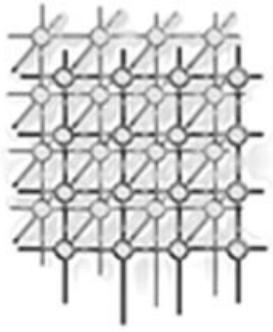
CONCURRENCY: PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper. 2000; **12**:251–280

Single address space implementation in distributed systems



Alberto Bartoli[1,*,†], Gianluca Dini[2,‡] and Lanfranco Lopriore[2,§]

[1]*Dipartimento di Elettrotecnica, Elettronica e Informatica, Università degli Studi di Trieste, via Valerio 10, 34100 Trieste, Italy*
[2]*Dipartimento di Ingegneria della Informazione: Elettronica, Informatica, Telecomunicazioni, Università degli Studi di Pisa, via Diotisalvi 2, 56126 Pisa, Italy*

SUMMARY

With reference to a distributed context consisting of computers connected by a local area network, we present the organization of a memory management system giving physical support to a uniform, persistent vision of storage according to a single address space paradigm. Our system implements a two-layer storage hierarchy in which the distributed secondary memory stores the valid data items and the primary memory supports a form of data caching, for fast processor access.

The proposed system defines a small, powerful set of operations that allow application programs to exert explicit control over the memory management activities at the levels of physical storage allocation, data migration across the network, and the data movements between the secondary memory and the primary memory. The system, that has been implemented in prototype form, is assessed from a number of viewpoints. We show that the storage requirements of the information for memory management are negligible. Moreover, the number of messages necessary to determine the network location of a given data item is low and independent of both the network size and the past movements of this data item in the distributed storage. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: distributed system; memory management; persistent storage; single address space; uniform storage

1. INTRODUCTION

The classical storage model is not uniform. Traditionally, programs see two distinct forms of store, a primary memory holding temporary data that exist as long as their creating process exists, and a

*Correspondence to: Alberto Bartoli, Dipartimento di Elettrotecnica, Elettronica e Informatica, Università degli Studi di Trieste, via Valerio 10, 34100 Trieste, Italy.
†E-mail: bartolia@univ.trieste.it
‡E-mail: dini@iet.unipi.it
§E-mail: lopriore@iet.unipi.it

Contract/grant sponsor: MURST, MOSAICO Project

Received 23 September 1999

Copyright © 2000 John Wiley & Sons, Ltd.

secondary memory giving permanent support to long-term data whose lifetime extends beyond the lifetime of the creating process [1]. In this storage model, pointers take the form of primary memory addresses and, as such, they lose their meaning if stored in the secondary memory.

In an alternative, *uniform* storage model, the program has a single perception of memory and addresses, which is at the virtual space level. The secondary memory is used for permanent storage of the information presently active in the virtual space, and the primary memory implements a form of caching of this information, for fast processor access. In this approach, we have a *persistent* vision of storage in which an information item created by a given process at a given virtual address maintains this address for its entire lifetime, irrespective of its position in the primary memory or secondary memory of any node [2–7]. The uniform storage model is well accommodated by a *single address space* memory addressing paradigm, in which the meaning of an address is unique and independent of the process issuing this address [8–10]. This contrasts with the traditional, multiple address space approach in which each process references a private address space and the interpretation of an address depends on the process using this address.

The advantages deriving from data persistence in a single address space environment are magnified in a distributed system consisting of *nodes* (computers) connected through a local area network. In a system of this type, data persistence not only eliminates the differentiation between primary and secondary memory, but also removes the separation between local and remote storage. Several distributed computer systems supporting the single address space abstraction have been developed. Examples are Angel [11], Mungi [12] and Sombrero [13].

With reference to a distributed context, we have approached the problem of giving physical support to a uniform, persistent storage environment in a memory management system based on the notion of a single address space. Our design effort has been guided by four main objectives, i.e.

- to support an application-controlled view of memory management, in which programs exert explicit control over the storage hierarchy in a distributed framework;
- to keep the storage requirements of the information for memory management low;
- to minimize the number of messages transmitted across the network to retrieve the information concerning page allocation in memory; and
- to fully distribute the memory management activities among the nodes.

