Addressing Moved Modules in a Capability-based Distributed Shared Memory

Frans A. Henskens

Department of Computer Science
University of Newcastle
N.S.W. 2308, Australia

The MONADS DSM consists of a single, very large, paged virtual memory space distributed across an arbitrary number of discrete nodes connected to a network. Each node acts as a server for the virtual memory pages stored at that node. The key to transparent access to the pages of the modules stored in the DSM is the use of structured addresses containing embedded location information. There is a problem with allowing modules to be moved because the location information embedded in the addresses used to access the data in them does not describe the correct storage location. In this paper we review an initial solution to the problem of accessing pages from moved modules. The required sequence of operations represents a significant, and usually unnecessary system overhead, since most page retrievals involve locally stored pages. An alternate solution to the problem is presented. This solution allows most page accesses, both local or remote and from moved or unmoved modules, to occur without unnecessary overheads.

1. Introduction

The MONADS project at the University of Newcastle in Australia has developed a local area network of MONADS-PC computers [15] connected to form a Distributed Shared Memory (DSM) [7]. DSMs allow processes executing on loosely coupled nodes to share data by reference, meaning that knowledge of the address of data in the shared memory space is sufficient to allow a process to access the data. The DSM paradigm has been implemented by other researchers to form shared memory systems, for example Ivy [11] and MemNet [6]. The main aim of these implementations was the exploitation of parallel algorithms on loosely-coupled processors, so their architectures were not expandable to large shared virtual memories, and were not oriented towards supporting persistence.

The MONADS DSM, on the other hand, implements a persistent distributed store, enabling users logged on to loosely coupled computers to share all resources including programs, data, and devices. The DSM has several important properties relating to the user's perception of the services provided by a networked node compared to those offered by a discrete node. These are:

1. The distribution of the system provides extra functionality without compromising the functionality provided by a discrete, non-networked machine.
2. A resource is identified by name. This name defines the resource only, and not its current location. As a consequence the resource may be moved to another node and still be accessed using its original name.
3. All users at all nodes have a coherent view of shared data.
4. The owner of a resource has control over access to the resource on a network-wide basis.
5. Programs developed for use on a non-networked machine execute without modification over the network.

The MONADS DSM provides a single network-wide virtual memory space in which all program code and data is stored. This virtual memory provides a store which complies with the orthogonal persistence principles of persistence independence, data type independence, and management orthogonality [2]. Since all MONADS processes execute within this virtual memory space, the conventional process address space protection scheme used by systems such as Unix cannot be used. A two-level capability-based protection technique enforced by the architecture is used to provide control over access to the programs and data in the store [9].

Programmers and compilers see the virtual memory as a collection of segments of arbitrary size. These segments are mapped onto the paged virtual address space in such a way that page and segment boundaries are orthogonal [8]. All segments have the same basic format, meaning that access to them is handled in a uniform manner. Segments contain data, and capabilities for other segments, allowing arbitrarily complex graph structures of segments to be constructed. Access to a segment is controlled by a segment capability which defines the segment start address, length, and type and access information. Segments are grouped together into information-hiding modules which present a purely procedural interface as proposed by Parnas [14]. Access
The implementation of a DSM system involves the maintenance of a cache of chunks or pages of the shared memory in the physical memory directly addressable by each processor. Associated with these caches are the problems of page location and data coherency. Data coherency is typically maintained using a multiple module, virtual page disk volume scheme similar to the one used for cache coherency in multiprocessors [10, 12]. Page data and protocol information are transferred between machines or nodes using messages transmitted on the network connecting the nodes.

Knowledge of the current location of the pages of a module is necessary to allow any node to transmit a message requesting a copy of a page for its local page cache, thus permitting processes executing at the node to access the data stored in the page. This knowledge is usually achieved in DSM systems using a full shared memory page table at each node [6, 11]. The existence of this page table renders these implementations unscaleable because it grows linearly with the size of the shared memory. It also creates an additional problem of maintaining the consistency of the multiple page tables. Determining the network location of objects is also a problem with other forms of distributed system. Those based on Remote Procedure Call (RPC) [3], for instance, typically locate object servers using a name handling group which provides the information required to build a table of object name to object manager mappings. These tables may be program-specific [5], or kernel-specific [13].

