this itself can only give a modest improvement, there is potentially a huge additional benefit from improved cache and TLB performance.
- Reduced Data Copying.
  Many continuous media applications deal with streams of data. In the general case this data may require processing by a number of different domains. Rather than provide special purpose 'bulk transfer' IPC mechanisms such as those implemented by operating system μ-kernels like Mach [**Young87**], EROS [**Shapiro96**] and L4 [**Liedtke96a**], a SASOS can simply share data between different domains. This means that IPC times are not dominated (or even affected) by the amount of data which is logically transferred. One- or zero-copy network stack implementations are also more easily achieved.
- Rich Sharing of Text.
  One important factor in the provision of Quality of Service (QoS) is the *independence* of scheduled tasks from one another. This may most conveniently be achieved by enabling each application to execute as much of its own code as possible. This provides true application accountability, aiding resource management and banishing "QoS crosstalk" [**Roscoe95**]. In a SASOS, sharing text is trivial.

Nemesis is almost entirely composed of shared library code. The linkage model chosen is that of *closures* [**Roscoe94**], which usefully allow for the sharing of program text among different instantiations with differing state. This model allows for very efficient sharing of code (one additional memory dereference per operation) at a fine granularity. It effectively allows nearly all functions traditionally considered "kernel" or "server" features to be executed by applications themselves. The resultant operating system structure has been described as "vertically integrated" [**Barham96**].

This strategy of forcing applications to execute nearly all operations themselves also helps achieve another of the Nemesis goals: *extensibility*. While there is a large amount of shared library code available to applications, there are no requirements that they make use of these standard implementations. Nemesis uses an interface definition language (IDL) to help demarcate the boundaries between *interface* and *implementation*. The operating system defines the system interfaces and provides default implementations, but allows applications choose the actual implementation they use.

The recent emergence of 64-bit microprocessor architectures has prompted the development of a number of new single address space operating systems including Angel [**Murray93**], Opal [**Chase94**] and Mungi [**Heiser94**]. All of these, however, are motivated by the potential inherent in a wide address space for the support of distributed shared virtual memory (DSVM) or persistent memory mapped objects

Nemesis does not view either of these as fundamental operating system issues. They represent choices which should be made by individual applications, not mandated by the operating system. The Nemesis operating system securely multiplexes resources between domains; applications are responsible for the rest.

In terms of the memory system, this means that while resource allocation (viz. virtual and physical address allocation) needs to be carried out in a secure and centralised manner, little else does. Setting up protection, virtual address mappings or even handling memory faults are application tasks.

3 System Architecture

This section presents the virtual memory system from an abstract point of view. A number of definitions are introduced, and a simple description of the overall system is given.

3.1 Some Definitions

The following definitions are essential to the understanding of the Nemesis virtual memory system:

Virtual Address Space: the set of valid virtual addresses. On a single machine in Nemesis, any process will 'see' the same virtual address space, the only difference being the *access rights* that the process has on each part of the address space.

Physical Address Space: the set of valid physical addresses. This includes any physically addressable location in a machine's memory, not just RAM.

Page: the smallest protectable component of virtual address space. In a sense, it is the granularity of the virtual address space with regard to access control.

Frame: the physical equivalent of a page. It is also likely to be the granularity at which areas of the physical address space are allocated. A frame typically contains the same number of bytes as a page.

Protection Domain: a function from the set of pages in the virtual address space to the set of *access rights*. Each protection domain thus represents the protection environment in which domains execute.

Access Right: zero or more of `read`, `write`, `execute` and `meta`, as applied to part of the virtual address space. These are described in Section 4.1.

Accessibility: a function from the set of active protection domains to the set of access rights. Thus the accessibility of a page is the set of its access rights in every protection domain.

Stretch: a contiguous adamantine region of the virtual address space with the same accessibility. This is similar to a *region* in Mach or Chorus, or even to a MS-DOS *arena*. It is not, however, to be confused with a Unix-style segment, which can shrink and grow at will — a stretch corresponds to the same set of contiguous pages throughout its lifetime.

Stretches are also disjoint, i.e., they cannot overlap. One consequence is that stretches cannot be *refined*: if you have a stretch it doesn't make sense to talk to the VM system about a subset of the stretch.

A stretch must be associated with a *stretch driver* before it can be useful. A stretch by itself is merely an abstract object created by the stretch allocator; it has no meaning to the rest of the system and any attempts by a process to address memory within a stretch which has no stretch driver will result in an unresolvable page fault to the process. All it does have are base address, length and access rights.

Stretch Driver: The entity that is associated with a stretch to provide physical resources, page fault handling, etc. The closest analog of a stretch driver in System VR4 is called a *segment driver*. In Mach, it is a *memory object*.

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 RAM, etc., and are responsible for setting up virtual to physical mappings by invoking the *translation system*.

Frame Stack: the set of frames granted to any individual domain and which it may use for the purpose of mapping.

Translation System: a machine-specific component which provides for entering, removing, retrieving and updating both mapping and protection information for virtual addresses. It may be considered to be somewhat like a memory-management unit.

The above represent the fundamental abstractions of the Nemesis memory system; further concepts and terms will be introduced throughout the document when necessary.

3.2 General Architecture

Considering the basic definitions provided, it is useful to see how they can fit together in a coherent manner. This general architecture is shown in Figure 1. Although the illustration represents a very high-level description of the memory system, a few points are worth mentioning:

- There is a clear separation of address *allocation* and address *translation*.
  This in accordance with the Nemesis philosophy of distinguishing between control- and data-path operations: allocation by its very nature requires a form of central arbitration to prevent conflict between multiple applications, while translation may be safely carried out without the involvement of any central authority.

- A *stretch* is directly involved with virtual addresses and protection; a *stretch driver* is directly involved with physical resources and stretches.
  This emphasises the definitions given earlier: a stretch represents merely a range of virtual addresses with a certain accessibility. It does not own — nor is it guaranteed — any physical resources. It is only via its association with a stretch driver that it becomes possible to talk meaningfully about its "contents".

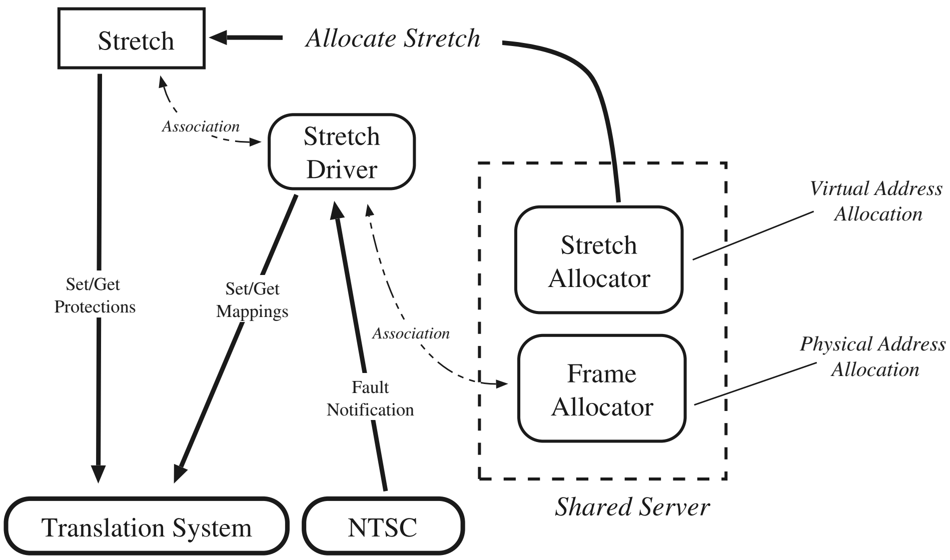


Figure 1: VM System Architecture

- The kernel (viz. NTSC) is not involved save for fault notification.
  This fits with the general structure of Nemesis — the NTSC provides only a small set of system calls and the scheduler. It does not play a large part in the virtual memory architecture presented here.

One further aspect of note is that protection operations are shown as being carried out independently of the shared server responsible for address allocation. This is due to the protection model chosen which includes explicit rights for "change permissions" operations. The protection model, address allocation & translation, and fault handling are each described in more detail in Section 4.

4 System Design

This section presents the design of the virtual memory management system in terms of the general concepts introduced in Section 3.

4.1 Protection Model

In a multi-address space operating system, protection is typically considered to be a function of address space management — each process executes with a myopic view of the virtual address space which prevents unauthorised accesses. For a SASOS this is inappropriate.