We partition the virtual space into fixed-size units of storage allocation and management called *pages*. The physical memory resources of all the network nodes support a two-layer storage hierarchy in which the *block server* of a given page is the node reserving a secondary memory *block* for storage of the page contents, and the *frame server* is the node storing a copy of these contents in a primary memory *frame*, for fast processor access. A set of operations, the *page operations*, allows programs to control the allocation of the virtual pages in the physical memory and the page movements between the primary memory and the secondary memory. These operations make it possible to state and dynamically change the block server and the frame server of a given page, for instance. Thus, pages can be freely moved across the network. A page operation involving a given page can be issued from every node, even from a node reserving no memory for this page. It follows that each node can act as a repository for persistent information that can be loaded into the primary memory and subsequently accessed by the processor of any node. Distribution of the memory management activities is obtained by linking the nodes that store address translation information for the same given page, including the block server and

Copyright © 2000 John Wiley & Sons, Ltd.

the frame server, into a *logical chain*. We shall show that this organization allows us to comply with our performance requirements.

The rest of the paper is organised as follows. Section 2 illustrates our implementation of the single address space concept with special reference to the virtual pages. Section 3 shows the organization of the information items related to memory management and their distribution among the network nodes. Section 4 introduces the page operations and describes the actions caused by each of them. Section 5 discusses the salient features of the proposed memory management system and evaluates the system from a number of important performance viewpoints. Our system has been implemented in prototype form on a network of workstations running the Unix[¶] operating system. This prototype is briefly described in Section 6.

2. SINGLE VIRTUAL ADDRESS SPACE

Let us consider a distributed architecture consisting of a local area network connecting up to 2^d nodes. All the nodes share access to a single virtual address space of 2^v bytes (Figure 1). The virtual space is divided into 2^d equal-sized *partitions*, the ith partition being associated with the ith node. Each partition is logically divided into 2^r *pages* of size 2^g bytes, $g = v - d - r$. It follows that a virtual page is completely identified by a p-bit *full page identifier* P, $p = d + r$, consisting of a d-bit *partition identifier* $P^{(\text{partition})}$ and the r-bit *local identifier* $P^{(\text{local})}$ of the page in the partition specified by $P^{(\text{partition})}$. We shall call the node corresponding to the partition of a given page P the *partition server* $PS(P)$ of this page. This node is specified by $P^{(\text{partition})}$. A v-bit virtual address consists of three fields, a partition identifier, a local page identifier and a g-bit *offset* (Figure 2). The offset selects the referenced information item within the page addressed by the partition identifier and the local page identifier.

2.1. Active pages

The primary memory devices and the secondary memory devices of the network nodes give physical support to the virtual space, as follows. The secondary memory of each node is logically broken into *blocks*. All the blocks have the same, fixed size, which is equal to the size of a page (2^g bytes). A page P of the virtual space can contain valid information only if it is *active*. This means that a secondary memory block has been reserved for storage of this page in a network node. This node is called the page *block server*, and is denoted by $BS(P)$. An active page can be *deactivated*. This action deletes the page contents and releases the block reserved for page storage.

Initially, when a page is made active, a block is always reserved in the secondary memory of the partition server of this page, i.e. for page P, $BS(P) \equiv PS(P)$. However, the block server can be changed dynamically. To this end, the page contents must be copied into a free block of the secondary memory of the new block server. Thus, the secondary memory of a given node can contain pages belonging to the partition associated with this node as well as to any other partition. Moreover, the number of active pages in a given partition may well exceed the capacity of the secondary memory

[¶]Unix is a registered trademark of AT&T.

Copyright © 2000 John Wiley & Sons, Ltd.

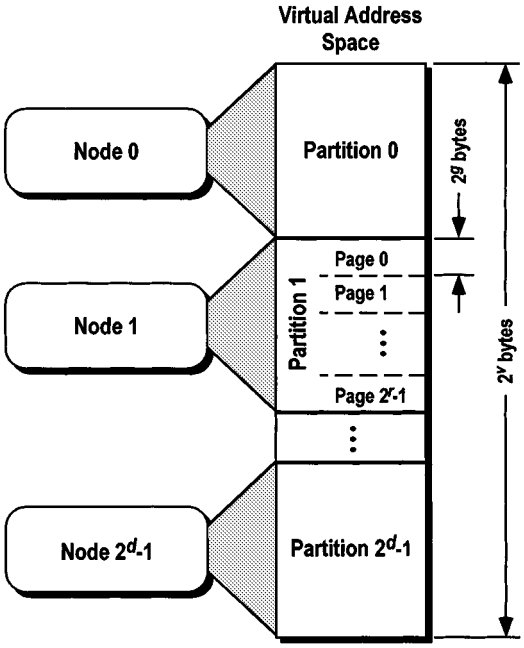


Figure 1. Configuration of the virtual address space.

