The DASH Virtual Memory System

David P. Anderson
Shin-Yuan Tzou
G. Scott Graham

November 8, 1988

ABSTRACT

The DASH project has defined the network communication architecture for a large, high-performance distributed system. We are now designing a portable operating system kernel for the nodes of this system. The kernel is designed to run on shared-memory multiprocessors, and to exploit the performance potential of such machines.

This report describes the DASH kernel's virtual memory (VM) system. The following are key features of the VM system:

- A virtual address space is partitioned into three *regions*, each providing a specific function: 1) private memory, 2) read-only shared memory, and 3) interprocess communication (IPC) buffers.
- The IPC region uses VM remapping to provide data movement between virtual address spaces. Software copying is minimized.
- Tasks such as page zeroing and pageout are done by processes that can execute concurrently with other activities.
- Most of the VM system implementation is machine-independent. The interface of the machine-dependent part is designed to allow efficient implementation on a range of architectures.

Sponsored by the California MICRO program, AT&T Bell Laboratories, Digital Equipment Corporation, IBM Corporation, Olivetti S.p.A, and the Defense Advanced Research Projects Agency (DoD) Arpa Order No. 4871. Monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089.

1. INTRODUCTION

The DASH project has defined a network communication architecture for large, high-performance distributed systems [31]. We are now designing an operating system kernel for the nodes of this system. The DASH kernel implements the network communication architecture, and allows services and their clients to run in protected virtual address spaces (VAS's). This report describes the design of the DASH kernel's virtual memory (VM) facility. The remainder of the kernel design is described in separate reports [30, 31].

The report is organized as follows: Section 2 lists the goals and assumptions of the VM system design, Section 3 is an overview of the VM facilities, Sections 4 through 6 describe the three regions of a VAS, and Section 7 describes the interfaces to the machine-dependent part of the system and to backing store services. Sections 8 through 10 document the VM system implementation. Section 11 describes the organization of the kernel VAS. Section 12 summarizes our design decisions and the reasoning behind them.

2. GOALS AND ASSUMPTIONS OF THE VM SYSTEM DESIGN

2.1. Functionality Requirements

The DASH VM system provides the following features:

- A mechanism, based on VM remapping, is provided for moving bulk data between VAS's (user/kernel and user/user). The efficiency of this mechanism is significantly greater than that of software copying. This allows data servers (e.g., file servers) to be implemented as user-level processes without a large performance loss.
- Large sparse VAS's are supported, with no expense for unused portions of the VAS and little expense for the non-resident portions.
- The VM system provides a pageable kernel VAS. This is needed because the DASH kernel caches information on remote services, owners, and hosts. If a host interacts with numerous other hosts and owners, these caches may become too large for physical memory.

Future versions of the VM system will provide a framework in which user-defined backing store servers can provide single-level store, network shared memory, and transaction management systems. This is not in the current design.

2.2. Software Engineering and Portability Requirements

The VM system is designed to be extensible and maintainable. It has an object-oriented structure that encapsulates data structures, implementation details, and machine dependencies. Rhe VM system is designed to be portable to a range of shared-memory multiprocessors architectures. For example, the design does not require a hardware mechanism for maintaining consistency of translation look-aside buffers (TLB's). Such a mechanism is not present in some architectures [9].

2.3. Assumptions

The DASH VM system design is based on the following assumptions, which are based on current technological trends:

- The average host will have enough physical memory to satisfy the short-term demands of all its processes. Therefore the emphasis on the use of physical memory is shifted from short-term concerns (e.g., LRU page replacement) to long-term concerns (e.g., maintaining frequently-used programs and libraries in physical memory).
- The approximate size of backing store (implemented either with local disk, or a remote server with a disk and a main-memory cache) will increase at about the same rate as physical memory size. Therefore it is feasible to maintain structures in physical memory whose size is proportional to the size of backing store.
- Processor and network speeds will increase faster than bus and memory access speeds. Therefore software memory copying will be a bottleneck in the movement of data between VAS's, and must be replaced by VM remapping where possible.

3. VM SYSTEM OVERVIEW

Figure 3.1 depicts the overall structure of the DASH VM system. The top part of it shows the abstraction provided by the VM system. The system supports multiple *virtual address spaces* (VAS's), each of which consists of three regions (described in Section 3.1). The middle part of the picture shows the core of the VM implementation, which is independent of hardware architectures and backing store services. It consists of a set of objects and processes (see Section 8). The lower part of the picture shows the rest of the implementation. The class `VAS_MD` encapsulates VM hardware architectures. Its implementation depends on hardware, but its interface does not. The class `BACKING_STORE` encapsulates backing store services. Its implementation varies with the nature of backing store services, but its interface does not.

3.1. The Virtual Address Space Abstraction

Process execution (user or kernel) takes place in a VAS. Multiple processes can execute concurrently in a single VAS. There is a single *kernel VAS* on a given host [1], and multiple *user VAS*'s may exist as well. At the lowest level of partitioning, a VAS is divided into *logical pages*, or simply *pages*. The logical page size must be a power-of-two multiple of the hardware page size. At the highest level of partitioning, a VAS is divided into three *regions*, each of which fulfills a particular role:

General region

This region contains data that is private to a VAS, such as stacks, heaps, and memory-mapped files. There is no sharing between VAS's in this region.

Shared Segment region

This region contains shared read-only named segments (e.g., programs, libraries, and databases). Physical pages containing these segments are shared between VAS's, and may be retained even when no VAS is using them.

Inter-Process Communication (IPC) region

---

[1] The kernel VAS is a separate, full-fledged VAS; it is not (as in most current UNIX implementations) a subset of each user VAS, visible only in kernel mode [2, 10].

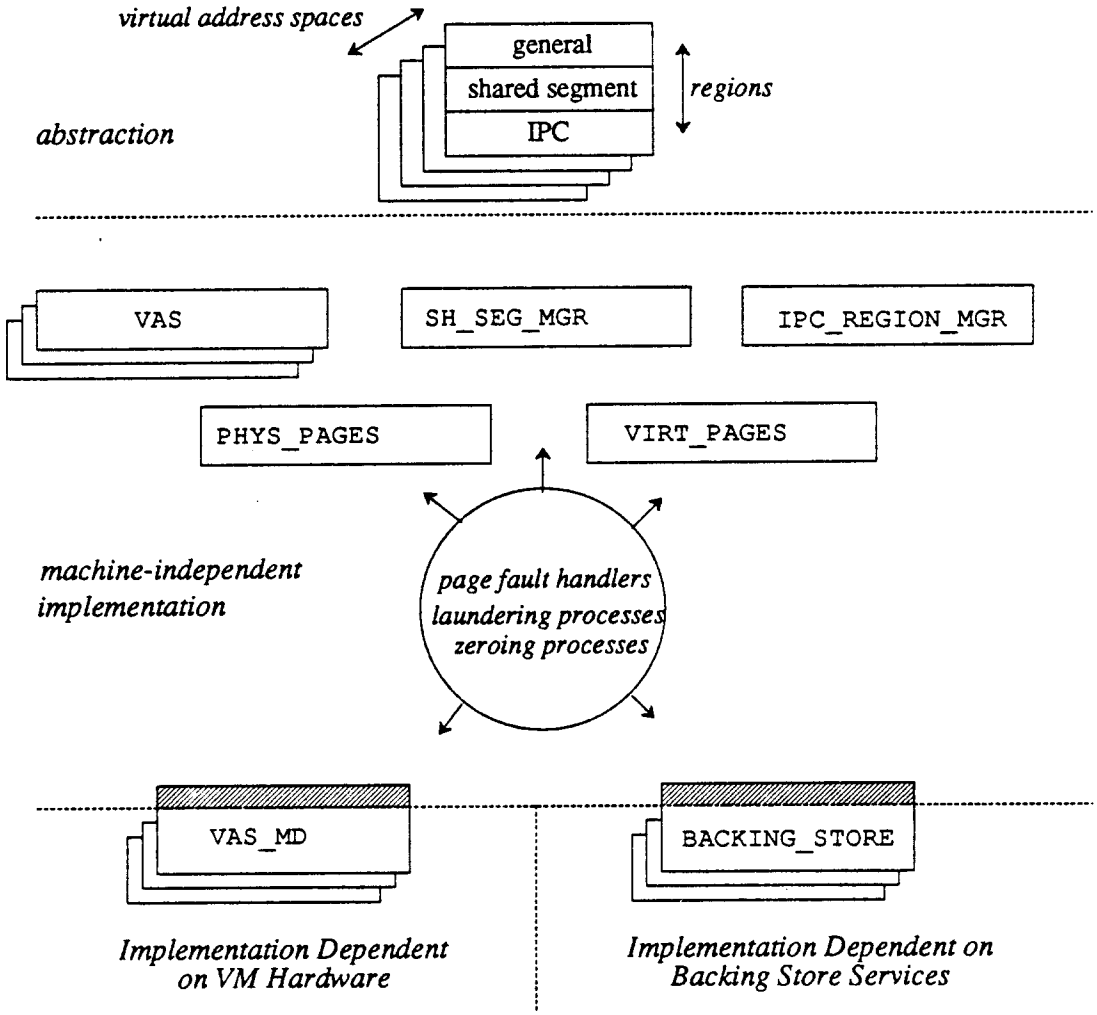

Figure 3.1: An Overview of the DASH VM System.

This region contains data to be moved between VAS's using VM remapping. Inter-space messages are created, sent, and received in this region. This region is shared by all VAS's, subject to protection; each VAS sees a subset of the region via a separate memory map.

Each region occupies the same range of virtual addresses in all VAS's.

3.2. Kernel-Level Structure and Interfaces

The DASH kernel is being implemented in an object-oriented language, C++ [26]. The VM system therefore consists of a set of *objects*, each of which encapsulates data in a procedural interface. The code and data of these objects reside in the kernel VAS, and

operations on them are invoked by performing procedure calls; this can be done directly only by processes in kernel mode. Each object belongs to a C++ *class*. Some classes have only a single state instance; these are called *modules*. Others have multiple dynamically-created instances. For example, a VAS is represented by a dynamically-created object of class `VAS`.