Instead, Nemesis uses *protection domains*. A protection domain is a mapping from the virtual address space to a set of access rights, i.e. a protection domain P may be formally defined by:

$$P : V \to R^*$$

where V is the set of valid virtual addresses, and $R \in \{ \textit{read}, \textit{write}, \textit{execute}, \textit{meta} \}$.

Logically, all protection domains are considered subsets of an additional *global* protection domain. This allows the global protecting/sharing of certain memory regions; that is, the granting of a base level of permissions to all protection domains.

Every domain is associated with exactly one protection domain, although multiple domains may share such an association. The protection domain of an executing domain determines the accessibility of any region of memory to that executing domain.

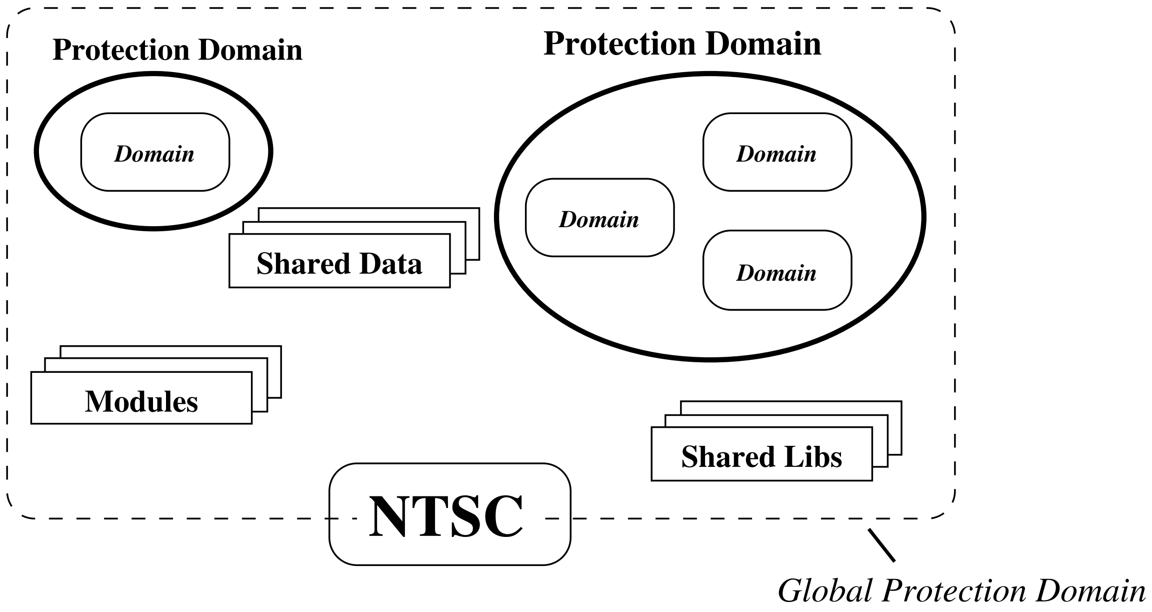


Figure 2: Protection Domains in Nemesis

A large portion of the Nemesis operating system is comprised of shared text in the form of stateless libraries and closure-invoked modules[2] These, along with small amounts of globally accessible data, may be considered to reside within the global protection domain. The NTSC — a rather special case — will in general exist partly within the global protection domain, and partly outside any protection domain. This is illustrated in Figure 2.

There is a close relationship between protection domains and stretches; since a stretch encapsulates a region of the virtual address space with the same accessibility, a protection domain may also be considered as a mapping from *stretches* to access rights. Hence the stretch interface provides a logical place to provide access control modification operations.

When a stretch is allocated, it is initialised with a default set of access rights — *read*, *write* and *meta* — for the protection domain of the caller. The third of these is the most interesting; a *meta* right provides the following functions:

- It identifies an 'authorised protection domain' of the stretch.
  There there may be arbitrarily many of these. A domain executing within an authorised protection domain for a stretch is said to be *authorised* on that stretch.

- It permits the modification of the access rights of the stretch.

[2] A good description of modules in Nemesis is given in [**Roscoe94**].

A stretch provides an interface whereby various access rights (including the *meta* right) may be granted to or revoked from protection domains.

- It authorises the mapping (or unmapping) of physical frames into the stretch. More details on this are given in Section 4.4.

Four kinds of access control operations are provided on a stretch:

1. `SetProt(pdom, rights)` : set the access rights on the stretch for protection domain `pdom` to be `rights`. This requires the invoking domain to be authorised.
2. `SetGlobal(rights)` : set the global access rights on the stretch to be `rights`. This also requires an authorised invoking domain.
3. `QueryProt(pdom)` $\rightarrow$ `rights` : retrieve the access rights for protection domain `pdom`.
4. `QueryGlobal()` $\rightarrow$ `rights` : retrieve the global access rights.

The "global" rights referred to are the rights of the global protection domain of which all other protection domains are subsets. One consequence of this is that the accessibility of a stretch from a particular protection domain is always *at least* the global rights.

4.2 Address Allocation

This section describes the mechanisms by which regions of an address space may be allocated to specific domains, and later freed. Other aspects of the overall memory architecture, such as the translation system, dealing with memory faults, etc. are dealt with in later sections.

4.2.1 Stretch Allocation

A region of the virtual address space is allocated by creating a stretch, and hence the part of the VM system which handles virtual address allocation is called the *stretch allocator*. 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 page size.

There may be a number of stretch allocators, although they will manage non-overlapping regions. This may be useful in cases where it is desirable for certain regions of the virtual address space to be associated with certain properties; for example, one allocator might provide stretches for DMA buffers, another provide stretches which must be mapped in a certain way, and so forth. Using different allocators for these various portions of the address space allows differing resource management policies to be implemented.

A stretch itself encapsulates a particular contiguous region of the virtual address space, as determined by a $(base, length)$ pair, with each address within the region having the same access permissions. It is not possible to move, shorten or extend the stretch once it has been created — the base and length remain the same throughout the lifetime of the stretch.

These properties rule out the possibility of *nested* stretch allocators, since it would not be possible to modify the access permissions of any "sub-stretch" independently of any other. It is of course possible, however, to subdivide and reallocate the virtual addresses of the stretch in terms of some second level allocator (e.g. a heap).

When allocated, a stretch need not in general be backed by physical resources. Before the virtual address may be referred to, then, 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.

Many implementations of stretch drivers exist; two in particular are worth mentioning:

1. Nailed Stretch Driver: this provides physical backing for the virtual addresses spanned by the stretch at all times. This essentially requires that sufficient physical frames be allocated to back the stretch when it is associated with the driver. In such a case, the stretch will never cause a page fault.

2. Demand-Paged Stretch Driver: this provides no physical backing at all for the stretch on first association. Instead, once an address within the stretch is accessed, a page fault will occur. Only at this point will the stretch driver actually provide any physical resources, and may map only the faulting page.

Clearly a large number of alternatives is also conceivable: the only essential property of a stretch driver is that it somehow deal with events involving any part of any stretch with which it is associated.

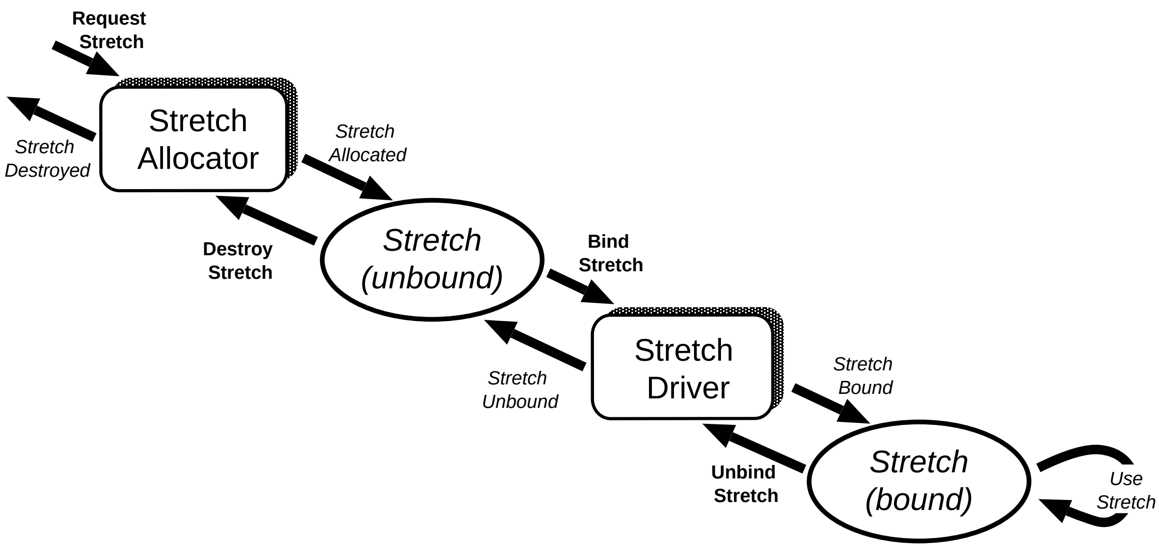


Figure 3: Life-Cycle of a Stretch

The freeing of a stretch is similarly a two stage process. First the stretch is *unbound* from its stretch driver; this allows any physical resources which may have been associated with the stretch to be disposed of correctly. Secondly, the stretch is *destroyed* via a stretch allocator. This must be same stretch allocator used to allocate the stretch initially. Figure 3 shows the life-cycle of a stretch.