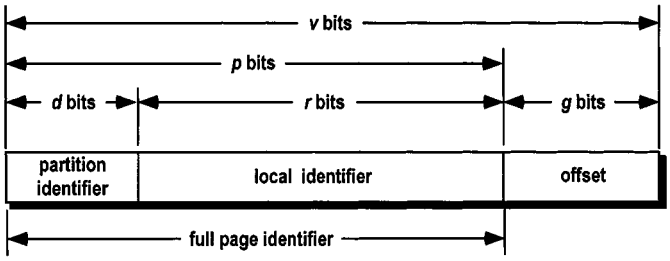


Figure 2. Configuration of a virtual address.

Copyright © 2000 John Wiley & Sons, Ltd.

of the corresponding node, provided that a subset of these pages have their block servers in the other nodes. In the following, we shall say that a page is active *in a given node* if the page has been made active and is currently stored in a block of the secondary memory of this node.

After a page has been deactivated, its contents are lost forever, and this page will never be made active again. It follows that a simple strategy to manage the virtual space is a sequential allocation of the pages in the same partition. This strategy can be implemented by using a *page counter* in each node. The page counter PC_N of node N contains the local identifier of the page that will be made active next in the partition corresponding to this node. Page counters are cleared at system initialization. When a new page is made active in N, the local identifier of this page is taken from PC_N, and then the page counter is incremented by 1. Other strategies can be easily devised, aimed at reusing page identifiers in the presence of a large number of nodes. These strategies will take advantage of knowledge of the pages that will be no longer used, as results from the explicit actions of page deactivation. We shall not address this issue at any further length [14].

2.2. Open pages

The primary memory of each node is logically broken into *frames*. All the frames have the same, fixed size, which is equal to the size of a page (2^g bytes). The processor of a node can access the information items contained in a given page only if this page is *open* in that node. This means that a primary memory frame has been reserved for this page in that node. An open page can be *closed*; this action releases the frame reserved for page storage.

A page can be open in the *external* mode or in the *internal* mode. In the external mode, any modification to the contents of the frame reserved for this page can be preserved before closing the page, by copying these contents back to the block storing the page in the page block server. In the internal mode, this copy action cannot take place. It follows that, in the internal mode, any modifications to the contents of a page will be confined to the frame reserved for this page; the modifications will never be reflected in the secondary memory.

Every given page P can be opened in the external mode in a single node at one time; this node is called the page *frame server*, and is denoted by $FS(P)$. On the other hand, no limit is imposed on the number of times a given page is opened in the internal mode. However, each page can be opened only once in each node; it follows that at any given time only one frame can be reserved in the given node for the same page. The internal mode allows processes running on different nodes to access the information stored in the same virtual page and carry out read actions in parallel, as is required to share a page containing program code in machine-executable form, for instance.

3. PAGE TABLES

The information for the management of the virtual space is distributed among the network nodes, and takes the form of three tables in each node, the *page tables*. Let us refer to a given node M (Figure 3):

- The *partition table* PT_M has one entry for each *active* page of the partition associated with node M, irrespective of the node where this page is presently active. The entry for a given page P contains the local identifier $P^{(\text{local})}$ of this page and the name of the page block server $BS(P)$.

Copyright © 2000 John Wiley & Sons, Ltd.

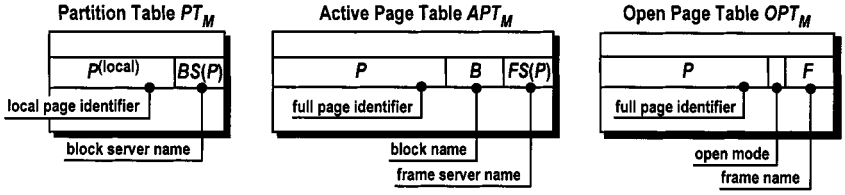


Figure 3. Configuration of the page tables of node M.