The approach taken in the MONADS DSM is the use of structured addresses coupled with the separation of disk page tables from main memory page tables. Translation between virtual and main memory addresses is achieved using address translation hardware which is effectively an inverted hash table [1]. Translation between virtual and disk addresses is accomplished by an alternate scheme, as described in section 2. The use of structured addresses provides a virtual address which defines the node on which the module containing the address was first created, and the within node disk volume on which the module is (or was) stored. Such addresses permit the easy and efficient location of pages from modules which have not been moved from their creating node and original volume. When the kernel at a node knows where a page is stored, it can obtain a copy of the page using a message protocol which is essentially an extension of the method used for the resolution of page faults in a single node virtual memory system. The storage location information embedded in addresses is therefore crucial to the page retrieval algorithm.

It is clearly desirable to allow modules to be moved between nodes in the network. The owner of the module may move to a different site and wish to move his modules to his new node. Moving such modules allows the owner to access them without using network bandwidth. A node may fail, and the disks connected to it may be mounted on another functional node, allowing users to access the data stored on these disks. It should also be possible to mount removable disks on any node with a suitable disk drive unit. There is a problem with allowing modules to be moved, however, because the location information embedded in the addresses used to access the data in such modules no longer describes the correct storage location. Moreover a scheme for the movement of modules must not render invalid the existing module capabilities used to access the module.

In this paper we first describe how copies of pages from non-moved modules are obtained. We then review a solution to the problem of locating and obtaining copies of pages from moved modules proposed in [4]. This solution requires the kernel to perform a sequence of operations every time a page is required, even if the page is stored locally. As a result, the kernel can detect pages from moved modules and adjust the page retrieval operation accordingly. We then show that this solution imposes severe overheads on accesses to non-moved modules, thus penalising the default situation, and propose an alternate solution based on the use of aliases for moved modules.

2. Obtaining pages from non-moved modules

When a process executing on a node attempts to access an address from a virtual page not currently stored in the local page cache, a page fault exception occurs, and the process is suspended. The node's kernel must then retrieve the page to allow the process to continue execution. If such a page is stored on a local disk, the local kernel's paging mechanisms are used to retrieve it. If the page is stored on a disk at some other node (called the page's owner node), a network message must be transmitted to the owner requesting provision of the page. When the kernel at the owner node receives this message, it transmits the page to the requesting node (subject to coherency issues), after which the requesting node's kernel restarts the suspended process. This sequence of events is basically an extension of conventional virtual memory management techniques, with the network acting as a large remote disk. The key to the efficient implementation of such network paging is kernel
knowledge of the owner node for any virtual memory page. Such knowledge is provided through the use of structured virtual addresses.

When a MONADS computer is manufactured, it is allocated a unique node number. The MONADS DSM is partitioned into regions, each of which corresponds to one of these numbers. A MONADS node may only create modules within the DSM region associated with its node number. Such partitioning is achieved by structuring virtual addresses so that the high order bits define the node number for the node on which the module containing an address was created. This is viable with MONADS addresses because of their width: 60 bits for the MONADS-PC, and 128 bits for the MONADS-MM [18].

As shown in figure 1, the range of virtual addresses allocated to a node is further partitioned to define within node disk volumes, and within volume address spaces. Volumes are disks or disk partitions. Each node maintains a Local Mount Table (LMT) defining which volumes are currently mounted at the node. Address spaces are contiguous regions of memory, each of which is used to store a single module. The name of a module is the unique number of the address space in which it is stored. Each address space has its own page table which maps from virtual addresses to disk addresses for that address space. This page table is stored within the address space which it describes [17].

Address space zero of each volume is special, and contains volume red-tape information, including a volume directory which indicates the disk address of the first or root page of each address space (and thus module) stored on the volume. Since the entries comprising the disk page table for an address space are either contained in or reachable from the root page, this information is sufficient to allow the disk address of any page of a module to be found. An address therefore contains embedded information defining the node which created the module containing the address, the disk volume on which the data at the address was originally stored, and the unique identity of the module containing the address.

When a page fault occurs, the local kernel examines the high order bits of the faulting address. For non-moved modules, these bits indicate

(1) whether the page is stored on local disk, and if not

(2) which node is the owner node for the page.