Together, the VM system objects offer an *external* interface supporting the creation, manipulation, and deletion of VAS's. This interface is used directly by kernel processes, and indirectly (via system calls) by user processes. Not all operations on VM objects are part of the external interface; some provide *internal* interfaces accessed only by the VM system objects themselves.

The C++ interfaces in this report use the following machine-dependent types:

```
VIRT_ADDR   // a virtual address
VIRT_SIZE   // a size of a virtual address range
PHYS_ADDR   // a physical address
```

3.3. User-Level Structure and Interfaces

The facilities for local (intra-host) user process communication and kernel requests are described in detail in [30]. Briefly, these facilities are:

- User processes can refer to kernel-level objects by *user object references* (UOR's), which are capabilities issued by the kernel for use by processes in a particular user VAS.
- User processes interact with other processes, and with the DASH kernel, exclusively by *message passing operations*. Both *request/reply* and *stream mode* (unidirectional) operations are supported. These operations are performed by trapping to the kernel, passing 1) a pointer to one or more message buffers, and 2) a UOR to a *message-passing object* (MPO).
- *System calls* are performed by performing request/reply operations on a particular MPO, called the *system call port*, for which a UOR is well-known. These request messages must conform to a fixed set of formats. Most system calls request an operation on (or the creation of) a kernel object. The request message contains a UOR for the object and the arguments to the operation. If the arguments include references to other kernel objects, these are passed as UOR's.

  The system call object dispatches the message to an *interface function* of the appropriate class. This function disassembles the request message, does error checking on the arguments, and calls a regular member function to perform the operation. On return, the interface function prepares a reply message and returns it to the system call object, which returns it to the user process.

3.4. Levels of Memory Mapping

A *mapping* of a VAS describes, for each virtual page in the VAS, 1) the logical-to-physical address translation of the page (if any), and 2) the allowable types of references to the page (read, write, and execute). Various data structures can be used to describe mappings. Some of these structures (e.g., a linked list of descriptors) might be accessed only by software, while others (e.g., a page table) might be used by VM hardware as well.

The DASH VM design uses three levels of VAS mappings:

- *High-level mapping*. This is described by machine-independent objects. Operations on these objects are serialized by locks, so a VAS has a single consistent high-level mapping even on a multiprocessor.
- *Low-level mapping*. This is described by machine-dependent `VAS_MD` objects. It is also consistent between processors.
- *Hardware mapping*. This is described by structures internal to `VAS_MD` objects. On a multiprocessor, each CPU has its own hardware mapping. When a hardware mapping is *in effect* on a processor, it defines the memory locations that the processor can access without a page fault.

For increased performance, the VM system allows two types of inconsistency between the mappings for a VAS. *Vertical inconsistency* is a difference between mappings at different levels. *Horizontal inconsistency* is a difference between the hardware mappings on different processors. These inconsistencies are allowed only if they do not violate the security semantics of the VM system. Sections 6.5 and 7.1 contain further discussion of inconsistencies between mappings.

3.5. Trust Types

Two types of trust appear in the VM system design:

- A VAS can be designated as *locally trusted*. A process in a trusted VAS is assumed to behave correctly (e.g., to only access pages in the high-level mapping of the VAS). Incorrect behavior may not be detected by the VM system.

  The kernel VAS is always locally trusted. The right to create locally trusted VAS's is restricted to kernel processes and user processes in existing locally trusted VAS's.
- When a page is remapped from one VAS to another (see Section 6), the destination VAS may indicate that it *trusts* processes in the source VAS to not write to the page after it has been transferred. If this trust is present, the VM system can defer or eliminate the work of making page inaccessible to processes in the source VAS.

3.6. Constructing VAS Objects

A VAS is created using the following constructor:

```
VAS::VAS(
        OWNER*          owner,
        BOOLEAN         locally_trusted
);
```

For purposes of security, the VAS is associated with the given `owner`. This association is used by the DASH service access and network communication systems (see [31]) for authentication and privacy. It may also be used by the VM system to allow pages from one VAS to be "recycled" for use in a different VAS without zero-filling; for privacy, this is done only if the two VAS's have the same owner.

4. THE GENERAL REGION

The *general region* of a VAS contains its private data (stacks, heap, static variables, memory-mapped files, and so forth). There is no sharing among the general regions of

different VAS's.

At present, the main goal of the general region design is to support large sparse address spaces efficiently. Future versions of the VM system will provide support for user-defined backing store for the general region, supporting applications such as transactional access to memory-mapped files, and network shared memory.

A general region contains a set of disjoint *subregions*, each of which is a contiguous range of virtual pages. A VAS has some preexisting subregions (e.g., for the shared-segment facility described in Section 5). Other subregions are created in response to system calls. A subregion exists until either it is explicitly deleted or the VAS is deleted. A page in the general region that is not part of a subregion is *unallocated*. A reference to such a page generates an exception.[2]

A page in a subregion may be *associated* with a page of data in physical memory or backing store. Allocated pages are not always associated. Hence a process may reserve a large, contiguous region of its VAS without consuming a proportional amount of physical memory or backing store.

Each subregion has a `BACKING_STORE` object (see Section 7.4) that is used by the VM system to store or retrieve pages. This may be either the default system-supplied backing store object, or a client-supplied object.

4.1. Subregion Parameters

Each subregion has a size, a base address, and the following attributes:

```
BOOLEAN          pageable;
BACKING_STORE    *backing_store;
BOOLEAN          zero_fill;
BOOLEAN          associate_on_reference;
```

If `pageable` is true, the VM system can page out associated pages (write them to backing store and reuse the physical pages) at any time. Otherwise they are flushed only in response to the `VAS::flush_subregion()` and `VAS::flush_page()` operations (see below).

If client-supplied backing store is used, dirty pages in the subregion are written to the backing store object when the VAS is deleted. The values of newly-referenced pages are obtained from the backing store object. All pages in the subregion are considered to be associated. The `zero_fill` and `associate_on_reference` flags are not used.

If system-supplied backing store is used, the data in the subregion is discarded when the VAS is deleted. If an unassociated page in the subregion is accessed and the `associate_on_reference` flag is false, an exception is generated. Otherwise the page becomes associated and is assigned an initial value as specified by the `zero_fill` attribute: if true, the value is all zeroes; otherwise the value is unspecified.

In either case, if an operation on the backing store object fails, an exception is generated.

---

[2] If the exception-generating process is in user mode, the process is stopped and an *exception message* is delivered to the *exception MPO* of its VAS (see [30]). If the process is in kernel mode, the system is stopped via a call to `panic()`.

4.2. The General Region Interface

Each subregion is represented by an object of class SUBREGION_DESC. The following operation on VAS objects allocates a subregion:

```
VAS::allocate_subregion(
        int                 npages,
        VIRT_ADDR*          base_addr,
        int                 flags,
        BACKING_STORE*      backing_store
);
```

This creates a subregion in the given VAS consisting of npages contiguous virtual pages. The starting address of the subregion is returned in base_addr. If backing_store is not NULL, it points to a client-supplied backing store object; otherwise system-supplied backing store is used. The Boolean subregion attributes (see Section 4.1) are specified in flags.

```
VAS::deallocate_subregion(
        VIRT_ADDR    base_addr
);
```

This deallocates the subregion starting at base_addr.

```
VAS::associate_page(
        VIRT_ADDR    addr,
        BOOLEAN      zero_fill
);
```

This associates the virtual page at addr with a physical page. Zero_fill determines the content of the physical page.

```
VAS::flush_page(
        VIRT_ADDR    addr,
        BOOLEAN      invalidate
);

VAS::flush_subregion(
        VIRT_ADDR    base_addr,
        BOOLEAN      invalidate
);
```

These cause a page (or all the pages in the subregion) to be flushed to backing store. Pages that are known to be unmodified are not flushed. If invalidate is true, the page(s) become unassociated.

5. THE SHARED SEGMENT REGION

Each VAS has a region containing *shared segments*[3]. Shared segments are read-only[4], and can be used to contain programs, libraries, or read-only databases. Each shared segment has a symbolic name in the DASH global name space [31]. The name consists of the name of a *service* (e.g., a file service) followed by an extension specifying an object

---

[3] The term *segment* simply refers to a contiguous range of virtual space; hardware support for segments is not assumed.

[4] The shared segment facility does not provide write-sharing, e.g., as in Multics [16]. In DASH, multiple processes can share a single VAS, and the general region of this VAS can be used for writing-sharing between those processes.

within the service.

The design goals of the shared segment facility include:

- To provide efficient program loading and execution. Frequently-executed programs should remain in memory, if possible, even while not in use.
- To save space by avoiding duplication of segment contents in physical memory, and by eliminating the need for program files to contain libraries.
- To support dynamic loading of code. This could be used to provide a Cedar-like environment [27] within a protected VAS.

For a shared segment to be accessible to a VAS, it must be *included* in that VAS. When a segment is included in a VAS, a *segment initialization routine* is executed in that VAS. This routine may initialize the segment's private data block (see Section 5.2) and may create processes.