- The *active page table* APT_M has one entry for each page *active in node M*, irrespective of the partition of this page. The entry for a given page contains the full identifier P of this page and the name of the block reserved in M for page storage. If the page is open in the external mode, the entry also contains the name of the page frame server $FS(P)$.
- The *open page table* OPT_M has one entry for each page *open in node M*, irrespective of the partition of this page. The entry for a given page contains the full identifier P of this page, the name of the frame reserved in M for page storage, and the specification of the open mode (external or internal).

Thus, the active page tables of all the nodes together contain the information for the management of the secondary memory resources of the network, and the open page tables contain the information for the management of the primary memory resources. The distributed memory management information for a given page P open in the external mode takes the form of a *logical chain* where (Figure 4):

- the partition identifier $P^{(\text{partition})}$ identifies the partition server $PS(P)$;
- in $PS(P)$, the entry reserved for P in the partition table $PT_{PS(P)}$ contains the name $BS(P)$ of the block server of P;
- in $BS(P)$, the entry reserved for P in the active page table $APT_{BS(P)}$ contains the name B of the block reserved for P in the secondary memory of $BS(P)$ and the name $FS(P)$ of the frame server of P; and, finally,
- in $FS(P)$, the entry reserved for P in the open page table $OPT_{FS(P)}$ contains the name F of the frame reserved for P in the primary memory of $FS(P)$.

Translation of a virtual address generated by the processor of node M into the corresponding physical address in the primary memory of this node uses the information contained in OPT_M. Conceptually, the full identifier P of the referenced page is extracted from the address and is used to associatively search OPT_M for the entry relevant for P (Figure 5). The contents F of the frame name field of this entry are then paired with the offset to obtain the physical address of the referenced information item in the primary memory. Several techniques have been devised, giving efficient page table support in the presence of a large virtual space [15–17]. The implementation of address translation will take advantage of these techniques.

Copyright © 2000 John Wiley & Sons, Ltd.

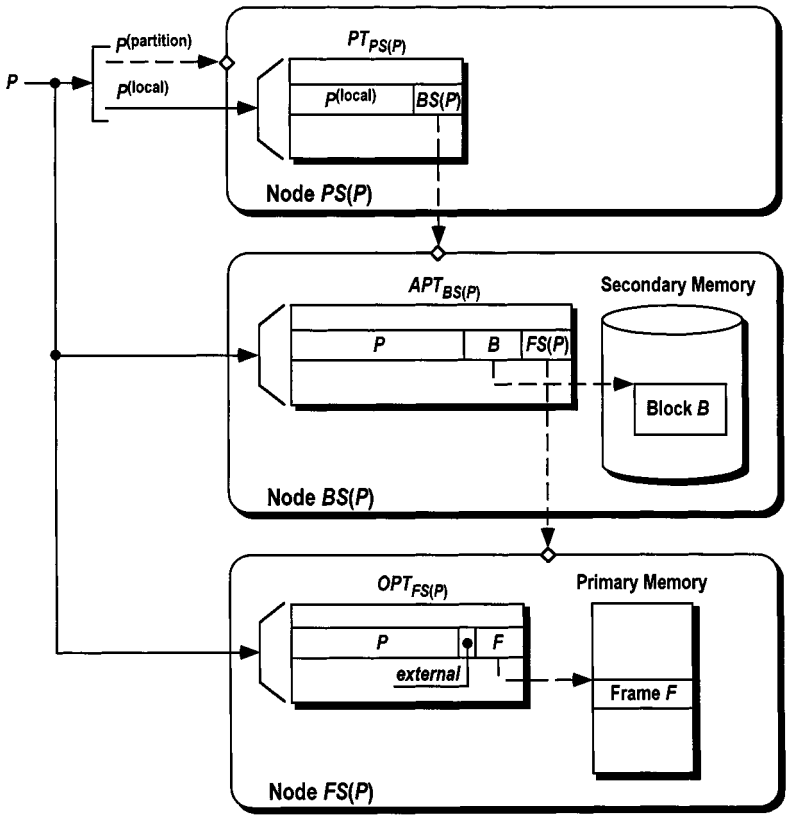


Figure 4. Relationships between the page tables: the *logical chain*.

4. PAGE OPERATIONS

The memory management system defines a number of operations for the management of the virtual pages. These *page operations* make it possible to activate, open, close and deactivate a page, to save the contents of a page open in the external mode into the secondary memory, and to change the page block server and frame server (Table I).