If the high order bits indicate that the page is stored at a remote node, the local kernel composes and transmits a network message requesting provision of the page according to the protocol described in [7].

As indicated in section 1, it is overly restrictive to prevent the movement of modules between volumes, and volumes between nodes. However the location information embedded in addresses is crucial to the efficient location of remote pages. If the volume containing a module is relocated, or the module is moved between volumes, this information becomes incorrect. In the following section we describe mechanisms which allow the location of such moved modules.

3. Brössler et al's proposal for addressing moved modules

It is tempting to move a module by simply copying it to an address space whose number describes the new location, and then deleting the old module. Such a 'solution' is not feasible because the name of a module is the number of the address space in which it is stored. As such the name of a module would change when the module was moved. Since this name is included in the capabilities used to access the module, all such capabilities would become invalid on the name change, and would need to be invalidated and replaced in order to enable continued access. Any process with the appropriate permission (defined by the module capability) may distribute copies of a module capability. As a consequence, the distribution of capabilities for a module may be unknown to any single user (such as the creator of the module), making such invalidation and replacement at best inefficient, and at worst impossible. It is therefore important that the movement of modules (as distinct from the copying of modules) must not involve any change to the module's name.

Techniques for accessing pages from moved modules may be divided into two cases:

(1) modules stored on relocated volumes, and
(2) modules moved between volumes.

We describe the techniques for each of these cases separately.

<table>
<thead>
<tr>
<th>Node No.</th>
<th>Volume Number</th>
<th>Within Volume Address Space Number</th>
<th>Within AS Page</th>
<th>Offset</th>
</tr>
</thead>
</table>

Figure 1. The Structure of a MONADS DSM Virtual Address.
3.1 Modules stored on moved volumes

The relocation of a complete volume may involve the
placement of a removeable disk in a disk drive unit
connected to a different node, or the connection of a fixed
disk to a different node. As a result of such movement
the node number embedded in addresses stored on the
volume describes the creating rather than the owner node.
This means that the node number/volume number part of
the module name may no longer be sufficient to allow a
kernel to locate pages of the module. From the point of
view of accessing the modules stored on a moved volume,
such volumes fall into two categories; volumes created on
some remote node and mounted locally, and volumes
mounted on some remote node but created on a different
node.

The LMT maintained by the kernel at each node is
used by the kernel to access locally mounted volumes.
For MONADS DSM nodes this table includes creating
node information for each such volume [4]. This allows
local access to locally mounted moved volumes to be
equivalent to local access to locally mounted locally
created volumes. The problem of access to the pages
stored on remotely mounted moved volumes is more
interesting.

To facilitate access to moved remotely mounted
volumes, the kernel at each node maintains a Foreign
Mount Table (FMT) [4]. This table contains entries
describing the current mounting node for remotely
mounted moved volumes that are of interest to the local
kernel. Relocated volumes of interest are either:

1. volumes not mounted on the node containing
   virtual memory pages being used by processes
   executing on the node, or
2. volumes that were originally created on the
   node and which have been moved to some
   other node. Such volumes are of interest if a
   node other than the mounting node is
   accessing virtual memory pages stored on the
   volume.

Nodes thus use the FMT for two purposes; to enable
the local kernel to direct page requests for pages from
moved volumes to the current owner node, and to enable
an ex-owner kernel to advise other nodes about the current
location of relocated volumes created by it. Unlike
Brössler et al [4] who propose that users should provide
advice on volume movement, we create FMT entries
automatically whenever the kernel detects interest in a
remotely mounted volume. Such interest may be either a
local page fault for a virtual page stored on the volume,
or receipt of a page request for a virtual page from a
locally created and subsequently moved volume.

If the kernel at a node receives a page request for a page
of a locally created moved module, it checks its local
FMT attempting to determine the new mounting node for
the volume. In the case that a matching FMT entry
exists, the kernel returns an advisory message containing
the identity of the current mounting node. If such an
FMT entry does not exist, the kernel broadcasts a
message requesting the current mounting node to identify
itself. The reply to this broadcast message is used by
the creating node to create an appropriate FMT entry, and
then to transmit an advisory message in reply to the page
request. On receipt of the advisory message, the kernel
which is attempting to resolve the page fault adds an
appropriate entry to its local FMT (thus allowing it to
directly resolve subsequent page faults from the volume),
and repeats the page request, this time to the mounting
node.