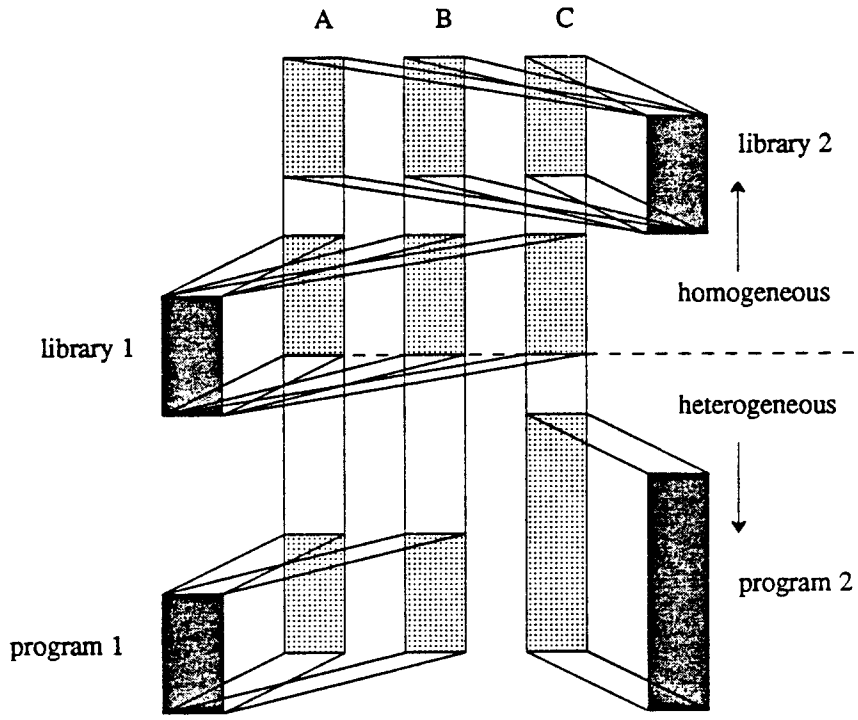

Figure 5.1: Structure of the Shared Segment Region

5.1. The Homogeneous and Heterogeneous Subregions

A shared segment appears at the same address in all VAS's in which it is included. The shared segment region is statically divided into two "subregions" (see Figure 5.1):

- A *homogeneous subregion* in which all VAS's have the same high-level mapping. A segment in this subregion is readable by all VAS's, even those that have not included it. Public libraries and programs can be placed in this subregion.
- A *heterogeneous subregion* in which each VAS has a separate high-level mapping. This subregion must be used for segments to which access is protected. In addition, it provides better error-detection than the homogeneous region, because unintentional references to non-included segments are detected.

5.2. Private Blocks

A shared segment may have an associated *private block* of non-shared read/write memory in each VAS that includes it. The private block might, for example, contain static variables for a program or library; these must be writable, and must have a separate instance in each VAS.

Private blocks for all shared segments are contained in a single general subregion (see Section 4). This *private block subregion* is preallocated in each VAS. Its size is fixed when the system is configured, imposing an upper bound on the total size of private blocks of all shared segments.

The base address and size of a segment's private block is static. Hence, all references to the private block can use absolute addresses. If necessary, the shared segment can obtain additional private memory by allocating a separate subregion at run time. This memory can then be referenced via a pointer in the private block.

5.3. Base Addresses and Inter-Segment References

The base address and size of a shared segment is static, as are the base address and size of its private block. The assignment of addresses is done in advance, perhaps by a central administrator. Within the heterogeneous subregion, different shared segments may be assigned overlapping address ranges. The processes in a VAS are responsible for not including overlapping segments.

Inter-segment references use absolute addresses. Since the base address of a shared segment is assigned in advance, all inter-segment references can be resolved at link time.

5.4. The Shared Segment Region Interface

A shared segment manager object (`SH_SEG_MGR`) manages all shared segments in the system, and has the following interface:

```
SH_SEG_MGR::include_segment(
        VAS*                vas,
        NAMED_ENTITY*       prefix,
        char*               extension,
        U32                 opaque_data
);
```

This includes a shared segment in the given VAS. `Prefix` and `extension` specify the name of the segment (see [31]). `Opaque_data` is passed as an argument to the

segment's initialization routine.

`SH_SEG_MGR::include_segment()` is called in two situations:

- *Internal inclusion*: a process may include a shared segment in its own VAS via a system call. The initialization routine of the segment is called in the context of the requesting process.
- *External inclusion*: a process may include a shared segment in another VAS. This is typically used to include the first segment of a new VAS. The kernel creates a new process in the target VAS to execute the initialization routine. The stack of this process is allocated from the general region of the target VAS; the size is given in the shared segment file.

A segment in the homogeneous subregion is readable in the high-level mapping of all VAS's, but a VAS wishing to access it must first include it using `SH_SEG_MGR::include_segment()`.

```
SH_SEG_MGR::exclude_segment(
        VAS*                vas,
        VIRT_ADDR           loaded_address
);
```

This removes a shared segment from the high-level mapping of a VAS. If the segment is in the homogeneous subregion, this has no effect.

5.5. Programs and Libraries

UNIX-style program execution (creating a new VAS and running the program in it) is done by creating a VAS and including a shared segment externally. Pilot/Cedar-style program execution (loading and running a program in an existing VAS) is done by including a shared segment internally. In this approach, multiple programs may be included from multiple files and executed concurrently in a VAS, as long as their code segments and private blocks do not overlap.

The shared segment facility does not distinguish between programs and libraries. They have the same file format, and use the same inclusion mechanism. A program is responsible for including a library before accessing it. This inclusion is typically done in the initialization routine of the program. (The set of libraries used by a program can be determined at link time.)

5.6. Shared Segment File Format

A shared segment is represented by a *file* (an entry in a service that offers the DASH *file access protocol* [31]). The name of the file serves as the name of the segment.

A shared segment file contains 1) a header, and 2) a segment image. This may be followed by other information, such as symbol tables for the linker and debugger. The structure of the header is

```
struct SH_SEG_FILE_HDR {
        VIRT_SIZE    header_size;
        VIRT_ADDR    segment_addr;
        VIRT_SIZE    segment_size;
        VIRT_ADDR    private_block_addr;
        VIRT_SIZE    private_block_size;
        VIRT_ADDR    init_routine;
        VIRT_SIZE    init_routine_stack_size;
};
```

6. THE IPC REGION

The *IPC region* contains page-size buffers that can be moved between VAS's by VM remapping. This facility is used by the DASH message-passing system [30] when messages are moved between VAS's. The design goals of this facility are:

- To move data between VAS's more efficiently than by software copying [5]. This reduces a potential bottleneck in network communication and I/O, and reduces the performance penalty incurred by placing services at the user level.
- To be portable to a large class of hardware architectures, including shared-memory multiprocessors with no hardware support for TLB consistency.

The IPC region is used only for moving data between VAS's. It is not used for for read/write sharing or for address space inheritance. Forcing it to include these functions would increase its complexity and reduce its performance.

6.1. Access to the IPC Region

A virtual page in the IPC region is called an *IPC page*. All data to be moved between VAS's must be placed in IPC pages. There is a single "meta-level" mapping from IPC pages to physical pages. The high-level mapping of the IPC region in each VAS is a subset of this mapping; some pages may be read-only or not accessible. If multiple VAS's all have a particular IPC page in their high-level mappings, they share the same physical page.

IPC pages are moved between VAS's by changing the access rights to the page in the high-level mappings of the VAS's. Hence "remapping" involves changing protection, not address mappings. An IPC page appears at the same virtual address in both the source and the destination VAS. This reduces overhead in several ways:

- No allocation is done when remapping a page. The virtual page in the destination VAS is predetermined, and is always free.
- Since remapping does not change the virtual address of a page, no pointer adjustment is needed. This simplifies moving structured data, e.g., a message containing pointers to data pages.
- To improve performance, the VM system can include the entire meta-level mapping of the IPC region in the high-level mapping of locally trusted VAS's (e.g., the kernel VAS). If this is done, little work is needed to move a message from a user VAS to the kernel VAS.

---

[5] VM remapping is not necessarily faster than copying; for example, it was slower in Accent [8].

6.2. IPC Page Ownership

Protection of IPC pages is based on the notion of *ownership*. An ownership has three elements: a VAS, a virtual page, and an access right. A page may have multiple ownerships, i.e., it may be owned by multiple VAS's, and by one VAS multiple times. The number of ownerships of a page determines the access right. When the number is one, the page can be both read and written by its owner (i.e., by processes in the owning VAS); otherwise the page can only be read by its owners.

An IPC page may be *transferred* from a source VAS to a destination VAS. This is typically done by a message-passing operation ([30]). When a page is transferred, the source VAS loses an instance of ownership of the page, and the destination VAS gains it.

When multiple VAS's own an IPC page at the same time, they share the corresponding physical page. However, the purpose of this sharing is not for exchanging data between VAS's; the page is read-only for all the VAS's. Instead, this mechanism is used for duplicating a page for retransmission. A typical scenario is as follows: a VAS duplicates the ownership of an IPC page and gives away one ownership. This is done by communication protocols and by services (e.g., name services) that often send out large data that is to remain cached.

6.3. Page Ownership and Hardware Mapping

The VM system use hardware mapping to enforce page ownership. Each VAS has a separate hardware mapping for the IPC regions, except that locally trusted VAS's may share the same mapping.

The cost of changing hardware mappings varies between machine architectures. To unmap a page, the VM system may have to invalidate data in the virtual address cache or remove the page table entry in the memory management unit. In a shared memory multiprocessor, this must be done on all processors on which this page is mapped, and thus requires interprocessor requests and synchronization.

Unmapping a page *asynchronously* may reduce the overall cost because the operations can be batched, and because they can be done when the system is not busy.

The DASH VM system design avoids unmapping pages at the hardware level when possible. When unmapping is needed, it is done asynchronously if possible. The IPC region interface is designed to allow these cases to be discriminated.

The efficiency of unmapping operations is of great concern only in the IPC region. This is because unmapping may be in the critical path of moving a page between VAS's. If it is done synchronously, the efficiency of data movement may suffer. In the general and shared-segment regions, a page is unmapped only when it is paged out, when a VAS is deleted, when a general subregion is deleted, when a general subregion or page is flushed, and when a shared segment is excluded. In all these cases, the unmapping operation can be done asynchronously.

6.4. The IPC Region Interface

An IPC region manager object (`IPC_REGION_MGR`) manages the IPC region of all VAS's, and has the following member functions.