Note that the permission to unbind or destroy a stretch is independent of the standard { *read*, *write*, *execute*, *meta* } rights associated with the stretch. Instead it is the responsibility of the relevant stretch driver or allocator to implement a protection scheme. Generally unbinding and destruction of a stretch is restricted to the owning domain.

4.2.2 Physical Memory Allocation

Nemesis provides fine-grained control over the allocation of physical memory, including (where applicable) I/O space. A stretch driver may request specific physical frames, or frames within a "special" region[3]. This allows a stretch driver with platform knowledge to make use of page colouring, or to take advantages of "super-page" TLB mappings, etc. Clearly, a default allocation policy is also supported for domains with no special requirements.

As with virtual memory, the allocation of physical memory is done by a central *frames allocator*. This allocator actually supports two different interfaces so that domains may request physical memory in appropriate ways:

1. Via the *Frames* interface: this allows for the allocation and freeing of individual (or contiguous blocks of) physical frames. In normal circumstances, however, this memory will be of limited use, since it is not possible to address it!

   Only certain privileged domains have the power to perform physical accesses, and hence the use of this interface is restricted to these domains.

2. Via the *FrameStack* interface: like the Frames interface, this allows the allocation and freeing of frames (or groups thereof), but instead of returning raw (and generally useless) physical addresses to the domain, it creates (or augments) that domain's *frame stack*.

   This represents the set of physical frames which a stretch driver may use for backing any of its stretches. The set is recorded in a nailed area defined by the translation system so that it is easy to validate mapping attempts by a domain.

   As has been seen in the previous section, the responsibility for providing physical backing for a stretch is delegated to the stretch driver. Thus in general it is the stretch drivers that deal with a domain's frame stack.

As for stretch allocation, it is often desirable to have more than one frame allocator. This is particularly convenient on machines with memory mapped I/O space or with physical addresses which encode information about caching or buffering. The use of a separate allocator allows:

a) additional access control checks to be carried out since different physical memory spaces are often best limited to certain privilege classes.

b) the use of a different physical frame size: for 'normal' memory on most architectures, frame size = page size = granularity of allocation. However larger frames may be more suitable for I/O-type physical memory regions.

c) the reservation of certain regions of physical memory for system, device or other purposes.

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

d) the separation of allocation and revocation policy based on the characteristics of a region of physical memory.

Clearly in the case where multiple frame allocators are available, it is not necessary for every one of them to provide both the `Frames` and `FrameStack` interface. This might rather be limited to a single allocator.

One further concern for physical memory allocators is that of *revocation*. Limited physical memory[4] generally implies considerable contention.

In Nemesis memory management the ideas of *guaranteed* and *optimistic* resources are used, as they are for other resources in the system, e.g. CPU. A domain has some explicitly guaranteed number of physical frames which are immune from revocation in the short term. In addition, any domain may also have some number of optimistically allocated physical frames. This latter set are currently available for use by the domain, but are subject to later revocation without notification. The fact that the domain knows explicitly which frames it has been granted and under what conditions, allows it to place data in appropriate places.

4.3 Address Translation

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.

Nemesis expects each domain to deal with mapping its own stretches, and as such a domain will generally require some physical frames which it may use for this purpose. Furthermore it is important that attempted use of the translation system can be easily validated to prevent unauthorised behaviour. This validation usually requires two checks to be made:

a) Valid virtual address: the calling domain must execute in a protection domain which holds a *meta* right for the stretch which contains the virtual address. A consequence of this is that it is not possible to map a virtual address which is not part of some stretch.

b) Valid physical address: the calling domain must own the frame which is being used for mapping. Checking this condition is facilitated by the use of the frame stack.

Conceptually, then, the translation system provides three operations:

1. `Map(va, pa)` : arrange so that the virtual address `va` maps onto the physical address `pa`. In practice the `va` will be page-aligned, and the `pa` frame-aligned.

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` : retrieve the currently installed mapping of the virtual address `va`, if any.

[4] In particular here we are concerned with installed RAM on a particular machine, not simply the physical address space of a system.

The translation system is, however, a rather machine-specific object, and so particular implementations may refine or augment the above. A good example of where this is useful is for machines which have software managed TLBs — on such architectures one may use an arbitrary page-table format and thus provide for reference bits if desired. An example of an implementation of the translation system will be seen in Section 5.

4.4 Fault Handling

A wide variety of memory faults may be generated by a typical system. Clearly the exact number, form and semantics of these are machine specific, but a generalised taxonomy may be provided in order to facilitate portable implementations[5]:

TNV (Translation Not Valid): the translation of a virtual address to a physical address failed because there is no valid information about the mapping. Generally this occurs when an unallocated virtual address is used.

ACV (Access Violation): an attempt to access some virtual address failed due to insufficient privilege (i.e. a protection fault).

FOR Fault On Read: a read access was attempted on a physical page marked as "fault on read". This may be used to collect access pattern statistics, or as a hook for persistence, and so on.

FOE (Fault On Execute): an instruction stream fetch was attempted from a non-executable page.

FOW (Fault On Write): a write access was attempted to a page marked "fault on write". This might, for example, be used to implement copy on write.

UNA (Unaligned Access): an unaligned access was attempted. This may simply be a bad virtual address, or may be a true unaligned attempt.

PAGE (Page Fault): the page to which an access was attempted is not currently mapped.

The above list includes all the faults which Nemesis currently recognises; it is not expected that every type of fault will be possible on every architecture.

Of key importance within Nemesis is the idea of accountability: every domain should perform whatever tasks are necessary for its own execution. In terms of the VM system, this means that a domain is responsible for satisfying any faults which occur on stretches which it owns. There is no concept of a "system pager", or of an "application level pager": instead, every domain is its own pager (or, more generally, memory fault handler).

Clearly this would be unduly complex in a traditional or μ-kernel operating system, but with a vertically integrated system, practically any system task may be easily shared by all domains. In this particular case, the domain will typically invoke the library code of an implemented stretch driver, though it may use a private implementation if it desires.

In order to achieve the above we require a light-weight low latency asynchronous mechanism whereby a domain may be notified of a fault occurring. The NTSC provides this by means of

[5]The astute reader may notice a similarity here to the alpha architecture reference manual.

the *event* mechanism, which allows the asynchronous notification of a domain of some happening. Event channels are used throughout Nemesis to provide inter-domain communication and to support device driver stubs — and to signal memory faults.

On a memory fault, the NTSC identifies the stretch containing the faulting address and sends the faulting domain an event. The next time this domain is activated, it should resolve the fault. This may involve simply mapping one of its unmapped frames, or may involve paging out a resident page. The decision on which page to replace is totally under the control of the domain which may therefore use whatever policy it prefers.

If the stretch was being shared, the domain which faulted may not be sufficiently authorised (i.e. may not possess *meta* rights for the stretch). In this case, it receives the event and notices that it cannot resolve the fault itself. If, as is common, it contains a user-level thread scheduler, it may decide to simply run another thread. Alternatively it may optimistically retry the faulting instruction, or attempt to contact a domain which can potentially resolve the fault.

It is important to understand how the domain handles the incoming event, and how it notifies the correct handler accordingly. Figure 4 illustrates how domains can handle events: Note that the structure of the user-space part illustrated is not a mandatory aspect of the design — any domain may deal with incoming events any way in which it wishes. The illustrated structure is, however, typical.