A page operation can be issued from every node, even from a node reserving no memory for the page involved in the operation. The actions caused by the execution of a page operation issued in node *M* usually involve other nodes as well. Interactions between these nodes take the form of *control messages* and *page messages*. A control message can be a *request message*, specifying actions to be performed by the recipient node and including the information necessary to carry out these actions; a *reply message*, containing the results of the actions carried out as a consequence of receipt of a request message; or a *completion message*, sent to each node involved in the execution of a page operation when execution

Copyright © 2000 John Wiley & Sons, Ltd.

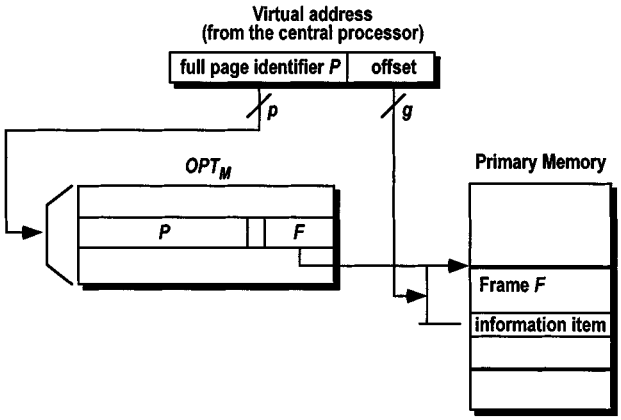


Figure 5. Translation of a virtual address into a physical address in the primary memory.

Table I. Page operations.

Operation	Effect
`P = Activate(N)`	Activates a new page in the partition associated with node *N*. Returns the full identifier *P* of this page.
`OpenExternal(P, N)`	Opens page *P* in the external mode in node *N*. Page *P* must be active. Fails if *P* is open in the external mode, or in the internal mode in *N*.
`OpenInternal(P, N)`	Opens page *P* in the internal mode in node *N*. Page *P* must be active. Fails if *P* is open in the external mode in *N*.
`Save(P)`	Copies the contents of page *P* back to the secondary memory. Page *P* must be open in the external mode.
`CloseExternal(P)`	Closes page *P* in its current frame server. Page *P* must be open in the external mode.
`CloseInternal(P, N)`	Closes page *P* in node *N*. Page *P* must be open in the internal mode in *N*.
`Deactivate(P)`	Deactivates page *P*. Page *P* must be active. Fails if *P* is open in the external mode.
`ChangeBlockServer(P, N)`	Changes the block server of page *P*. The new block server will be node *N*. Page *P* must be active.
`ChangeFrameServer(P, N)`	Changes the frame server of page *P*. The new frame server will be node *N*. Page *P* must be open in the external mode.

Copyright © 2000 John Wiley & Sons, Ltd.

terminates. A page message includes a copy of the contents of a page, for storage in a memory device of the recipient node. A message of this type is sent when the contents of a page must be moved from one node to another, to change the block server or the frame server of this page, for instance.

4.1. Accessing the page tables

In the presence of two or more page operations being executed in the same node at the same time, we shall take advantage of the usual critical section mechanisms to protect the concurrent accesses to the page tables generated by these operations. Moreover, we associate a *lock flag* with each partition table entry. The lock flag of a given entry is initially clear. It is set when a page operation accesses this entry, and is cleared when execution of the operation terminates, as part of the actions connected with the receipt of the completion message. When the partition table entry relevant to a given page is locked, any access attempt to this entry will be delayed until the lock is cleared. In this way, we force sequentiality on the concurrent accesses to all the page table entries relevant to that page, as follows from the fact that these entries are always accessed starting from the partition table, in the order imposed by the logical chain.

Of course, if the execution of a given page operation fails, each page table entry modified prior to failure must be restored to its initial state before unlocking this entry. This means that the previous contents of the entry must be saved for recovery purposes, and eventually be discarded, on receipt of the completion message.

In the rest of this section, we shall detail the actions involved in the execution of each page operation. To keep the presentation short, we do not mention the following actions, which, however, are part of the execution of every page operation:

- the acquisition and the subsequent release of the critical sections protecting the page tables;
- the setting and the clearing of the lock flags;
- the transmission of the completion messages; and, finally,
- the actions connected with failure treatment, and in particular, the saving and subsequent possible restoring of the contents of the page table entries, and the notification of the failure to the nodes involved in the failing operation.