6.4.1. Allocating IPC Pages

```
IPC_REGION_MGR::get_ownership(
      VAS*          vas,
      BOOLEAN       zero_fill,
      VIRT_ADDR*    virt_addr
);
```

This allocates a new IPC page for a VAS and returns the address of the page. A flag specifies whether to zero-fill the newly allocated page.

```
IPC_REGION_MGR::release_ownership(
      VAS*          vas,
      VIRT_ADDR     virt_addr
);
```

This releases an instance of ownership of an IPC page from a VAS.

6.4.2. Transferring IPC Pages

The transfer operation is divided into two parts: a function that starts the operation, and a function that completes it.

```
IPC_REGION_MGR::start_transfer(
      VAS*          source,
      VAS*          destination,
      VIRT_ADDR     virt_addr,
      BOOLEAN       trust
);
```

This starts the transfer operation, which may include an operation that unmaps the page from the source VAS. This function may return before the remapping operation is completed. `Trust` is true if the source VAS believes that the destination VAS trusts it. If so, the unmapping operation is not time-critical, so it can be done in a way which has higher latency but greater total efficiency. For example, on a shared-memory multiprocessor, the operation can be "batched" to reduce the number of interprocessor interrupts.

```
IPC_REGION_MGR::finish_transfer(
      VAS*          destination,
      VIRT_ADDR     virt_addr,
      BOOLEAN       trust,
      BOOLEAN       immediate_use
);
```

This is called by message-receiving operations. It ensures that the destination VAS is protected from security violations, e.g., subsequent modifications by the sender. If `trust` is false, all unfinished unmapping operations of the IPC page are completed (except for those operations that unmap a read-only page). If `immediate_use` is true, the page is mapped into the hardware mapping of the destination VAS.

6.4.3. Sharing IPC pages

```
IPC_REGION_MGR::duplicate(
      VAS*          vas,
      VIRT_ADDR     virt_addr
);
```

This increases the ownership of an IPC page in a VAS by one. If the access right to the page was read/write, it is changed to read-only. This change is made asynchronously in

the hardware mapping of the VAS; Typically, an instance of the duplicated page will be transferred to another VAS later. The `finish_transfer()` function will ensure that the reprotection operation started by `duplicate()` is completed.

```
IPC_REGION_MGR::make_writable(
        VAS*        vas,
        VIRT_ADDR   old_virt_addr,
        VIRT_ADDR*  new_virt_addr
);
```

This makes an IPC page writable; if the page has multiple ownerships, it is copied it to a new page. It returns a pointer to a writable copy of the page.

6.5. Inconsistency in the IPC Region

This section summarizes possible types of inconsistency between IPC page ownership (i.e., high-level mapping) and hardware mapping. By tolerating these inconsistencies, the DASH VM system improves performance without sacrificing security.

- *Deferred mapping*. A VAS may own an IPC page without having it mapped in its hardware mapping. This can be corrected by the page fault handler if the page is accessed. A later hardware unmapping is avoided if the page is transferred out without being accessed.
- *Transient inconsistency*. If the ownership count of a page is one, between `IPC_REGION_MGR::transfer_start()` and `IPC_REGION_MGR::transfer_finish()` the source VAS has lost the ownership of the page, but may still have the page in its hardware mapping. The source VAS may modify the page during this period, but the effect is equivalent to transferring the page after the modification. If the page is read-only, `IPC_REGION_MGR::transfer_finish()` does not finish all asynchronous unmapping operations of this page immediately. The source VAS may read the page after it has lost its ownership, but the contents will necessarily be the same as they were before the transfer.
- *Trust of the source VAS*. If the destination VAS trusts the source VAS, the page may continue to be writable in the source VAS's hardware mapping even after `IPC_REGION_MGR::transfer_finish()` returns.
- *Locally trusted VAS*. The unmapping operation from a locally trusted VAS may be completely omitted. All locally trusted VAS's can share one hardware mapping for the IPC region, and have all IPC pages mapped read/write.

7. MACHINE-DEPENDENT PARTS OF THE VM SYSTEM

This section describes the interfaces to the machine-dependent parts of the VM system. Implementations of these interfaces depend on the VM hardware architecture and on the details of the backing store mechanism; however, the interfaces do not.

7.1. VAS_MD Objects

The class `VAS_MD` encapsulates the machine's hardware-level mapping facility. A `VAS_MD` object represents the low-level mapping for a single VAS. This mapping is specified by a sequence of *map* and *unmap* operations on the `VAS_MD` object.

The difference between a high-level mapping, a low-level mapping, and a hardware mapping was discussed in Section 3.4. This section further explains the difference between a low-level mapping and a hardware mapping.

A processor's hardware mapping depends not only on the VAS in effect, but also on the mode (user or kernel) of the processor. When a processor is in kernel mode, the hardware mapping must contain a set of pages of the kernel VAS that we denote *root pages*. These pages include the code and data accessed by interrupt handlers before switching to and after switching from the kernel `VAS_MD` object. They also include the code and data for `VAS_MD::switch_to()` and `VAS_MD::map()` (see below).

Besides this, the hardware mapping of a processor does not have to be identical to the low-level mapping of the VAS in effect on that processor, but must obey the following rules.

- For a VAS that is not locally trusted, the hardware mapping must be a subset of the low-level mapping.
- For a locally trusted VAS (including the kernel VAS), the hardware mapping must agree with the low-level mappings on all pages where the low-level mapping is defined. The hardware mapping may also contain *superfluous* mapped pages. It is the responsibility of the processes in the VAS to not access these pages.

7.1.1. Creation and Miscellaneous Operations

A `VAS_MD` object has the following operations:

```
VAS_MD::VAS_MD(
        BOOLEAN     locally_trusted,
        BOOLEAN     kernel
);

VAS_MD::switch_to();
```

`VAS_MD::switch_to()` causes the low-level mapping defined by the object to be put in effect on the calling processor.

```
VAS_MD::share(
        int         sharing_id,
        VIRT_ADDR   start_addr,
        VIRT_ADDR   end_addr
);
```

This declares that the given address range is to be associated with the given `sharing_id`, and made available for sharing by other VAS's. This is used for the shared segment region. The `VAS_MD`'s that have called this operation with the same `sharing_id` form a group. A mapping or unmapping operation on any `VAS_MD` object in a group affects the entire group. The `start_addr` and `end_addr` are ignored if the `sharing_id` is already in use. This operation overrides previous declarations on that address range. The `sharing_id` has two special cases: 1) when it is zero, the address range is shared by all `VAS_MD`'s; 2) when it is -1, sharing is canceled for this particular `VAS_MD`.

7.1.2. Mapping and Unmapping Operations

```
VAS_MD::map(
        VIRT_ADDR    virt_addr,
        PHYS_ADDR    phys_addr,
        enum         {READ, WRITE, EXECUTE} type
);
```

This adds the page to the low-level mapping of the VAS_MD object. If the page is shared, this operation affects all VAS_MD objects sharing the page. It also adds the page to the hardware mapping of the calling processor if the VAS_MD object is currently in effect on that processor. The page fault handler normally calls this function before returning from the fault, at a point when the VAS_MD object is in effect on the processor (see Section 10).

```
VAS_MD::synch_unmap(
        VIRT_ADDR    addr,
        BOOLEAN      read_only
);
```

When read_only is false, this removes the page at addr from the low-level mapping of the VAS_MD object. Otherwise it changes the access right to the page from read/write to read-only. If the page is shared, this operation affects all VAS_MD objects sharing the page. If necessary, it also updates hardware mappings to ensure that they are a subset of the low-level mapping. It is "synchronous", and does not return until the necessary changes have been made in the hardware mappings on all relevant processors.

Unmapping operations can also be done asynchronously (Section 6.3 discusses the motivation for this):

```
VAS_MD::asynch_unmap(
        VIRT_ADDR          addr,
        STREAM_MPO*        mp_object,
        MESSAGE*           message,
        BOOLEAN            fast,
        BOOLEAN            read_only
);
```

This is similar to VAS_MD::synch_unmap() except that it may return while the unmapping is still in progress. If so, the given message will be sent to the given message-passing object [6] when the unmapping is completed. If *fast* is false, the operation is not time-critical and can be done in a way that has higher latency but greater total efficiency. For example, the unmapping can be postponed and batched with other unmapping operations.

The VAS_MD class encapsulates machine dependencies, including issues related to multiprocessors. For example, if a low-level mapping is in effect on more than one CPU at once, it may be desirable to maintain, for each page, the set of CPU's on which the page is present in the hardware mapping. If so, this is done by the VAS_MD object rather than in the machine-independent part of the VM system.

---

[6] The MP object may be either dual-process or uniprocess [30]. In the latter case, the send operation is equivalent to a procedure call.

7.1.3. Page Status Operations

The following operations are used to detect modified pages:

```
BOOLEAN
VAS_MD::is_dirty(
        VIRT_ADDR    addr
);

VAS_MD::clear_dirty(
        VIRT_ADDR    addr
);
```

`VAS_MD::is_dirty()` returns true if the page has (or may have) been modified since the time it was added to the low-level map, or since the last call to `VAS_MD::clear_dirty()`, whichever is later.

Currently, the interface does not support the checking and clearing of reference bits.

7.1.4. Configuration Parameters

The following function returns the machine-dependent configuration parameters of the VM system.

```
struct VM_CONFIG {
        VIRT_SIZE    logical_page_size;
        VIRT_SIZE    physical_page_size;
        VIRT_ADDR    user_general_region_addr;
        VIRT_SIZE    user_general_region_size;
        VIRT_ADDR    kernel_general_region_addr;
        VIRT_SIZE    kernel_general_region_size;
        VIRT_ADDR    ipc_region_addr;
        VIRT_SIZE    ipc_region_size;
        VIRT_ADDR    sh_homo_subregion_addr;
        VIRT_SIZE    sh_homo_subregion_size;
        VIRT_ADDR    sh_hetero_subregion_addr;
        VIRT_SIZE    sh_hetero_subregion_size;
        VIRT_ADDR    sh_private_subregion_addr;
        VIRT_SIZE    sh_private_subregion_size;
        BACKING_STORE*     default_backing_store;
};

VM_CONFIG*
VAS_MD::configuration();
```

The machine-independent part of the VM system calls this function once at initialization, and maintains a pointer to the result.