In the case of memory faults, the first two stages of the event delivery happen as normal. In the notification handler, however, there are usually two choices:

i) The fault is fixed up by the stretch driver within the notification handler itself. This is the "fast path" option and is suitable where no further action is required in order to satisfy the fault.

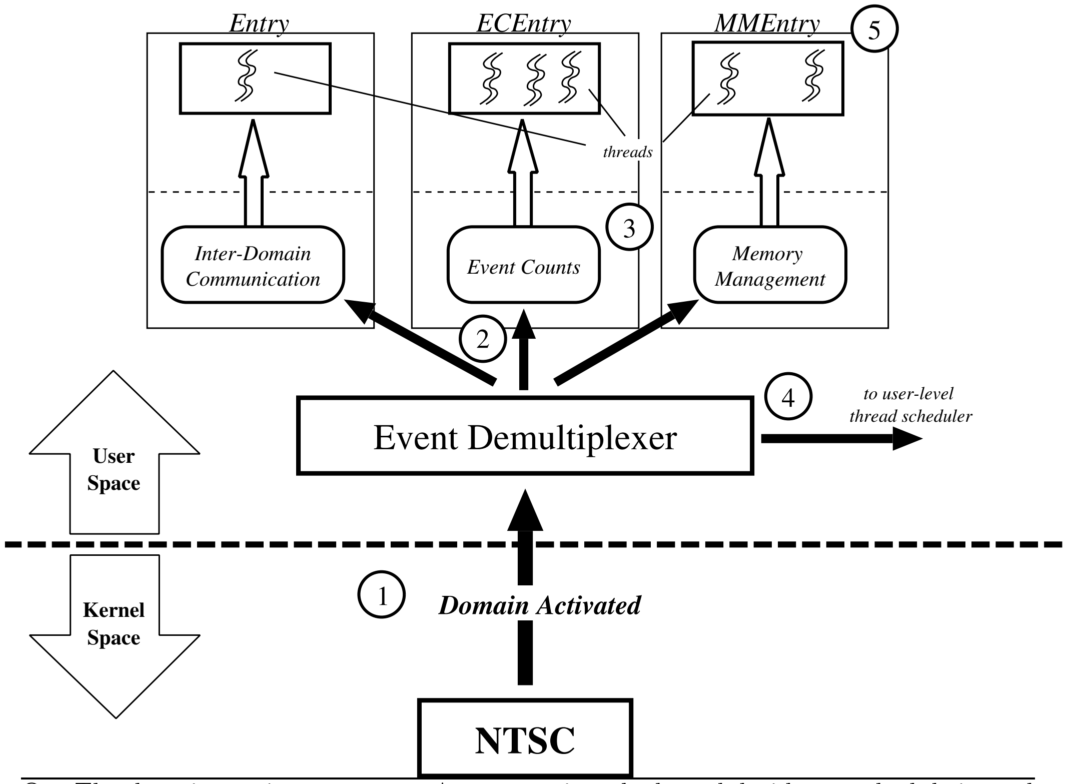
ii) The notification handler simply unblocks a worker thread in the memory management entry (MMEntry). This thread will later be scheduled by the ULS and will invoke the stretch driver to satisfy the fault.

In the latter case, IDC operations are possible, facilitating paging, the handling of faults on shared stretches, and the invocation of a user-level debugger for unresolvable faults. More details on these aspects of fault handling are given in Section 5.5.

4.5 Low-Level Interfaces

Nemesis is most suited to RISC architectures — the ability to choose a page-table structure for the SAS model, to prefetch TLB entries on a protection domain boundary, or to make use of software TLBs allow a fine-tuning process not possible with CISC MMUs. One additional aspect of certain RISC architectures is software cache invalidation and/or prefetching. Nemesis supports such architectures by means of *cache hints*.

Cache hints provide a declarative interface to the hardware; the actual effect they have is implementation and machine specific. Likely semantics include the invalidation of a cache line (or of the entire cache) and the prefetching of a line (or lines) into a cache. Currently three cache hints are defined:



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 channel(s).
③ Each notification handler may prepare (viz. unblock) a worker thread, and potentially will block a faulting thread.
④ The user-level scheduler is entered.
⑤ A thread in a suitable *entry* is scheduled and deals with the event.

Figure 4: Event Handling

1. `RequireRead(region)` — the caller intends to read the specified addresses.
   The system may decide to prefetch some or all of the relevant lines, or may do nothing.

2. `RequireWrite(region)` — the caller intends to write the specified addresses.
   The system may decide to prefetch some or all of the relevant lines for writing (which might be the same as prefetching for writing).

3. `Modified(region)` — the caller has modified the specified addresses.
   In some architectures[6], this hint may be *required* in order to ensure cache consistency in the presence of DMA access by IO devices. In most cases, however, cache consistency is provided by the hardware and the likely reaction to this hint will be a `nop`.

In each case the associated range of virtual addresses is specified by `region`. The range

[6] Most notably on the StrongARM.

is specified at a level of bytes (or at least words) rather than pages to allow caches to be invalidated or pushed over small objects (for example ATM cell payloads).

It is important to note that various other "useful" low-level features (such as a partitioned cache [**Hayter94**], or a protection translation buffer [**Koldinfer92**]) may be available on other or future architectures. It is difficult to design interfaces for all possible enhancements, and it is not attempted here. Whenever such interfaces *are* designed, however, they should be *declarative*. This allows implementations without the specific hardware to trivially support a `nop` reaction.

5 Implementation

An initial implementation of much of the architecture described here has been completed on one of the Nemesis platforms, the EB164. which uses a Digital Alpha 21164 processor [**DEC95**]. This section gives an overview of the current status and looks ahead to the work remaining to be completed.

5.1 The Translation System

The main component of the translation system is the page-table; this is walked (in software) in order to perform TLB fills, and is modified by the mapping/unmapping operations.

Currently a *guarded page table* [**Liedtke95a**] is used. Guarded page tables (GPTs) are a modification of standard n-level page tables in which each level may translate a variable number of bits. This provides for a flexibility in terms of the "page" sizes which may be mapped. In principle using GPTs can reduce the overall memory requirement for translation information. It is also possible to arrange for a relatively shallow tree (which should result in shorter lookup times).

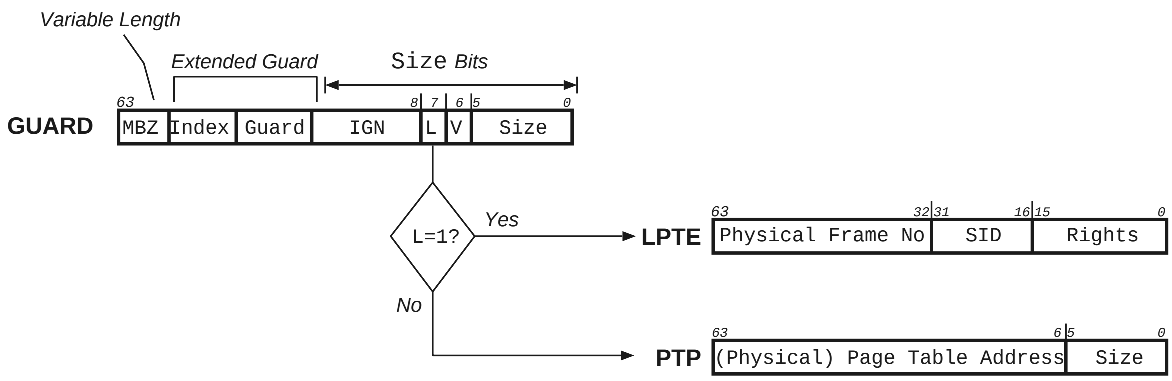


Figure 5: Guarded Page Table Entries

The Nemesis implementation of GPTs on the 21164 makes use of several of the ideas presented in [**Liedtke96b**]. A guarded page table entry (GPTE) comprises two words: the first word is the *guard* word, while the second word is either a page table pointer (PTP) or a leaf page table entry (LPTE). These structures are illustrated in Figure 5.

The guard word contains four significant parts:

- the *index*, a bit string whose width is $\log_2$ of the size of the current page table,
- the *guard*, a variable length bit string,
- the information bits, currently a 2-bit field with a *valid* bit `V` and a *leaf* bit `L`, and
- the guard size, a 6-bit field `size`. This encodes the guard and index lengths (in bits) in the form 64−`size`.

The index and guard fields combined make up the *extended guard*.