We wish to point out that, if an operation for a given page is issued in a node that is a server of this page, part of the actions described below will not be carried out. These actions can be easily identified; we shall not discuss this issue at any further length.

4.2. Activating a page

The `P = Activate(N)` operation activates a new page in the partition associated with node N and returns the full identifier P of this page. The execution of this operation in node M causes the sending of a page activation request from M to N (Figure 6). On receiving this message, N searches a free block B in its own secondary memory (if no free block is available, `Activate` fails). Then, the local identifier $P^{(\text{local})}$ of the new page is read from the page counter PC_N of node N, this page counter is incremented by 1, and quantity N is paired with $P^{(\text{local})}$ to form the full identifier P of the new page. Quantities $P^{(\text{local})}$ and N are inserted into a free entry of the partition table PT_N, and quantities P and B are inserted into a free entry of the active page table APT_N (these two actions reflect the fact

Copyright © 2000 John Wiley & Sons, Ltd.

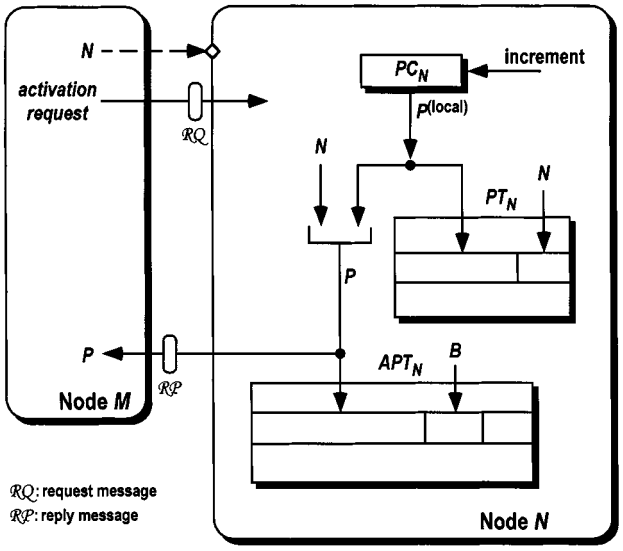


Figure 6. Actions involved in the execution of the `P = Activate(N)` operation.

that now N is both the partition server and the block server of P). Finally, a reply message containing quantity P is returned to M to form the operation result.

4.3. Opening a page

The `OpenExternal(P, N)` operation opens a given active page P in the node specified by the operation parameter N; the page is opened in the external mode. The execution of this operation in node M produces the following actions (Figure 7):

1. Node M sends a request message containing the local identifier $P^{(\text{local})}$ of page P to the partition server $PS(P)$ identified by the partition identifier $P^{(\text{partition})}$. On receiving this message, $PS(P)$ searches the partition table $PT_{PS(P)}$ for an entry for P to verify that P is active (if P is not active, `OpenExternal` fails). Then, the name $BS(P)$ of the current block server of P is read from $PT_{PS(P)}$, and a reply message containing quantity $BS(P)$ is returned to M.
2. Node M sends a request message containing quantities P and N to $BS(P)$. On receiving this message, $BS(P)$ accesses the frame server field of the entry reserved for P in the active page table $APT_{BS(P)}$ and ascertains whether P is already open in the external mode (if P is open in the external mode in node N, `OpenExternal` terminates successfully, whereas if P is open in the external mode in any other node, `OpenExternal` fails). Then, the name B of the block storing P is read from $APT_{BS(P)}$, quantity N is inserted into the frame server field of this entry, and a page message Π containing quantity P and the contents of block B is sent to node N.

Copyright © 2000 John Wiley & Sons, Ltd.