7.2. The Interface to Backing Store Objects

There can be many different providers of backing store. Each is encapsulated in an object whose class is derived from `BACKING_STORE`. The VM system assumes that a backing store service can efficiently support "holes" caused by writing pages sparsely.

Backing store could be provided using a local disk; this particular object would be machine-dependent. Alternatively, backing store can be provided by a network service that supports the DASH file service access protocol. The constructor for this class is

```
SVC_BACKING_STORE::SVC_BACKING_STORE(
        SERVICE_TOKEN*      token
);
```

The `token` is obtained by calling the DASH service access mechanism (see [31]). For user-supplied backing store for a subregion of the general region, a user obtains the token and passes it to the kernel.

```
BACKING_STORE::read_page(
        int                 bs_offset,
        int                 desired_access,    // read, write, execute flags
        VIRT_PAGE_DESC*     page
);
```

This reads a page from the backing store at `bs_offset`. Information about the page (e.g., its physical address and its access rights) is passed and returned in the `VIRT_PAGE_DESC` object (see Section 8.2). The operation is synchronous; it returns after the read operation has completed.

```
BACKING_STORE::write_page(
        int                 bs_offset,
        VIRT_PAGE_DESC*     page
);
```

This synchronously writes a page to the backing store at `bs_offset`.

8. IMPLEMENTATION: OVERVIEW

This section is an overview of the implementation of the DASH VM system. It discusses the object structure of the system, and sketches the implementation of some of these objects. The details of other parts of the system (physical memory management and page fault handling) will be covered in Sections 9 and 10.

The DASH VM system implementation is structured as a set of objects. Some of these objects (`VAS`, `SH_SEG_MGR`, and `IPC_REGION_MGR`) are part of the client interface, and have already been described. Others are internal to the VM system, and are described in the following subsections.

8.1. The Physical Pages Object

The physical pages module (`PHYS_PAGES`) provides a descriptor (`PHYS_PAGE_DESC`) for each page of physical memory. The descriptor is used for various purposes, depending on the state of the page. Most commonly, it stores pointers by which physical pages are linked into lists. The structure of a descriptor is:

```
class PHYS_PAGE_DESC : DLINK {
        VIRT_PAGE_DESC*     vpd;
};
```

The base class `DLINK` contains forwards and backwards links. `Vpd` points to a "virtual page descriptor"; see Section 8.2.

The interface of the `PHYS_PAGES` object is:

```
PHYS_PAGE_DESC*
PHYS_PAGES::lookup(
        PHYS_ADDR    addr
);
```

This locates the `PHYS_PAGE_DESC` for the given physical page.

8.2. The Virtual Pages Object

The virtual pages object (VIRT_PAGES) maintains descriptors (VIRT_PAGE_DESC) for virtual pages in the general or shared-segment region of a VAS and that satisfy either

(1) the page currently has a high-level mapping to a page of physical memory, or

(2) the page is in the general region, is non-resident, and is backed by system-supplied backing store.

A page in the shared segment region may be shared by multiple VAS's, but has at most one VIRT_PAGE_DESC entry. Information on IPC pages is kept in a separate structure (see Section 8.4). The structure of a virtual page descriptor is as follows:

```
class VIRT_PAGE_DESC {
      SPINLOCK     lock;        // lock for this descriptor
      VIRT_ADDR    virt_addr;   // virtual address of the page
      void*        obj;         // pointer to VAS or SH_SEG_DESC object
      PHYS_ADDR    phys_addr;   // physical address, NULL if none
      PAGE_STATE   state;       // values listed below
      DLINK        process;     // sleep queue during paging I/O
};


enum PAGE_STATE {               // see Section 9.1
      IN_USE,
      CLEAN,
      BEING_WRITTEN,
      BEING_READ,
      BEING_UNMAPPED,
      NOT_PAGEABLE,
      PAGED_OUT
};
```

The interface provided by the VIRT_PAGES module is:

```
VIRT_PAGE_DESC*
VIRT_PAGES::create(
      VIRT_ADDR    virt_addr,
      void*        obj
);

VIRT_PAGE_DESC*
VIRT_PAGES::lookup(
      VIRT_ADDR    virt_addr,
      void*        obj
);

VIRT_PAGES::destroy(
      VIRT_PAGE_DESC*    entry
);
```

VIRT_PAGES::create() adds a descriptor for the given *(page, object)* pair (such a descriptor must not exist already). VIRT_PAGES::lookup() returns a pointer to an existing descriptor for the given *(page, object)* pair, and returns NULL if none exists. VIRT_PAGES::delete() deletes a descriptor.

The VIRT_PAGES object is implemented as a hash table. The arguments to the hash function are the virtual address and a pointer to the VAS or SH_SEG_DESC object

containing the page. Collisions are handled by chaining. The size of the hash table is proportional to the size of system-supplied backing store.

8.3. Implementation of the General Region

A VAS object contains information describing the high-level mapping of the general region. The VAS object maintains a linked list of descriptors, one for each allocated subregion. The VAS::allocate_subregion() operation searches for space within the general region that is large enough to accommodate the new subregion. The structure of a subregion descriptor is:

```
class SUBREGION_DESC : DLINK {
        VIRT_ADDR           virt_addr;        // starting virtual address
        VIRT_SIZE           length;           // length in bytes
        int                 attributes;       // pageable, etc.
        BACKING_STORE*      backing_store;    // ptr to backing store object
        DLINK               assoc_pages;      // pages in VIRT_PAGES
};
```

Assoc_pages is a list of VIRT_PAGE_DESC entries for a subregion, and is used during the deletion or flushing of the subregion.

8.4. Implementation of Shared Segments

The shared segment manager object SH_SEG_MGR maintains a table of "active" shared segments. Each shared segment is represented by a descriptor:

```
class SH_SEG_DESC {
        SPINLOCK            lock;
        VIRT_ADDR           start;            // base address
        VIRT_SIZE           size;             // size in bytes
        VIRT_ADDR           init;             // initialization routine
        VIRT_SIZE           stack_size;       // size of stack for init routine
        BACKING_STORE*      backing_store;    // backing store object
        int                 ref_count;        // number of including VAS's
        LOCAL_OWNER*        owners;           // list of authorized owners
                                              // NULL if public accessible
        int                 sharing_id;       // used in VAS_MD::share()
        DLINK*              pages;            // list of virtual page descriptors
                                              // used during deleting a segment
};


SH_SEG_DESC*
SH_SEG_MGR::lookup_segment(
        NAMED_ENTITY*       prefix,
        char*               extension,
        LOCAL_OWNER*        owner
);

SH_SEG_MGR::delete_segment(
        SH_SEG_DESC*        entry
);
```

Each shared segment is assigned a unique sharing_id. It is passed to VAS_MD::share() when the segment is included in a VAS.

SH_SEG_MGR::lookup_segment() checks if the given segment is in the table. If it is, and is authorized for this owner, the segment descriptor is returned immediately. Otherwise an authorization operation is performed on the service token of the segment's BACKING_STORE object. If the segment is not present, a service token is obtained, the segment description data is read, the SH_SEG_DESC object is initialized, and a sharing ID is assigned. Segments in the homogeneous and heterogeneous subregion are handled similarly, except that the sharing_id of a homogeneous segment is always zero. SH_SEG_MGR::delete_segment() deletes a descriptor from the table. It may be called when the segment is not included in any VAS.

Each VAS object maintains a linked list of descriptors of shared segments included in that VAS. The page fault handler searches this list to determine the SH_SEG_DESC object corresponding to the faulting virtual address.

8.5. Implementation of the IPC Region

In the current DASH VM system design, the IPC region is not pageable; i.e., an IPC page is present in physical memory if it is owned by a VAS. This makes the design simpler and more efficient, but it limits the way in which IPC pages can be used. To avoid using unbounded amounts of physical memory, processes must not own unbounded numbers of IPC pages.

Future versions of the VM system may have a pageable IPC region. This will allow processes to store long-lived data structures (such as data caches) in the IPC region, from which they can be transferred to other VAS's efficiently. This change will not affect the IPC region interface.

The IPC region is implemented as follows. An IPC_REGION_MGR object maintains an array of descriptors (IPC_PAGE_DESC) for IPC pages.

```
class IPC_PAGE_DESC : DLINK {
      SPINLOCK                lock;
      int                     ownership_cnt;     // ownership count
      PHYS_ADDR               phys_addr;         // physical address
      IPC_PAGE_OWNER_DESC     owner1;            // room for 2 owners
      IPC_PAGE_OWNER_DESC     owner2;
      DLINK*                  owners;            // list of additional owners
};
```

The owner descriptor (IPC_PAGE_OWNER_DESC) stores the information about ownership of the IPC page by a particular VAS. The IPC page descriptor reserves the room for two owner descriptors, so it is not necessary to allocate and deallocate owner descriptors in most cases. The structure of an owner descriptor is

```
class IPC_PAGE_OWNER_DESC : DLINK {
      VAS*              vas;                  // owner VAS
      int               ownership_cnt;
      BOOLEAN           separate_block;       // for deallocation
      BOOLEAN           free;                 // for allocation
      BOOLEAN           ever_mapped;
      BOOLEAN           being_unmapped_fast;
      BOOLEAN           being_unmapped_slow;
};
```

The last three flags maintain the status of the page in the low-level mapping. IPC_REGION_MGR::start_transfer() calls VAS_MD::asynch_unmap()

to unmap a page. This function may return asynchronously. In such a case, when the low-level unmapping operation is done, a message is delivered to a uniprocess message-passing object maintained by IPC_REGION_MGR. The message handler function updates the flags and finishes the high-level unmapping.