3.2. Modules moved from their original
volume

Both the node and volume numbers embedded in
addresses from modules that have been moved from their
original volume are potentially erroneous. The
mechanisms which allow access to pages from such a
module are initiated when the module is opened.

When a process wishes to access a module, it executes
the open system call. This call, which includes the
module capability as a parameter, causes the creation of a
Module Call Segment (MCS) for the module. The MCS
contains segment capabilities for the root of the data
encapsulated in the module and for the available module
interface procedures.

---

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Access List</th>
<th>Node Number/ Volume Number</th>
<th>Status</th>
</tr>
</thead>
</table>

Figure 2. The Structure of a Module Capability.
To set up the MCS, the root page of the module's address space must be obtained from disk, assuming it is not currently in main memory [9]. To facilitate this initial read operation, module capabilities include information which advises the probable storage location for moved modules. This advisory data is readily available to the kernel when the module is opened because the module capability must accompany the open call.

The probable storage location is provided by the inclusion of an advisory field in module capabilities [4]. The advisory field consists of a node number/volume number pair, as shown in figure 2. Unlike the other fields of module capabilities, this information is not privileged, and the owner of the capability can cause arbitrary values to be placed in it. The purpose of the advisory field is to provide a place for the capability owner to store the new node/volume number for moved modules. When the capability owner moves a module (a module may not be moved while an user has it open), the advisory field of the module capability is changed to the destination node/volume as part of the called system routine. Similarly, a user may change the advisory field in an owned module capability if he becomes aware that another user has moved the module. As we will describe, the system also changes the value of the advisory field when it detects that the current value is incorrect. Because the advisory field of a module capability may be out of date, a second less efficient mechanism for locating moved modules is also provided. When a module is moved from a volume, the old volume directory entry for the module is altered to indicate the new location of the module, thus creating a forwarding address [4]. This allows an ex owner node to provide advisory information for other nodes attempting to access the module. This may occur, for instance when the use of out-of-date advisory information results in a page request message being transmitted to the wrong node.

Using the module capability advisory information the requesting node transmits a message to the probable owner node asking for provision of the module root page. This message contains the requested page number, with embedded creating node, volume number, and advisory information defining the full identity of the volume that probably contains the required page. When an owner node receives such a message its kernel checks the advisory volume number field. If the field is not null, its contents are used by the kernel to find the module containing the requested page.

Once the module has been opened, a subsequent page fault for a page in the module presents only the virtual address of the page to the kernel, and this does not include the advisory information. The resolution of such page faults is difficult for pages in moved modules because the new location must be determined for each individual page. To facilitate such determination, the kernel maintains a list of open moved modules called the Moved Object Table (MOT) [4]. Using this table it can detect that the page comes from a module for which it holds advisory information, and uses this information to direct a page request message to the appropriate node.

A MOT is maintained by the kernel at every node, and has entries consisting of a moved module number field describing the creating node number, volume number, and address space number for the moved module, a location field describing the node number and volume number for the new location of the module, and a MCS count field, as shown in figure 3. Each MOT entry maps a unique module number to a new location.

If the advisory field in the module capability used to open a module is not null, the MOT is checked to see whether an entry already exists for the module. If not, an entry is created in the MOT using the module number for the moved module field, the contents of the advisory field for the location field, and one (1) for the MCS count field. If an entry does exist, the MCS count field for the existing entry is incremented by one to indicate that another local process also has the module open, and the location information in the MOT is used to access the module. When the module is subsequently closed, the appropriate MOT entry's count field is decremented. If this results in a zero value, the entry is removed from the MOT.

It is possible that the owner of a module capability is unaware of the fact that the module has been moved from its creating volume, and that the module capability has not been used since the move, meaning that the advisory field is null. It is also possible that the module has been moved since the user or system last updated the advisory field. In both of these cases the advisories of the advisory field are incorrect. Prior to any attempt to read the root page, the kernel checks the MOT to see whether an entry exists for the module. If a corresponding entry does exist, the kernel updates the advisory field in the presented module capability, and uses the location information in the MOT to access the page.

If not, the incorrect information in the advisory field is used, and consequently the attempt to read the module root page results in the return of a forwarding address. In this case the kernel updates its MOT, updates the advisory field in the presented module capability, and repeats the read using the updated location information.