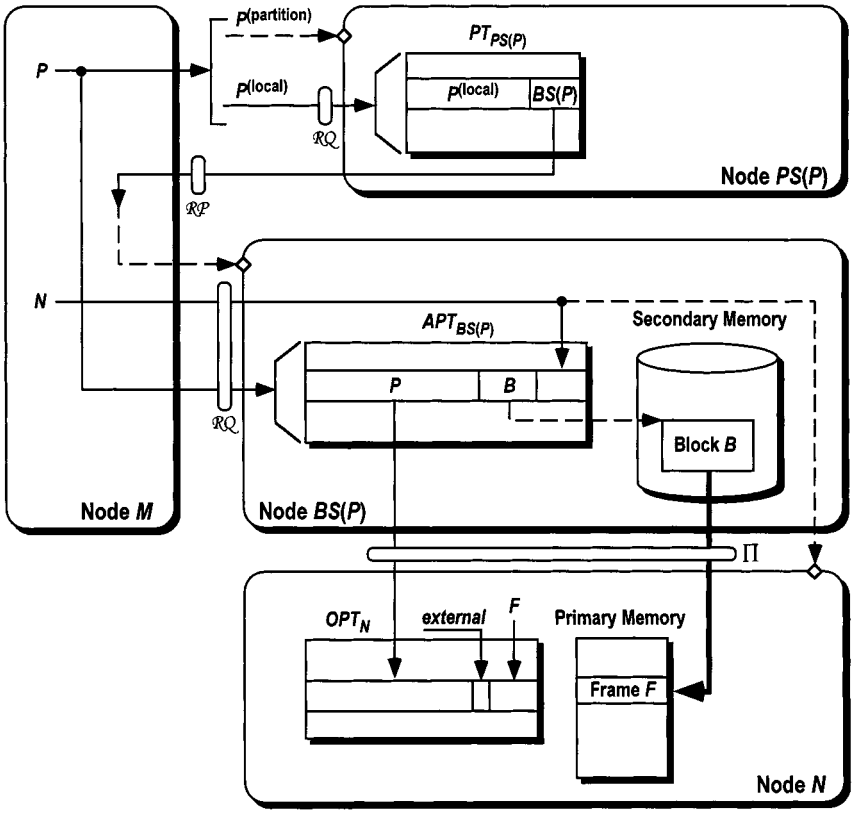


Figure 7. Actions involved in the execution of the `OpenExternal(P, N)` operation.

3. On receiving message Π, node N accesses the open page table OPT_N and ascertains whether page P is already open in the internal mode in N (in this case, `OpenExternal` fails). Then, a search is made for a free frame F in the primary memory of N, and the contents of P are copied from Π into F (if no free frame is available in N, `OpenExternal` fails). Finally, an entry is reserved for P in OPT_N; this entry is filled with quantities P and F and the specification of the external mode.

The `OpenInternal(P, N)` operation opens a given active page P in the node specified by the operation parameter N; the page is opened in the internal mode. The actions produced by the execution of this operation are similar to those described above and relevant to the `OpenExternal` operation. However, at execution step 2, `OpenInternal` fails only if P is already open in the external mode *in node N*; this is a consequence of the fact, stated in sub-Section 2.2, that a given page can be open in the internal mode in more than one node at one time.

Copyright © 2000 John Wiley & Sons, Ltd.