9. IMPLEMENTATION: PHYSICAL MEMORY MANAGEMENT

The physical memory management part of the DASH VM system manages the allocation of physical memory pages. The interface is provided by the PHYS_PAGE_MGR module:

```
PHYS_ADDR
PHYS_PAGE_MGR::allocate_any();

PHYS_ADDR
PHYS_PAGE_MGR::allocate_zero();

PHYS_PAGE_MGR::free(
      PHYS_ADDR   addr
);
```

PHYS_PAGE_MGR::allocate_zero() returns a zero-filled page. Allocate_any() returns a possibly nonzero page. Free() frees a previously allocated page.

To implement this interface, PHYS_PAGE_MGR maintains three *lists* of physical pages (see Figure 9.1). These lists are represented as doubly linked lists using PHYS_PAGE_DESC entries.

- in_use_list: pages that are low-level mapped in some VAS or shared segment.
- clean_list: pages that are not low-level mapped, may be high-level mapped in some VAS, and may be nonzero.
- zero_list: pages that are not high-level mapped in any VAS or shared segment and are zero-filled.

Three types of processes manage the flow of physical pages among the lists (this flow is illustrated in Figure 9.1 and discussed in later subsections):

- When the supply of allocatable pages is below a threshold, an *unmapper* process selects physical pages that are currently low-level mapped. It starts an asynchronous unmapping operation on each page, arranging for a message to be delivered to an *unmap notification port* (the message contains information describing the page). It also removes the page from the in_use_list.
- Multiple *launderer* processes receive messages from the unmap notification port. They write each page to backing store if necessary, then add it to the clean_list.
- Multiple *zero-filler* processes remove pages from the clean_list, zero them, and add them to the zero_list.

9.1. Page States

The following values of the state field of a VIRT_PAGE_DESC entry are relevant to physical memory management and page-fault handling:

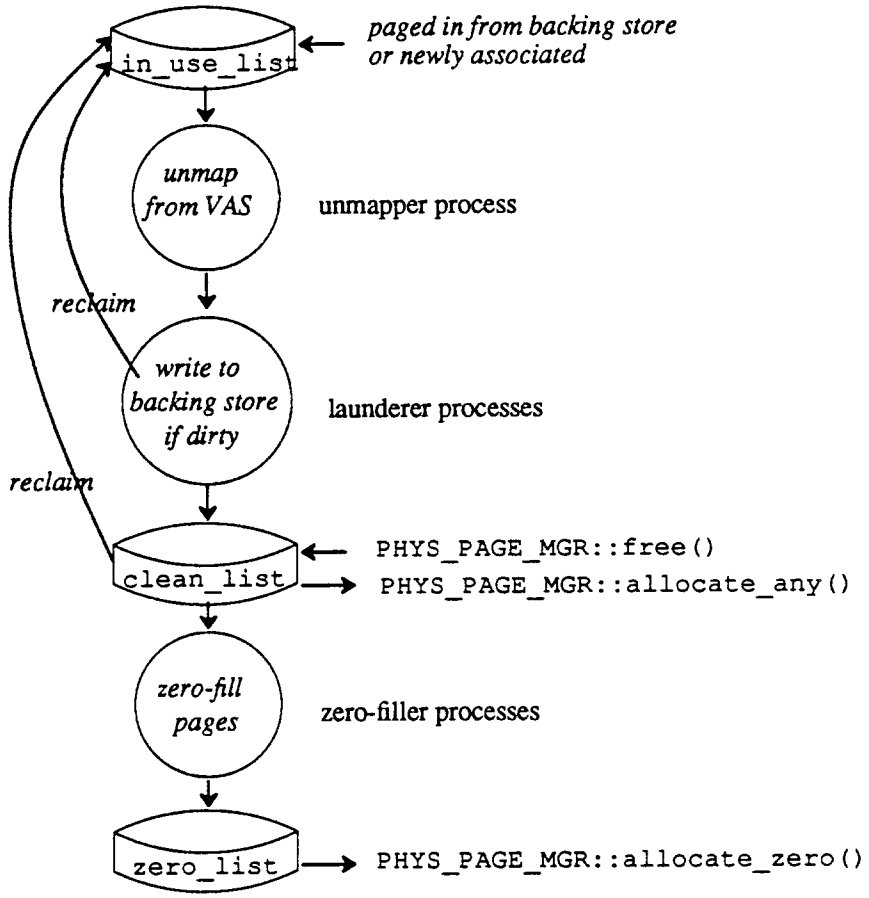

Figure 9.1: The Flow of Physical Pages.

IN_USE: the page is low-level mapped in a VAS.

BEING_READ: the page is being read from backing store.

BEING_WRITTEN: the page is being written to backing store.

CLEAN: the page is not low-level mapped, but it is still high-level mapped and its contents have not been changed since it was low-level unmapped. Hence it can be low-level

mapped on reference.

PAGED_OUT: the page has been paged out to backing store.

NOT_PAGEABLE: the page is not pageable.

BEING_UNMAPPED: an asynchronous unmap operation is pending on the page.

If a virtual page has no VIRT_PAGE_DESC entry, its state is considered PAGED_OUT, except when it is in a subregion with a system-supplied backing store object. The transitions between states are shown in Figure 9.2. The remainder of this section concentrates on transitions not initiated by page faults; Section 10 treats those transitions in more detail.

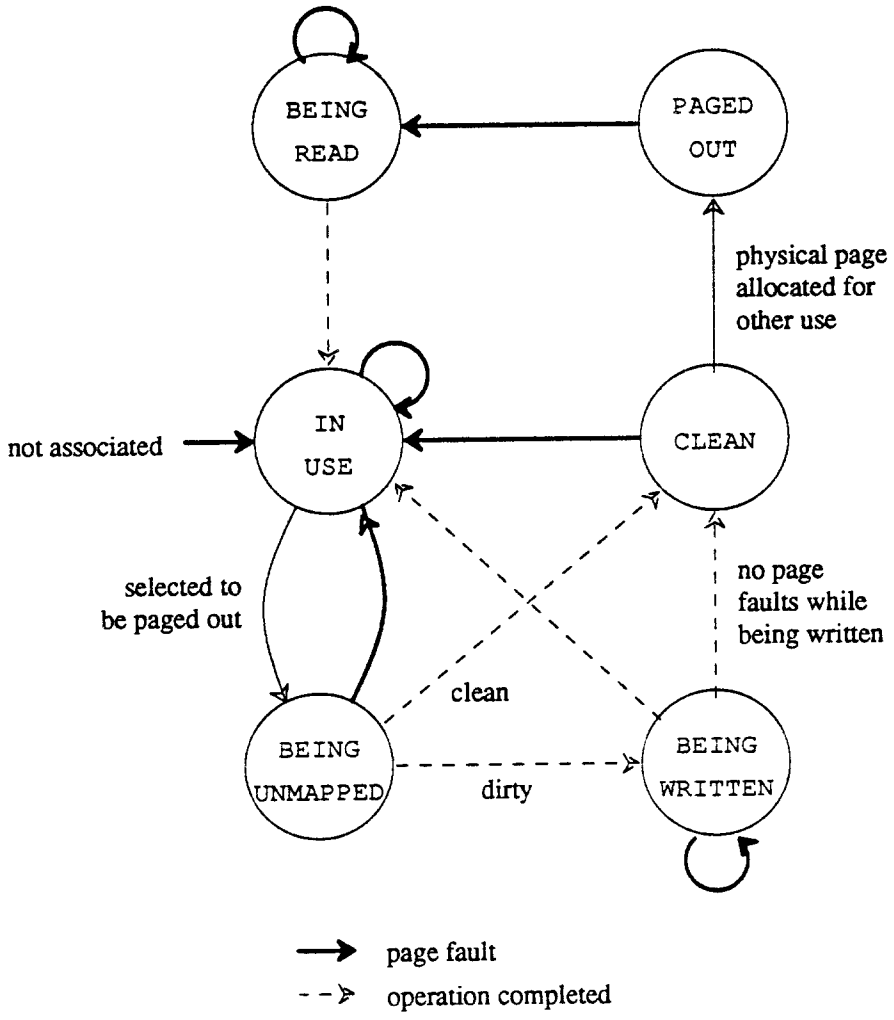

Figure 9.2: State Transitions of Virtual Pages.

9.2. The Unmapper Process

Many algorithms are possible for selecting virtual pages to be unmapped, including the UNIX clock algorithm [2], the VMS second-chance FIFO algorithm [14], a working-set algorithm [6], or a hybrid algorithm [4]. We assume that, because of large physical memory sizes, page replacement will be infrequent. Attempts to make intelligent choices (e.g., to approximate LRU) will not be needed; a random choice will probably be sufficient. It is possible that other readily available information (such as the recent CPU usage of processes in the VAS) may be useful in making a heuristic choice.

For each page selected, the unmapper process locates the corresponding `VAS_MD` object, and changes the state in the `VIRT_PAGE_DESC` from `IN_USE` to `BEING_UNMAPPED`. It performs a `VAS_MD::asynch_unmap()` operation with the *slow_unmap* flag set. This operation will asynchronously unmap the page on all processors, and send a message to the unmap notification port. The message includes the page address, and pointers to the `VIRT_PAGE_DESC` and `VAS_MD` objects.

9.3. Launderer Processes

A *launderer* process receives unmap notification messages. If the state of the page is `IN_USE`, the page was reclaimed during the unmap operation, and the message is ignored. Otherwise, the state of the page is `BEING_UNMAPPED`. `VAS_MD::is_dirty()` is called to see if the page has been modified. If not, the state is set to `CLEAN` and the page is moved to the `clean_list`. If so, the page is "laundered" by setting its state to `BEING_WRITTEN`, and writing it to backing store.

When the write to backing store finishes, the process queue in the `VIRT_PAGE_DESC` is checked. If it is empty (meaning that the page was not referenced while being written), the state is set to `CLEAN`, and the page is moved to the `clean_list`. Otherwise the state is set to `IN_USE` and the sleeping processes are awakened.

To allow multiple backing store write operations to proceed concurrently, there are multiple launderer processes.

9.4. Zero Filler Processes

Each CPU has one *zero-filler process*. Each process executes a loop in which it removes a page from the `clean_list`, zeroes it, adds it to the `zero_list`, and deletes its `VIRT_PAGE_DESC` entry.