A PTP represents the pair (p, s), where p is the physical address of the next level page table and 2^s is its size (in entries). This is encoded by using the top 58 bits of a word for p and the lower 6 bits to hold s. A consequence of this is that all page tables must be aligned to 64-bit boundaries. If the leaf bit is set in a guard, then the second word holds a LPTE. This holds the physical frame number (32 bits), the stretch identifier (16 bits) and the protection bits as defined in [**Sites92**].

When translating a virtual address (`va`), the procedure is as follows:

1. The first n untranslated bits of `va` are used as an index into the current page table, where n is $\log_2$ of the current page table size. This results in a GPTE.
2. If the GPTE has the `V` bit clear, then the translation is not valid, and is aborted. Otherwise, `va` is XORed with the guard, and the result shifted right by `sz`. If `va` does not match the guard and index of the GPTE, then the results will be non-zero. This causes a guard-fault to be reported. Otherwise the extended guard has matched.
3. If the GPTE has the `L` bit set, then the second word contains the LPTE, and the translation is complete. Otherwise the second word contains a PTP for the next level. This contains the base address and size of the new page table, required for step 1.

Figure 6 illustrates the first stage of this translation process. In subsequent stages, the guard word will always have a zero-filled prefix.

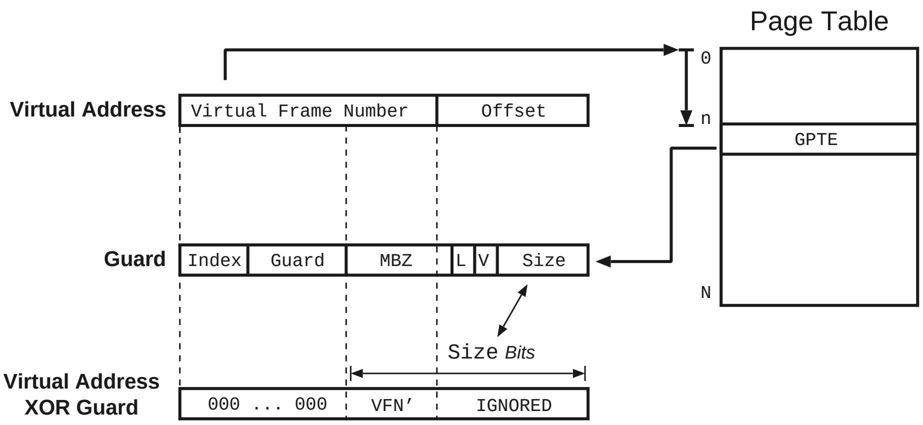


Figure 6: Translating Virtual Addresses

Notice the section now marked `MBZ` in the guard word. By forcing these bits to be zero, the result of a successful guard match leaves the remaining bits of the virtual address untouched. Thus the result of the XOR performed may be used as the input to the next translation stage.

A single page table is used for all mappings, and is not context switched. In addition to mapping (i.e. VA $\rightarrow$ PA) information, each PTE also contains some protection information. This is used on the EB164 to hold the *global* protections for any individual page.

The page table base is kept in an internal processor register and hence is available to the PALcode routines used to handle ITB and DTB misses. It is also used by the non-privileged PALcode calls `map`, `unmap` and `trans` which implement the address translation functions roughly as specified in Section 4.3. The main difference is that physical addresses are not themselves passed to the calls, but rather indices into the caller's frame stack. This provides a simple way to validate the frames referred to.

5.2 The Protection System

As mentioned above, the global protection information is kept in the page table. Protection domain information is used to augment the rights found there: a TLB fill involves first a page-table lookup (thereby getting the mapping and global protection information), and secondly a check of the protection domain for additional access rights.

This is implemented efficiently on the EB164 by using *stretch identifiers*; i.e. each stretch is assigned a unique identifier. Recall that the protections on every page of a stretch must be the same. This means we only require to keep an array of access rights indexed by stretch identifier. This is currently implemented by a page frame (8K) per protection domain, which can hold information for up to 16384 stretches.

On a TLB miss, the virtual address is first looked up in the page table as described in Section 5.1. Assuming that this succeeds, we have a PTE which contains the PFN, the global protection bits — and the stretch identifier! Using this stretch identifier (SID), we index into the current protection domain and retrieve the access rights. These are ORed into the PTE (and hence may only augment the access rights), and finally the TLB is updated. This procedure is illustrated in Figure 7.

A pointer to the current protection domain is kept, like the page table base, in an internal processor register. Unlike the PTBR, it must generally be context switched. To minimise the effects of the context switches, we make use of the 21164's *address space numbers* (ASNs). A pair of internal processor registers (`itbAsn` and `dtbAsn`) is set to any 7-bit value[7], and any subsequent fill of the TLBs is tagged with this value. The current implementation uses one of these per protection domain. Whenever a context switch occurs, the ASN of the protection domain of the new domain is inserted into `itbAsn` and `dtbAsn`.

This means that the TLBs need not be flushed at all on a protection domain switch; invalid protections stored in the TLB from the previous domain simply result in misses and refills. As a further optimisation, the `ASM` bit is also used. This bit, if set in a TLB entry, allows the entry to cause a hit regardless of the current ASN values. Hence it may be used in Nemesis

[7]Note: although there are two IPRs, the 21164 requires they be set to the same value.

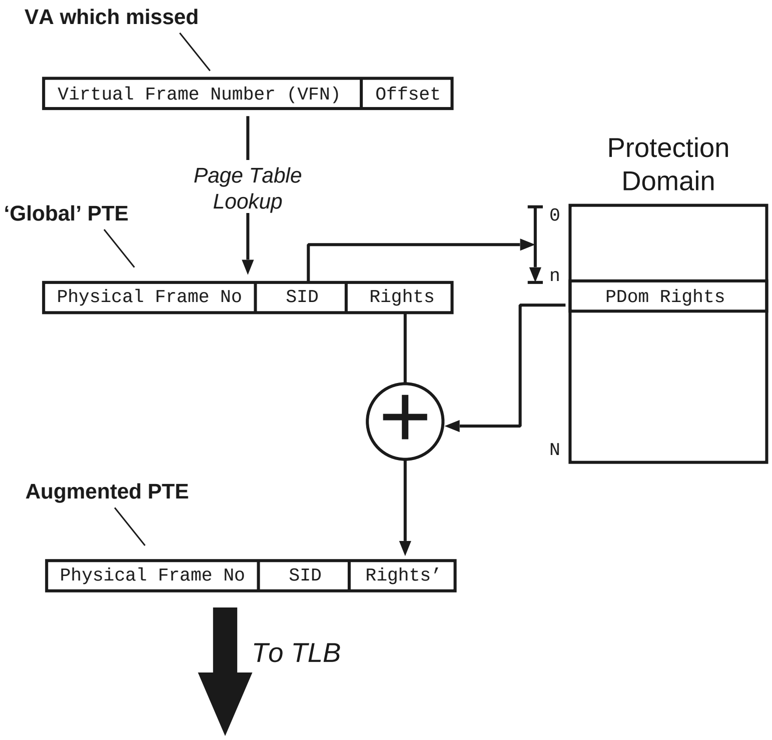


Figure 7: Protection Domain Implementation

to map pages within a stretch which has significant global rights. Care must be taken to ensure that no domain has additional rights for the particular stretch, since this might cause a protection fault where none was necessary.

5.3 Frame Stack

The frame stack is logically composed of two parts:

1. The set of frames allocated to the domain. It is important that the system know which these are so that it can validate mapping attempts.
2. The attributes — clean, dirty, referenced etc. — of these frames. The translation system does not in general need to know about these.

The current implementation of the frame stack is similarly bipartite. The set of frames allocated to any domain, plus their current `mapped` status, are kept in an array mapped read-only to that domain. This array is conveniently located within the domain's DCB, read-only part (DCBRO). Also kept in the DCBRO are the indices (high-water marks) of the number of guaranteed and optimistic frames allocated to the domain.

A second array, indexed in parallel, is kept in the read-write part of the DCB (i.e. the DCBRW). This array contains 16 bits of information for each respective PFN; the format and meaning of this information is entirely up to the domain, but typically contains flags to mark pages as mapped, accessed, dirty, etc.

By laying out the frame stack like this, authenticating mapping attempts becomes easy. When inserting a new mapping, instead of explicitly passing down the frame, the caller gives an index into its frame stack. Thus the call looks like `ntsc_map(va,idx)`. The called routine simply needs to:

a) Range check $0 \leq$ `idx` $\leq$ `xtraf`, where `xtraf` is the high-water mark of the array; and

b) Check the virtual address is within a stretch for which the current protection domain holds *meta* rights.