It is possible for the return of forwarding information to occur several times before the read operation is successful. This would happen, for instance, if a module had been moved several times since the last update of the module capability advisory field. Under these circumstances the module capability advisory field would be out-of-date, resulting in the following of forwarding addresses prior to finding the module. When such a module is finally located, the system updates the module capability with the new location information, enabling the open operation to proceed more efficiently next time.
3.3. The open page fault algorithm using the MOT

The full sequence of steps taken by the kernel to determine where the root page of a module is located during an open operation is:

1. Compare the module name (address space number) with the equivalent field in the MOT. If a matching entry is found, use the node number and volume number in the MOT entry to access the page. If a matching entry is not found, then use the node number and volume number from the module capability advisory field, or from the page address if the advisory field is blank.

2. Compare the node number/volume number with the equivalent field in the LMT. If a matching entry is found then the volume that probably contains the page is locally mounted and the page fault may be resolved locally. If the appropriate LMT entry does not exist, then

3. Compare the node number/volume number with the equivalent field in the FMT. If a matching entry is found, transmit a page request message to the node indicated in the FMT, including any module capability advisory information as a parameter to the message. If the appropriate FMT entry does not exist, then

4. Assume that the volume containing the page is still mounted on the creating node, and transmit a page request message to the creating node indicated in the faulting address, then

5. If the result of the read attempt is a forwarding address, update the MOT and go back to step (1).

When the root page is successfully obtained, the advisory field of the module capability used to open the module is corrected if necessary to reflect the true location of the module.

3.4. The general page fault handler algorithm using the MOT

The full sequence of steps taken by the page fault handler to determine where a page is located is:

1. Compare the module name (address space number) with the equivalent field in the MOT. If a matching entry is found, use the node number and volume number in the MOT entry to access the page, including the node and volume numbers obtained from the MOT as a parameter to the page request message. If a matching entry is not found, then use the node number and volume number derived from the page address.

2. Compare the node number/volume number with the equivalent field in the LMT. If a matching entry is found then the volume that probably contains the page is locally mounted and the page fault may be resolved locally. If the appropriate LMT entry does not exist, then

3. Compare the node number/volume number with the equivalent field in the FMT. If a matching entry is found, transmit a page request message to the node indicated in the FMT. If the appropriate FMT entry does not exist, then

<table>
<thead>
<tr>
<th>Moved Module Number</th>
<th>Location</th>
<th>MCS Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating Node, Volume, and Address Space Number of Module</td>
<td>Node and Volume Number of New Location</td>
<td>Number of Local Processes with Module Open</td>
</tr>
</tbody>
</table>

Figure 3. The Structure of the Moved Object Table.
(4) Assume that the volume containing the page is still mounted on the creating node, and transmit a page request message to the creating node indicated in the faulting address.

Notice that this sequence of steps involves checking the MOT for every page fault. In the usual case such a check will result in a negative result because, in general, a module will not move independently of the volume on which it was created. The efficiency of page fault resolution in the typical case therefore suffers because of the atypical case. In contrast, the scheme for accessing pages from a module that remains on its original but relocated volume incurs no additional overhead for the typical page fault, that is a page fault for a page on a locally mounted volume.

We feel that handling of the common situation should be as efficient as possible, and that efficiency certainly should not suffer because of the atypical case. Any strategy used should initially assume that the problem to be solved is of the most commonly occurring type. In the following section we describe an alternate scheme which incurs no overhead for pages of modules which have not been moved.

4. Use of aliases in addressing moved modules

The essence of the new technique presented in this section is that the identity of an open module may be temporarily altered to reflect its current location, thus allowing efficient access to the pages of the module.

The implementation of this technique for accessing moved modules requires that:

(1) A new unique address space number defining the new node, volume, and (logical) within-volume address space is allocated to a moved module. This number is called the current name for the module, and is used for internal system purposes only. The name by which the module is known to users remains the name allocated when the module was created, and as a result all existing module capabilities still allow access to the module. The current name may be viewed as an alias for the original name.

(2) An additional table is maintained in address space zero of each volume. This table is called the Foreign Address Space Table (FAST), and contains mappings between module names and current names for moved modules currently stored on the volume, as shown in figure 4. The FAST is accessed using the original module name as a key, and allows the current name for any moved module stored on the volume to be determined.