10. IMPLEMENTATION: PAGE FAULT HANDLING

The page fault handler is a kernel routine that handles page fault traps. It is executed on the same processor as the faulting process. When invoked, it has the following information available to it:

```
VIRT_ADDR   virt_addr;  // the virtual address of the faulting access
ACCESS_TYPE type;       // type of memory access (read, write, execute)
VAS*  vas;              // the VAS currently in effect
```

The general sources of page faults are:

- Reference to an invalid address (i.e., one not contained in the high-level mapping).

- Reference to an address that is valid, but whose page is not in the low-level mapping of the VAS. This occurs, for example, when the page has been paged out to backing store.
- Reference to an address that is valid, and whose page is in the low-level mapping of the VAS, but is not in the hardware mapping of the processor generating the fault. This occurs 1) when the hardware mapping was unable to accommodate the low-level mapping, and 2) on a multiprocessor, when the reference occurs on a different processor than the earlier `VAS_MD::map()`.

The details of page fault handling depend on which of the three regions contains `virt_addr`. Initial error checking and processing are done as follows:

- If `virt_addr` is not in any region, an exception is generated.
- If `virt_addr` is in the IPC region, the corresponding `IPC_PAGE_DESC` object is checked. If `VAS` is on the owner list and the type of request is valid (i.e., it is not a write request on a page with read-only ownership), `VAS_MD::map()` is called to establish the hardware-level mapping. Otherwise, an exception is generated.
- If `virt_addr` is in the general region, the list of subregion descriptors in the `VAS` object is searched. If a `SUBREGION_DESC` containing `virt_addr` is found, processing continues on the `VIRT_PAGE_DESC` object (see below). Otherwise, an exception is generated.
- If `virt_addr` is in the shared segment region, an exception is generated if the access type is `write`. The rest of the processing is similar to the general-region case, except that the list of shared segment descriptors in the `VAS` object is searched.

In the last two of these cases, if no error has been detected then the page fault handler calls `VIRT_PAGES::lookup()` to obtain a `VIRT_PAGE_DESC` for the faulting address. The arguments are the `virt_addr` and either the `VAS` pointer (if in the general region) or the `SH_SEG_DESC` pointer (if in the shared segment region). Subsequent actions depend on whether the `VIRT_PAGE_DESC` entry is present.

10.1. Case 1: VIRT_PAGE_DESC Entry Absent

We first suppose that the `VIRT_PAGE_DESC` entry is absent. This is the case if the page was paged out or has never been referenced. Then the following steps are done:

(1) A `VIRT_PAGE_DESC` entry is created and locked.

(2) If `virt_addr` is in a subregion with a system-supplied backing store object and the `associate_on_reference` flag is set, a physical page is allocated (zero-filled if the `zero-filled` flag is set). The state of the page is set to `IN_USE`. If the `pageable` flag for the subregion is set, the page is placed on the `in_use_list`. If the `associate_on_reference` flag is false, an exception is delivered.

(3) Otherwise, the page is read from backing store. The page state is set to `BEING_READ`, and the backing store object (pointed to by the `SUBREGION_DESC` or `SH_SEG_DESC` object) is invoked to synchronously read the page. The faulting process blocks during the read operation. The `VIRT_PAGE_DESC` entry is unlocked before the read is started, and locked again

after the read is completed. If the call fails, an exception is delivered to the process. The page state is then set to IN_USE, and all processes on the sleep queue of the VIRT_PAGE_DESC entry are awakened. If the virt_addr is in the shared segment region or in a subregion with the pageable flag set, the page is placed on the in_use_list.

(4) The hardware mapping is restored by performing a map() operation on the VAS_MD object.

10.2. Case 2: VIRT_PAGE_DESC Entry Present

We now suppose that the VIRT_PAGE_DESC entry is present. The entry is locked; subsequent actions depends on the state:

PAGED_OUT: the procedure for reading in the page is identical to that described in the previous subsection. Usually, when a page is not in physical memory, its VIRT_PAGE_DESC entry is deleted. The entry is kept only for pages in a subregion with a system-supplied backing store object.

IN_USE: no action is taken.

BEING_READ: the faulting process sleeps on the VIRT_PAGE_DESC entry. Neither the page state nor the list it is on changes. The process will be awakened on I/O completion.

BEING_UNMAPPED: the state is set to IN_USE. The page is added to the in_use_list. This is an example of "reclaiming" a page (Figure 9.1).

BEING_WRITTEN: the faulting process sleeps on the VIRT_PAGE_DESC entry; it will be awakened on completion of the page-out I/O. Neither the page state nor the list it is on changes.

CLEAN: the state is set to IN_USE. The page is removed from the clean_list and added to the in_use_list. This is another example of reclaiming a page.

In each case, before returning from the page fault, the mapping is restored by performing a map() operation on the appropriate VAS_MD object.

11. ORGANIZATION OF THE KERNEL VAS

Like other VAS's, the kernel VAS consists of three regions. Currently, the kernel does not use the shared segment region, although it could do so without difficulty. The IPC region is used in the same way as other VAS's.

The general region of the kernel VAS contains at least the following three subregions:

(1) *Static subregion*: not pageable. This contains the kernel's code and static data. Physical pages are allocated during system initialization.

(2) *Resident data subregion*: not pageable, associate_on_ref. This contains dynamically allocated data to which access is time-critical (i.e., that should not be paged out).

(3) *Pageable data subregion*: pageable, associate_on_ref. This contains dynamically allocated data to which access is not time-critical.

Each subregion is defined by a page-aligned virtual address, a size, and a set of flags (see Section 4.2). These parameters are machine-dependent. The kernel's general region

may also contain machine-dependent subregions, e.g., subregions for I/O areas that must lie in a specific virtual address range.

11.1. Subregion Page Allocators

Subregions such as the resident and pageable data subregions are *page allocatable*; i.e., their virtual pages can be allocated and freed. Every such subregion has a SUBREGION_MGR object to perform this allocation. This object is created during system initialization.

```
VIRT_ADDR*
SUBREGION_MGR::alloc_page(
        int             cnt=1
);

SUBREGION_MGR::free_page(
        VIRT_ADDR*  virt_addr,
        int             cnt=1
);
```

SUBREGION_MGR::alloc_page() allocates virtual pages from a subregion. It returns a pointer to a virtual page. When multiple pages are requested, contiguous pages are allocated.

11.2. Memory Blocks and Pseudo-Permanence

A KVMEM_BLOCK object represents a virtual memory block of *sub-page* size. The object consists of a header followed by the memory block. KVMEM_BLOCK's are allocated from the subregions described above. The C++ runtime system in the kernel uses KVMEM_BLOCK's to store C++ objects created by the new statement.

Some kernel objects have finite lifetimes: they are created and later deleted. These objects are often shared by several modules, each storing its own pointer to the object. When an object is deleted, these pointers become *dangling pointers*. It is difficult in general to track down and notify the entities that share it [7]. To address this problem, the DASH kernel supports *pseudo-permanent* objects. Dangling pointers to such objects can be detected, so they can safely be deleted at any time. This facility is implemented as follows:

- The header of a KVMEM_BLOCK contains a a non-zero unique ID and a spinlock [30]. These fields are initialized when the KVMEM_BLOCK is allocated. When the KVMEM_BLOCK is freed, the ID field is zeroed.
- A reference to a pseudo-permanent object consists of a *(pointer, ID)* pair. The validity of a reference can be checked by comparing its ID component to the ID in the KVMEM_BLOCK header. This comparison must be done while the spin lock is held.
- When a KVMEM_BLOCK is freed, its memory cannot be allocated for any other purpose. This ensures that its ID field cannot accidentally assume a value matching a previous ID.

---

[7] Some language systems use *garbage collection* to solve this problem: an object is not explicitly deleted, and continues to exist as long as there is a reference to it.

11.3. Memory Block Allocation

The allocation of KVMEM_BLOCK objects from a particular subregion is managed by a KVMEM_BLOCK_MGR object. This is the only facility for allocating memory blocks. KVMEM_BLOCK_MGR objects for the resident and pageable data subregions are created during system initialization.

```
KVMEM_BLOCK*
KVMEM_BLOCK_MGR::alloc_block(
        int    size
);

KVMEM_BLOCK_MGR::free_block(
        KVMEM_BLOCK*        block
);
```

These functions allocate and free memory blocks. A KVMEM_BLOCK_MGR maintains linked lists of free memory blocks of various sizes (e.g., 128, 256, 384, ..., up to page size), plus a list of partially-used pages. KVMEM_BLOCK_MGR::alloc_block() uses the requested size to select a block list, and gets an entry from the list. If the list is empty, it allocates a block from a partially-used page. If no page has enough free space, it calls SUBREGION_MGR::alloc_page() to allocate one or more complete pages. KVMEM_BLOCK_MGR::alloc_block() assigns a unique ID to the memory block before returning.

KVMEM_BLOCK_MGR::free_block() sets the unique ID field to zero and inserts the block in the appropriate list. Memory blocks are not merged after they are released, and pages occupied by them are not returned to the SUBREGION_MGR.

12. SUMMARY

This section summarizes the major design decisions in the DASH VM system, and the reasoning behind them.

12.1. Specialized Mechanisms instead of Copy-on-Write

Our design divides a VAS into three regions (general, shared segment, and IPC), each of which provides a specific function. Other systems [1, 22] use a single *copy-on-write* mechanism for multiple purposes: data movement, VAS duplication, and read/write sharing. We prefer our approach for the following reasons:

- Copy-on-write requires more manipulation of hardware memory maps than does DASH. This manipulation is expensive on certain VM architectures. In particular, unmapping a page or changing the protection from read/write to read-only requires cache flushing on machines with a virtual address cache[8] [10], and can cause TLB inconsistency on a shared-memory multiprocessor [28]. Furthermore, the virtual address synonyms (multiple virtual addresses for a physical word) created by copy-on-write are not allowed on some shared-memory multiprocessors [11] and on machines with an inverted page table [12].