Assuming the authentication checks are successful, the page-table is walked, the mapping inserted, the frame's `mapped` status updated and the relevant TLB entries flushed.

Initial allocation and subsequent extension of the frame stack are achieved via IDC to the Nemesis domain. Currently, however, there is no proper support for dealing with the revocation of frames — a frame may be "revoked" by deallocating it from a particular domain, but no notification of this is currently given. A frame which is currently mapped cannot be revoked at all.

5.4 Fault Dispatching

The 21164 defines a number of PALcode entry points for faults and interrupts. The entry points relevant to memory management are:

- `IACCVIO`: I-Stream access violation
- `DFAULT`: D-Stream access violation
- `UANLIGN`: D-Stream unaligned access
- `ITBMISS`: I-Stream TLB miss
- `DTBMISS`: D-Stream TLB miss

When any of the above happen, the pipeline is drained, the faulting PC is stored in an internal processor register, and the new PC is loaded by indexing from the start of the PALcode base. The TLB miss handlers load mappings as described above as long as the translation is valid. If a miss handler determines a translation or access fault, or if one of the other three entry points is entered, then the following occurs:

1. The reason for the fault (one of TNV, ACV, FOR, FOE, FOW, UNA or PAGE) is determined.
2. The faulting virtual address is identified. This is typically easy to find from the value of one of the internal processor registers `va` or `excAddr` at the time of the fault.
3. The stretch containing the virtual address is obtained by using a table mapping stretch identifiers to stretches. Clearly this only succeeds for addresses within a currently allocated stretch.
4. The above three pieces of information are stored in the current (i.e. faulting) domain's DCB.

5. A short assembly stub in the NTSC is entered with interrupts off. If the current domain has not yet set an event channel to receive memory management events, or if it has activations off, then it is simply destroyed. Otherwise an event is sent on the domain's memory management event channel, and the scheduler entered.

At some point in the future the faulting domain will be activated. Its event demultiplexing code will call the memory management entry *notification handler* about the event on the memory management channel. This notification handler does the following:

- Blocks the faulting thread.
  All user-level threads packages in Nemesis provide an interface for the use of entries which allows for the blocking and unblocking of individual threads.
- Determines whether the fault is resolvable or not.
  Currently it is considered that ACV, UNA and TNV are always unresolvable; i.e. it is not sensible to talk about retrying the faulting instruction.
  - If the fault is unresolvable, a `todo` structure is filled in with details of the fault (including the faulting thread's *context*). This `todo` structure is put on a queue for a worker thread which is then unblocked.
  - If the fault is potentially recoverable, the stretch driver for the faulting stretch is invoked. If the stretch driver succeeds, the faulting thread is unblocked. If, on the other hand, the stretch driver fails to satisfy the fault, a `todo` structure is used as described above.
- Returns to the event demultiplexer.

Once the notification handler returns, the activation handler will typically pass control to a user-level thread scheduler, which will schedule a memory management worker thread at some point. This thread checks its queue of `todo` structures. If there is no work present, it blocks itself. Otherwise it takes the first item from the queue and determines what action to take. There are two possibilities here:

1. If the fault is an unrecoverable one, the memory management worker thread will invoke the user-level debugger.
2. If the fault is potentially recoverable, the thread invokes the stretch driver for the second time. It is possible that the fault resolution will succeed on this occasion due to the potential for communication with other domains (viz. IDC). This is explained in more detail in Section 5.5.

The structure of the implementation of fault handling is illustrated in Figure 8.

5.5 Stretch Drivers

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

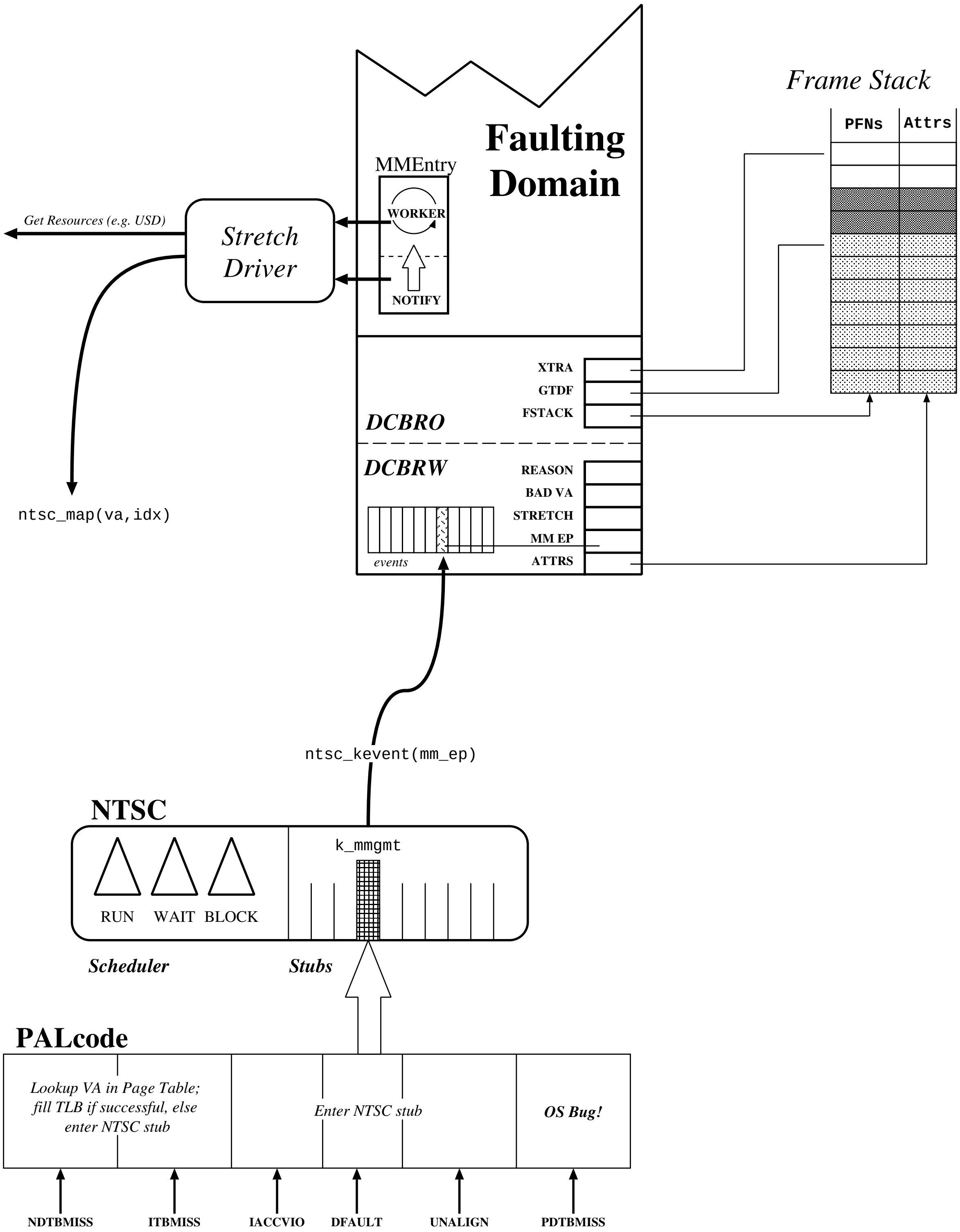


Figure 8: Implementation Structure

is dispatched in the manner described in Section 5.4. The physical stretch driver is invoked from within the notification handler, where the following occurs:

- Basic sanity checks are performed. These include checking that the relevant stretch is bound to this stretch driver, that the faulting virtual address is within the stretch, and that the faulting domain holds *meta* rights for the stretch. If any of these checks fail, the stretch driver returns `Failure`.
- The stretch driver looks for an unused (i.e. unmapped) frame within its portion of the frame stack. 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`.
- The stretch driver sets up the new mapping with a call to `ntsc_map(va,idx,bits)`, where `idx` is the index of the unused frame within the frame stack and `bits` holds the new value for the reference bits, 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 frame allocator via the *FrameStack* interface. 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 in the case where no physical frames are available for mapping, the paged stretch driver has two choices: it may request more frames via the *FrameStack* interface, or it may swap a page out to a backing store. The backing store used currently is a *user-safe disk* (USD) [**Barham96**,**Barham97**]. This allocates resources in terms of *extents* — contiguous ranges of blocks on the disk — and *rate guarantees*. These latter are supported by the use of a custom QoS entry in the driver which schedules requests according to these guarantees.