(3) The method for moving modules is modified slightly so that the volume directory entry for a moved module at the new owner node contains the current name for the module rather than the original module name.

(4) When a module is moved from a volume the volume directory of the source volume is changed to link the current name used on the source volume to a forwarding address in the same manner as described in the previous section.

(5) A new message is used to request the root page of a module during an open operation on the module. This is called the request_root page message, and it contains requesting node number, requested page number, and storage volume number fields.

(6) A new message is used to supply the root page of a module during an open operation. This is called the supply_root page message, and it contains supplied page number, page data, and requested page number fields.

When a module is opened, the root page of the module must be accessed to allow creation of the MCS. When the MCS is set up the kernel knows whether the module is stored on its original volume or has been moved to a different volume. If the module has been moved, the kernel knows its new location because it has successfully obtained a copy of the module's root page.

Efficient access to the pages of the module is achieved by altering the segment capabilities used to access the module's data as they are stored in the MCS. This alteration replaces the original node number, volume number, and address space number fields with values indicating the current node, volume, and address space numbers. Subsequent accesses to these data segments generate virtual addresses containing the current name rather than the original name, meaning that the kernel can obtain pages from these segments as if the module had never been moved. Since all the segments of a module are contained within the module, and pointers to data within a module are relative to the address space in which it resides, these pointers do not need to be changed as a result of the change of address space name.

The issue of forwarding addresses was discussed in the previous section, and this discussion also applies to their use with the FAST. If the attempt to read the root page of a module results in the return of a forwarding address, the kernel:

(1) updates the advisory field of the presented module capability, and
(2) repeats the read using the updated location information.

Forwarding addresses effectively form a chain which leads to the current location of a moved module. When a module is moved then either:

(1) it is being moved for the first time, from the volume on which it was created, or
(2) the module has been previously moved.
When the module is moved, the forwarding address is mapped with the module name in the source volume directory. In case (1), no FAST entry for the module is necessary at the source volume because the volume directory entry for the module contains the original module name.

In case (2), the FAST entry must be retained at the source volume. This mapping from the module name to the source current name allows the module name to be associated with the forwarding address.

If all volumes in the forwarding address chain are mounted on active nodes in the network, this sequence will eventually find the moved module. The use of a Foreign Address Space Table (FAST) in accessing the pages of moved modules allows page faults for moved modules to be resolved efficiently whilst not increasing the overhead of resolution of page faults for non-moved modules.

4.1. The page server algorithm using the FAST

Every node with attached disk(s) acts as a page server for virtual pages stored on its disk(s). When a server node receives a request_root_page message, it must check the stored volume number field to determine whether the requesting node has included module location information in the message. If the field is null, the server node attempts to find the requested page on the volume whose number is embedded in the page address is used.

To find the page, the kernel at the server node checks the LMT to see whether the indicated volume is locally mounted. If it is, then the volume directory is checked to see whether an entry exists for the module. If an entry does exist, then the requested page is either:

1. from a module that has not been moved from its original volume, or
2. from a module which was originally stored on the indicated volume but has been subsequently moved.

In case (1) the page is supplied to the requesting node using the supply_root_page message with the supplied page number field and requested page number fields set to the same value, indicating that the module name and current name are the same. In case (2) a forwarding address is returned to the requesting node.

If the stored volume number field of the request message is not null, the kernel at the server node knows that the requested root page comes from a moved module, and the FAST for the indicated volume is checked. If no FAST entry exists for the module, then an invalid_address_space message is returned to the requesting node. If a FAST entry does exist, then it maps the module name to the corresponding current name. The current name is then found in the volume directory. The volume directory entry indicates either:

1. a forwarding address for the module, or
2. the disk address of the required root page.

In case (1), the server node transmits an advisory message with a forwarding address for the module back to the requesting node.

In case (2), the supplied page number field of the supply_root_page message is used to inform the requesting node of the current name for the module. The page number used in the request is returned in the requested page number field. Provision of the root page is subject to coherency issues [7].