[8] To avoid flushing virtual address caches, Sprite uses copy-on-reference instead of copy-on-write [18].

- Many uses of the copy-on-write mechanism are not needed by DASH programs. In Accent [7, 8], copy-on-write is mainly used for copying address spaces when processes are created (44% of the data copied) and for file I/O (55%). DASH does not copy a VAS when a new process is created. A process is created either in the same VAS as its creator, or in a new VAS. If a separate VAS is needed, the shared segment mechanism eliminates the need for copying the VAS.

  DASH supports memory-mapped file I/O directly. A general subregion can be mapped to client-supplied backing store, and a shared segment is mapped to a shared segment file. Furthermore, we do not support write-sharing of mapped files (see Section 12.2), so copy-on-write is not needed. In DASH, a mapped file is either non-shared (in the general region) or shared in read-only mode (in the shared segment region).

- DASH is targeted at high-performance communication such as real-time video data, and needs an efficient mechanism for moving large data between VAS's. In particular, we need to move memory-resident data efficiently since communication buffers are memory-resident. Like DASH, copy-on-write uses memory remapping to avoid software copying. However, it is difficult to optimize the performance of copy-on-write for a particular purpose (i.e., moving a resident page). For example, in Accent [7, 8], remapping a resident page is slower than copying it (1.1 msec vs. 0.3 msec), though this is not important to Accent because 92.5% of the pages remapped are non-resident.

  In DASH, the IPC region is designed exclusively for moving large data between address spaces. With a limited usage, we are able to tailor the design toward high performance, and concentrate on reducing the overhead of flushing TLB's and virtual address caches.

12.2. No Direct Support for Write-sharing of Mapped Files

We do not directly support write-sharing of mapped files for three reasons:

- Studies of real-world systems [16, 19] show that write-sharing occurs rarely. We are willing to sacrifice it to increase the performance and simplicity of the rest of the system.
- Write-sharing is mainly used in database and transaction systems. Such applications generally have their own semantics for sharing [24, 25], so the semantics should not be imposed by the operating system kernel.
- Our design has hooks for supporting write-sharing or transactions outside the kernel. The design to use these hooks, to be included in a future version of the VM system, is as follows. A subregion of the general region is mapped to a *backing-store service* rather than a file. The client contacts the service, specifies how the file is to be shared, and establishes the VM mapping using the *service token* (see [31]) it obtained from the service. The client also calls the service directly for special operations such as "commit". The kernel calls the service, on a page fault, to read, write, or change the access rights to a page. The VM-to-backing-store interface has parameters for the desired and granted access rights. The backing-store service calls the VM system back to flush and/or invalidate a page in the subregion using standard system calls. DASH system calls (local or remote) are performed on

objects, and a capability to the subregion object is passed to the backing-store service on the previous read request.

The above scheme is similar to several other systems. The Mesa file system [23] used the idea of call-back procedures, but did not combine it with the VM system. The file server calls the client back using the routines supplied by the client when opening a file. In Mach, the interface between the VM system and the external pager is a two-way protocol based on asynchronous messages [29].

12.3. No Direct Support for Shared Memory

A *shared memory mechanism* allows programs to equate corresponding ranges of different VAS's, so that a change in one VAS is instantly visible in the others. Early systems like Multics [3,5] and Tenex [17] provide a sharing facility that is general but has poor performance. More recently, 8th Edition UNIX [21] and Sprite [20] provide simpler mechanisms with better performance.

A DASH VAS may contain multiple processes, so we support shared memory between processes. We do not support shared memory between VAS's for two reasons:

- DASH emphasizes security, and provides well-defined security semantics for message-passing operations between mutually distrustful VAS's. However, if two mutually distrusted VAS's share a memory segment, nothing prevents one VAS from corrupting the shared data at an unexpected time.
- DASH is a distributed operating system. An application based on the shared memory abstraction works fine on a single host, but is hard to extend over the network. The Apollo Domain system [13] and Li [15] extend the shared memory model to a network-wide single-level store. However, this approach is applicable only to homogeneous processors, and its performance depends heavily on locality of references.

13. ACKNOWLEDGEMENTS

We would like to acknowledge the contributions of Raj Vaswani, Robert Wahbe and Kevin Fall, who were involved with the VM system both as implementors and as clients.

REFERENCES

1. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, "Mach: A New Kernel Foundation for UNIX Development", *Proceedings of the 1986 Summer USENIX Conference*, Atlanta, Georgia, June 9-13, 1986, 81-92.
2. O. Babaoglu and W. Joy, "Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Reference Bits", *Proc. of the 8th ACM Symp. on Operating System Prin.*, Pacific Grove, California, Dec. 14-16, 1981, 78-86.
3. A. Bensoussan, C. T. Clingen and R. C. Daley, "The Multics Virtual Memory: Concepts and Design", *Comm. of the ACM 15*, 5 (May 1972), 308-318.
4. R. Carr and J. Hennessy, "WSCLOCK — A Simple and Effective Algorithm for Virtual Memory Management", *Proc. of the 8th ACM Symp. on Operating System Prin.*, Pacific Grove, California, Dec. 14-16, 1981, 87-95.
5. R. C. Daley and J. B. Dennis, "Virtual Memory, Processes and Sharing in Multics", *Comm. of the ACM 11*, 5 (May 1968), 306-312.
6. P. J. Denning, "Working Sets Past and Present", *IEEE Transactions on Software Engineering SE-6*, 1 (Jan. 1980), 64-84.
7. R. P. Fitzgerald, *A Performance Evaluation of the Integration of Virtual Memory Management and Inter-Process Communication in Accent*, Ph.D. Dissertation, CMU, Pittsburgh, PA, Oct. 1986.
8. R. Fitzgerald and R. Rashid, "The Integration of Virtual Memory Management and Interprocess Communication in Accent", *Trans. Computer Systems 4*, 2 (May 1986), 147-177.
9. D. D. Gajski and J. Peir, "Essential Issues in Multiprocessor Systems", *IEEE Computer*, June, 1985, 9-28.
10. R. A. Gingell, J. P. Moran and W. A. Shannon, "Virtual Memory Architecture in SunOS", *Proceedings of the 1987 Summer USENIX Conference*, Phoenix, Arizona, June 8-12, 1987, 81-94.
11. M. Hill, "Design Decisions in SPUR", *IEEE Computer*, Nov. 1986, 8-22.
12. "IBM RT PC hardware technical reference", Order no. SV21-8024, IBM, Austin, TX, 1985.
13. P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson and B. L. Stumpf, "The Architecture of an Integrated Local Network", *IEEE Journal on Selected Areas in Communication 1*, 5 (Nov. 1983), 842-857.
14. H. M. Levy and P. H. Lipman, "Virtual Memory Management in the VAX/VMS Operating System", *IEEE Computer*, Mar. 1982, 35-41.
15. K. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors*, Ph.D. Dissertation, YALEU/DCR-492, Yale University, Sep. 1986.

16. W. A. Montgomery, "Measurements of Sharing in Multics", *Proc. of the 6th ACM Symp. on Operating System Prin.*, West Lafayette, Indiana, Nov. 16-18, 1977, 85-90.

17. D. L. Murphy, "Storage Organization and Management in TENEX", *Proc. of the Fall Joint Comp. Conf., AFIPS National Computer Conf. Proc.*, 1972.

18. M. Nelson and J. Ousterhout, "Copy-on-Write for Sprite", *Porceedings of the 1988 Summer USENIX Conference*, San Franscisco, CA, June 20-24, 1988, 187-202.

19. J. Ousterhout, H. D. Costa, D. Harrison, J. Kunze, M. Kupfer and J. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System", *Proc. of the 10th ACM Symp. on Operating System Prin.*, Orcas Island, Eastsound, Washington, Dec. 1-4, 1985, 15-24.

20. J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson and B. Welch, "The Sprite Network Operating System", *IEEE Computer 21*, 2 (Feb. 1988), 23-36.

21. D. Presotto and D. Ritchie, "Interprocess Communication in the 8th Edition Unix System", *Proceedings of the 1985 Summer USENIX Conference*, Portland, Oregon, 1985, 309-316.

22. R. Rashid and G. Robertson, "Accent: A Communication-Oriented Network Operating System Kernel", *Proc. of the 8th ACM Symp. on Operating System Prin.*, Pacific Grove, California, Dec. 1981, 64-75.

23. L. G. Reid and P. L. Karlton, "A File System Supporting Cooperation Between Programs", *Proc. of the 9th ACM Symp. on Operating System Prin.*, Bretton Woods, New Hampshire, Oct. 10-13, 1983, 20-29.

24. M. Stonebraker, "Operating System Support for Database Management", *Comm. of the ACM 24*, 7 (July 1981), 412-418.

25. M. Stonebraker, D. DuBourdieux and W. Edwards, "Problems in Supporting Database Transactions in an Operating System Transaction Manager", *Operating Systems Review 19*, 1 (Jan. 1985), 6-14.

26. B. Stroustrup, "The C++ Programming Language", *Addison-Wesley*, 1986.

27. D. C. Swinehart, P. T. Zellweger, R. J. Beach and R. B. Hagmann, "A Structural View of the Cedar Programming Environment", *Trans. Prog. Lang and Systems 8*, 4 (Oct. 1986), 419-490.

28. S. Tzou, D. P. Anderson and G. S. Graham, "Efficient Local Data Movement in Shared-Memory Multiprocessor Systems", *Technical Report No. UCB/Computer Science Dpt. 87/385*, Berkeley, CA, Dec. 1987.

29. 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", *Proc. of the 11th ACM Symp. on Operating System Prin.*, Austin, Texas, Nov. 8-11, 1987, 63-76.

30. "The DASH Local Kernel Structure", UCB/Computer Science Dpt. Technical Report, in preparation, August 1988.

31. "The DASH Network Communication Architecture", UCB/Computer Science Dpt. Technical Report, in preparation, August 1988.