Each domain making use of the paged stretch driver has a pre-established binding to the USD, owns certain extents (which it may use for paging or other purposes) and holds a particular rate guarantee. Hence it is possible for the paged stretch driver to predictably determine an upper bound on the amount of time required to load a given page from disk. This allows individual stretch drivers to make both disk layout and page replacement decisions which maximise either throughput or capacity as they see fit.

5.6 Evaluation

The prototype implementation has proved the validity of the architecture, but is not yet complete. A number of items remain for future work, the most urgent being:

- Page-Tables.
  Although one might expect GPTs to be especially suitable for a 64-bit single address space, it turns out that in general one achieves *either* small space requirement *or* shallow trees. It seems difficult to get both at once. While there are various theorems on

GPTs [**Liedtke93**, **Liedtke94**] which assert the existence of various minimal trees, the transformations required to achieve them seem rather computationally expensive.

In particular what is lacking is a good heuristic for local transforms when adding or removing entries close at hand. Further work will be required to see if one of these can be discovered, and to investigate other translation structures such as clustered page-tables [**Talluri95**].

- Physical Memory Revocation.
  The idea of 'optimistic' allocation of physical frames is only useful if there is a sensible implementation of some revocation mechanism. Exactly how this should work is not yet clear: the idea of sending a "revocation event" may suffice, but seems rather imprecise.

- Other Architectures.
  The system as implemented here conforms rather closely to the design. This is due at least in part to the fact that the EB164 is a 64-bit RISC processor — precisely one of the target architectures for the Nemesis VM system.

  When implementing the system on the CISC Pentium/Pentium Pro machines, or on the StrongARM, more difficulties arise. For example, the fact that page-tables are walked in hardware provides little flexibility in the translation systems used. Nonetheless good progress has been made in realising a version of the VM system on the other Nemesis platforms.

Additionally there is considerable scope for the development of stretch drivers. Currently three are implemented: a NULL implementation for nailed-down stretches, a demand-paged stretch driver which uses only physical memory, and an extended version of this which also pages to and from the USD. All of these require optimisation and might benefit from some reorganisation. A number of other 'interesting' stretch drivers (providing support for DSVM or peristence, for example) might also be implemented.

6 Related Work

The idea of using a single address space (SAS) within an operating system is not new — indeed the model is found as far back as OS/360 [**Witt66**]. And with the emergence of 64-bit microprocessors and architectures such as the DEC Alpha [**Sites92**], the HP PA-Risc [**HP94**], the Mips R4600 and the IBM PowerPC, many researchers have taken a new look at the SAS idea resulting in operating systems such as Angel [**Murray93**], Opal [**Chase94**] and Mungi [**Heiser94**].

The Angel system was developed by the Software Architecture Research Center (SARC) at City University, London. It is "designed as a generic parallel and distributed operating system", [**Angel**], and hence has support of distributed shared virtual memory (DSVM) as a major goal. It is based on a μ-kernel architecture which provides persistent virtual memory and a local RPC mechanism based on shared memory. The initial prototype implementation of Angel was an emulation running under SunOS. I am not aware of any subsequent native implementation.

Opal was developed by the Department of Computer Science, University of Washington. It provides dynamically variable size *segments* as a basic abstraction, and makes use of password capabilities [**Anderson86**] for access control. Threads explicitly *attach* to segments, presenting a capability for validation. Opal makes use of reference counts to automatically delete a segment after the last *detach* operation. It also supports explicit reference counting via a set of primitives [**Chase95**, pp. 49–50]. These may be used to prevent automatic deletion and hence provide persistent segments.

An implementation of Opal was produced running as a server task on top of the Mach μ-kernel. Segments were represented somewhat transparently by Mach *memory objects*, and a modified version of the Mach default pager was used to provide the backing store.

Mungi is an operating system which has been developed mainly by the School of Computer Science and Engineering at the University of New South Wales, Australia. Its main aim appears to be the support of persistent objects [**Elphinstone96**]. These are protected via a password capability scheme similar to that used by Opal, although in Mungi capabilities need not be presented explicitly to objects. Instead a cache of capabilities is maintained per task in its *active protection domain*.

The most recent implementation of Mungi is on top of the L4 μ-kernel for the Mips R4600 processor [**Heiser97**]. The Mungi API is represented as an L4 user-level server. Mungi system calls are implemented as lightweight IPC to this server, which can then either perform the system call itself or (in some cases) forward it to the L4 μ-kernel. This L4 server also incorporates a default pager for all Mungi tasks.

Nemesis is also a single address space operating system, yet its motivation for this is almost entirely orthogonal to that of these other systems:

- Persistence is not a major goal of the system, although it is of course possible to implement persistent objects (or whatever) at user-level on top of the architecture presented here.
- Distributed shared virtual memory (DSVM) is also not a design goal. While DSVM may in some cases provide a convenient abstraction, it does not scale and furthermore may unduly obfuscate the performance of certain applications. Once again, DSVM may be supported at user-level if it is desired.

As there is no built-in support for persistence, there is no need for reference counting to determine if there are outstanding users of a particular stretch. Memory resources allocated to a domain may be explicitly freed at any time, and are implicitly destroyed when the domain exits. This does not preclude the use of user-level stretch driver which maintains its own state on a backing store across reboots. It simply means that there are no assumptions built in regarding the use (or usefulness) or persistent memory mapped objects.

Furthermore, due to the non-distributed nature of the address space, the use of capabilities for access control is unnecessary. Rather than hide the unwieldly nature of explicit capabilities like Mungi, Nemesis does not use them at all. Clearly were a DSVM-style stretch driver to be implemented, it would require some form of authentication and identification mechanism. However this is a problem which should be solved by that particular stretch driver, not by the entire operating system.

Another aspect of the Nemesis memory management system which might be compared with other work is the removal of all *paging* operations from the kernel. Instead the NTSC is simply responsible for dispatching fault notifications.

At first this may sound rather similar to the ideas pioneered in Mach [**Young87**] and subsequently used in μ-kernel systems such as L4 [**Liedtke95b**]. However it is considerably different from such μ-kernel architectures, or even from "exokernel" systems such as Aegis [**Engler95**]:

- Other systems *allow* the use of one or more external (i.e. non-kernel) pagers in order to provide flexibility/extensibility, to simplify the kernel or to support some higher-level "library operating system".
  However several applications will typically still share an external-pager, and hence the problem of QoS crosstalk remains.
- In Nemesis we *require* that every domain 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 is completely independent to that of other domains save when interaction is desired.

The difference between previous approaches and the Nemesis one is illustrated in Figure 9.