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Current Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating Node, Volume, and Address Space Number of Module</td>
<td>Current Node, Volume, and (Logical) Address Space Number of Module</td>
</tr>
</tbody>
</table>

Figure 4. The Structure of a Typical Foreign Address Space Table Entry.
4.2. The open page fault algorithm using the FAST

The full sequence of steps taken by the page fault handler to determine where the root page of a module is located during an open operation is:

1. compare the node number/volume number from the module capability advisory field, or that embedded in the original module number if the advisory field is empty, with the equivalent field in the LMT. If a matching entry is found then the volume that probably contains the page is locally mounted and the page fault may be resolved locally. If the appropriate LMT entry does not exist, then

2. compare the node number/volume number with the equivalent field in the FMT. If a matching entry is found, transmit a request_root_page message to the node indicated in the FMT, with any module capability advisory information included in the storage volume number field of the message. If the appropriate FMT entry does not exist, then

3. assume that the volume containing the page is still mounted on the creating node, and transmit a request_root_page message to the creating node indicated in the faulting address, with any module capability advisory information included in the storage volume number field of the message, then

4. if the result of the read attempt is a forwarding address, go back to step (1).

When the root page is successfully received, the advisory field of the module capability used to open the module is corrected if necessary to reflect the true location of the module. A difference between the supplied page number and requested page number fields of the message indicates that:

1. the current name for the module differs from the original module name, and

2. the module name embedded in the supplied page number field is an alias for the module. This alias contains current location information for the module.

If the supplied and requested page numbers are the same, then the module has not been moved from its original volume, and the current and original module names are also the same. As the MCS is built, the current name is used within the root segment capabilities for the moved module so that the process may directly access the pages of the module while it keeps the module open.

4.3. The general page fault handler algorithm using the FAST

The full sequence of steps taken by the page fault handler to determine where a page is located is:

1. compare the node number/volume number with the equivalent field in the LMT. If a matching entry is found then the volume that contains the page is locally mounted and the page fault may be resolved locally. If the appropriate LMT entry does not exist, then

2. compare the node number/volume number with the equivalent field in the FMT. If a matching entry is found, transmit a page request message to the node indicated in the FMT. If the appropriate FMT entry does not exist, then

3. assume that the volume containing the page is still mounted on the creating node, and transmit a page request message to the creating node indicated in the faulting address.

By taking these steps the kernel is able to determine the location of pages for any module stored on a mounted volume. It should be noted that this sequence of steps is exactly the same for a moved or non-moved module, indicating that the use of the FAST technique has greatly simplified access to pages compared to the use of the MOT.

5. Conclusion

The described schemes allow the efficient movement of volumes between nodes and modules between volumes in the MONADS DSM in a manner that is completely transparent to users. This transparency is achieved even though the location information embedded in virtual addresses is used to obtain pages of the virtual address space. As a result the module capability provided to the owner of a module when it was created, and any copies of the capability subsequently distributed to other users, continue to enable access to such a moved module. Such movement may involve relocation of the entire volume on which the module is stored, or transfer of an individual module between volumes.

The scheme is able to transparently handle the movement of individual modules provided that forwarding addresses remain on mounted disks. With fixed disks, this will normally be the case. However, if the forwarding address is on an unmounted disk, or if the forwarding address information has been deleted, the forwarding address mechanism fails. The inclusion of an advisory field in module capabilities allows moved modules to be located without the need to rely on forwarding addresses, provided that the advisory field information is current. Such advisory information is readily available when a process first accesses the module because this first access must be accompanied by the module capability. Any subsequent access of the module is not accompanied by the module capability, and so the
advisory information is not readily available to allow resolution of a possible page fault caused by such access. Two solutions to this problem are presented, the first making advisory information available for use in resolution of every page fault. This solution is inefficient because the advisory information must be checked as part of the resolution of every page fault, even those for which it is not needed.

The new second solution is more efficient, allowing a moved module to (logically) adopt the identity of a module that had been created on its current storage volume. Such a change of identity allows the system to access the module pages using addresses that describe its current location whilst still allowing processes to open the module using its original name. This solution represents a significant improvement on that previously described because it imposes no overhead on access to non-moved modules.

Acknowledgements

The MONADS DSM project is supported by grants from the Australian Research Council and the University of Newcastle Research Management Committee. The author also wishes to thank Professor John Rosenberg of the University of Sydney for his contributions and suggestions.

References