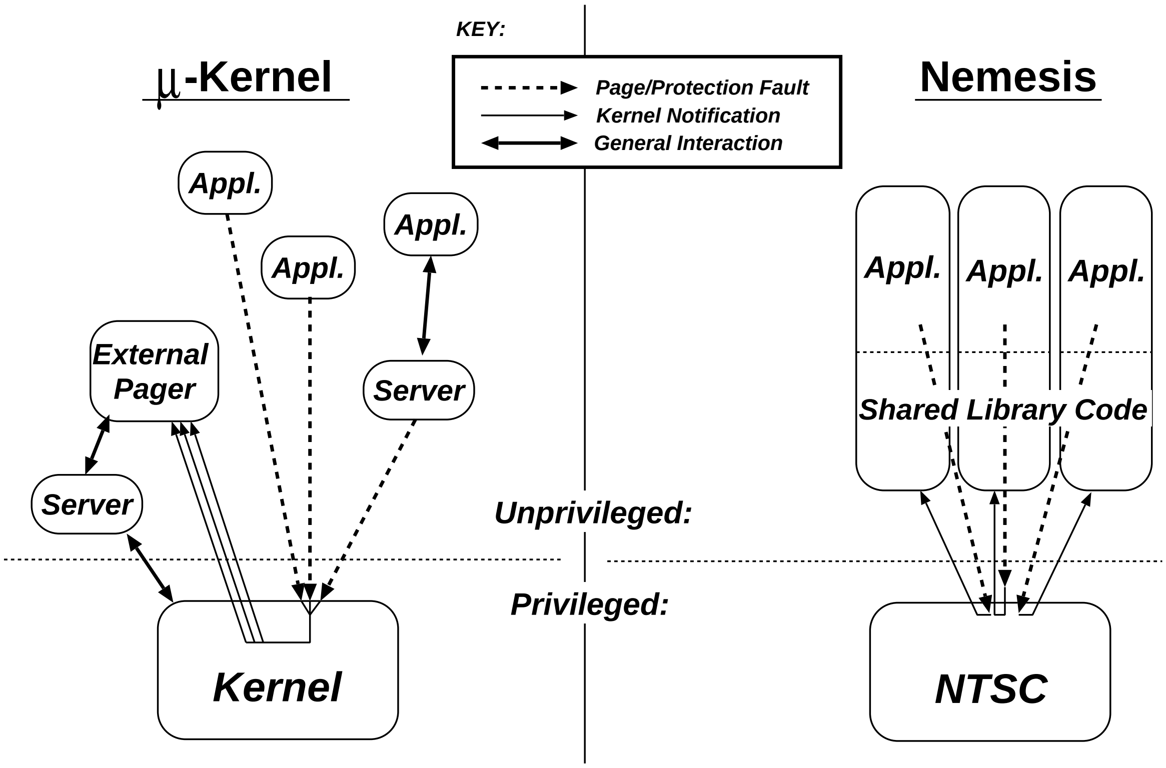


Figure 9: External Paging versus Self Paging

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:

- 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, then, degrades the overall system performance but bears only a fraction of the cost.
- 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 NTSC. However rather than sending a notification to some external pager, the NTSC 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.

7 Summary

This document has presented the architecture, design and initial implementation of the virtual memory system for the Nemesis operating system. The key concepts of isolation and accountability lead to an architecture in which most memory management operations are carried out by each domain for itself. There is no "system pager" or "user-level pager" as in typical monolithic or μ-kernel systems: instead domains are all *self-paging*.

The role of the system is merely to allocate resources and maintain potentially critical information within system-wide tables. Protection and translation are simply relations stored within these tables and generally are publicly readable. Distribution and persistence may be implemented at user-level if desired.

A prototype implementation has demonstrated the validity of the architecture, but is by no means yet complete. There is still a considerable amount of work to be done, in particular with regard to translation structures, supporting CISC architectures, and the difficult problem of resource revocation.

References

[**Anderson86**] M. Anderson, R. Pose, and C. S. Wallace. *A password-capability system.* The Computer Journal, 29(1):1–8, 1986. (p 24)

[**Anderson92**] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.* ACM Transactions on Computer Systems, 10(1):53–79, February 1992. (p 1)

[**Angel**] *Angel.* `http://web.cs.city.ac.uk/research/sarc/angel.html`. (p 23)

[**Barham96**] Paul Barham. *Devices in a Multi-Service Operating System.* PhD thesis, University of Cambridge Computer Laboratory, July 1996. Available as Technical Report no. 403. (pp 2, 22)

[**Barham97**] Paul 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. (p 22)

[**Chase94**] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. *Sharing and Protection in a Single Address Space Operating System.* ACM Transactions on Computer Systems, 12(4):271–307, November 1994. (pp 2, 23)

[**Chase95**] Jeffrey S. Chase. *An Operating System Structure for Wide-Address Architectures.* PhD thesis, Department of Computer Science and Engineering, University of Washington, August 1995. (p 24)

[**DEC95**] Digital Equipment Corporation. *Alpha 21164 Microprocessor Hardware Reference Manual*, April 1995. Order Number EC-QAEQA-TE. (p 14)

[**Elphinstone96**] Kevin Elphinstone, Stephen Russell, Gernot Heiser, and Jochen Liedtke. *Supporting Persistent Objects Systems in a Single Address Space.* Technical Report UNSW-CSE-TR-9601, University of New South Wales, February 1996. (p 24)

[**Engler95**] Dawson Engler, Sandeep K. Gupta, and Frans Kaashoek. *AVM: Application-Level Virtual Memory.* In Processings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V), Rosario Resort, Orcas Island, Washington, USA, May 1995. (p 25)

[**Hayter94**] Mark Hayter. *DAN Processor Node: The MDH-9. Design and Ideas.* In ATM Document Collection 3 (The *Blue* Book), chapter 32. University of Cambridge Computer Laboratory, March 1994. (p 14)

[**Heiser94**] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochteloo. *Mungi: A Distributed Single Address-Space Operating System.* In Proceedings of the 17th Australasian Comp. Sci. Conf, pages 271–280, January 1994. Available as Technical Report UNSW-CSE-TR-9303. (pp 2, 23)

[**Heiser97**] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. *Implementation and Performance of the Mungi Single-Address-Space Operating System.* Technical Report UNSW-CSE-TR-9704, University of New South Wales, June 1997. Available from `http://www.cse.unsw.edu.au/~disy/papers.html`. (p 24)

[**HP94**] Hewlett Packard. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 3rd ed. edition, 1994. (p 23)

[**Hyden94**] Eoin Hyden. *Operating System Support for Quality of Service*. PhD thesis, University of Cambridge Computer Laboratory, February 1994. Available as Technical Report No. 340. (p 1)

[**Koldinfer92**] Eric J. Koldinfer, Jeffresy S. Chase, and Susan J. Eggers. *Architectural Support for Single Address Space Operating Systems*. In Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), Boston, Massachusetts, USA, pages 175–186, October 1992. (p 14)

[**Leslie96**] 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. (p 1)

[**Liedtke93**] Jochen Liedtke. *Some Theorms About Guarded Page Tables*. Technical Report 792, German National Research Center for Computer Science (GMD), October 1993. (p 23)

[**Liedtke94**] Jochen Liedtke. *Some Theorms About Restricted Guarded Page Tables*. Technical Report 834, German National Research Center for Computer Science (GMD), April 1994. (p 23)

[**Liedtke95a**] Jochen Liedtke. *Address Space Sparsity and Fine Granularity*. ACM Operating Systems Review, 29(1):87–90, January 1995. (p 14)

[**Liedtke95b**] Jochen Liedtke. *On μ-Kernel Construction*. In Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles, Operating Systems Review, pages 237–250, December 1995. (p 25)

[**Liedtke96a**] Jochen Liedtke. *L4 Reference Manual*, April 1996. Version 2. Copy available from author on request. (p 2)

[**Liedtke96b**] Jochen Liedtke and Kevin Elphinstone. *Guarded Page Tables on Mips R4600 Or An Exercise in Architecture-Dependent Micro Optimization*. ACM Operating Systems Review, 30(1):4–15, January 1996. A longer version is available as UNSW Technical Report UNSW-CSE-TR-9503. (p 14)

[**Murray93**] K. Murray, A. Saulsbury, T. Stiemerling, T. Wilkinson, P. Kelly, and P.E. Osmon. *Design and Implementation of an Object-Orientated 64-bit Single Address Space Microkernel*. In 2nd USENIX Symposium on Microkernels and other Kernel Architectures, pages 31–43, 1993. See also `http://web.cs.city.ac.uk/research/sarc/angel.html`. (pp 2, 23)

[**Roscoe94**] Timothy Roscoe. *Linkage in the Nemesis Single Address Space Operating System*. ACM Operating Systems Review, 28(4):48–55, October 1994. (pp 2, 6)

[**Roscoe95**] Timothy Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, University of Cambridge Computer Laboratory, April 1995. Available as Technical Report No. 376. (pp 1, 2)

[**Shapiro96**] Jonathon S. Shapiro, David J. Faber, and Jonathon M. Smith. *The Measured Performance of Fast Local IPC.* In Proceedings of 5th International Workshop on Object-Orientation in Operating Systems, Seattle, Washington, USA, 1996. Also available as `http://www.cis.upenn.edu/~eros/devel/iwooos96-ipc.ps`. (p 2)

[**Sites92**] Richard L. Sites, editor. *Alpha Architecture Reference Manual.* Digital Press, 1992. (pp 16, 23)

[**Talluri95**] Madhusudhan Talluri, Mark Hill, and Yousef Khalidi. *A New Page Table for 64-bit Address Spaces.* In Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles, Operating Systems Review, pages 184–200, December 1995. (p 23)

[**Witt66**] B. I. Witt. *The Functional Structure of OS/360 Part II: Job and Task Management.* IBM Systems Journal, 5(1):12–29, 1966. (p 23)

[**Young87**] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathon Chew, William Bolosky, David Black, and Robert 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, Operating Systems Review, pages 63–76, November 1987. (pp 2, 25)