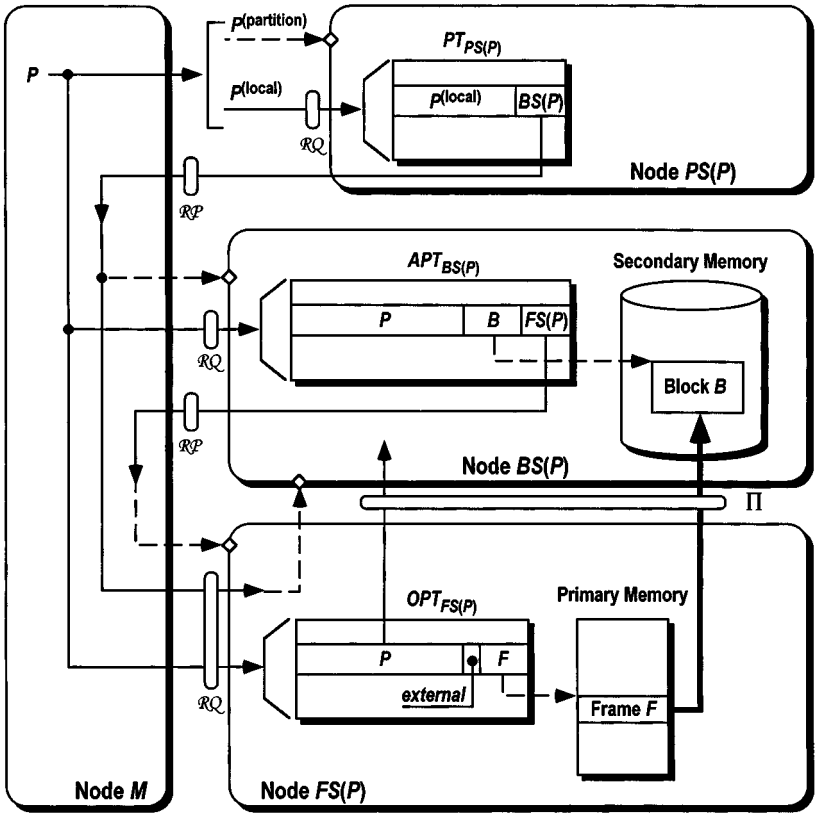


Figure 8. Actions involved in the execution of the `Save(P)` operation.

4.4. Saving a page

The `Save(P)` operation copies the contents of the given page P from the frame storing this page in its current frame server to the block reserved for the page in its current block server. The execution of this operation in node M is as follows (Figure 8):

1. Node M sends a request message containing the local identifier $P^{(\text{local})}$ to the partition server $PS(P)$. On receiving this message, $PS(P)$ accesses the partition table $PT_{PS(P)}$ and verifies that P is active (if P is not active, `Save` fails). Then, the name $BS(P)$ of the current block server of P is read from $PT_{PS(P)}$, and a reply message containing quantity $BS(P)$ is returned to M.
2. Node M sends a request message containing quantity P to $BS(P)$. On receiving this message, $BS(P)$ accesses the entry reserved for P in the active page table $APT_{BS(P)}$ and verifies that P is open in the external mode (if this is not the case, i.e. the frame server field of this entry is clear,

Copyright © 2000 John Wiley & Sons, Ltd.

`Save` fails). The name $FS(P)$ of its current frame server and the name B of the block reserved for P are read from the table (quantity B is set aside for later use; see step 4 below). Then, a reply message containing quantity $FS(P)$ is returned to M.

3. Node M sends a request message containing quantities P and $BS(P)$ to $FS(P)$. On receiving this message, $FS(P)$ reads the name F of the frame reserved for P from the open page table $OPT_{FS(P)}$. Then, a page message Π containing quantity P and the contents of frame F is sent to $BS(P)$.
4. On receiving message Π, $BS(P)$ copies the contents of P from Π into block B.

4.5. Closing a page

The `CloseExternal(P)` operation closes the given page P in its current frame server. The page must be open in the external mode. The execution of this operation in node M is as follows (Figure 9):

1. Node M sends a request message containing the local identifier $P^{(\text{local})}$ to the partition server $PS(P)$. On receiving this message, $PS(P)$ accesses the partition table $PT_{PS(P)}$ and verifies that P is active (if P is not active, `CloseExternal` fails). Then, the name $BS(P)$ of the current block server of P is read from $PT_{PS(P)}$, and a reply message containing quantity $BS(P)$ is returned to M.
2. Node M sends a request message containing quantity P to $BS(P)$. On receiving this message, $BS(P)$ accesses the active page table $APT_{BS(P)}$ and verifies that P is open in the external mode (if this is not the case, `CloseExternal` fails). Then, the name $FS(P)$ of the current frame server of P is read from the entry of $APT_{BS(P)}$ reserved for P, the frame server name field of this entry is cleared, and a reply message containing quantity $FS(P)$ is returned to M.
3. Node M now sends a request message containing quantity P to $FS(P)$. On receiving this message, $FS(P)$ reads the name F of the frame storing P from $OPT_{FS(P)}$. Then, frame F is made free, and P is deleted from the table.

The `CloseInternal(P, N)` operation closes page P in the given node N. The page must be open in the internal mode in N. The execution of this operation in node M produces the sending of a control message containing the full page identifier P from M to N (Figure 10). Node N now accesses the open page table OPT_N and verifies that P is open in the internal mode (if this is not the case, `CloseInternal` fails). Then, the name F of the frame reserved to P is read from OPT_N, frame F is made free, and P is deleted from the table.

4.6. Deactivating a page

The `Deactivate(P)` operation deactivates the given page P, thereby freeing the block reserved for this page in its current block server. The execution in node M is as follows (Figure 11):

1. Node M sends a request message containing the local identifier $P^{(\text{local})}$ to the partition server $PS(P)$. On receiving this message, $PS(P)$ accesses the partition table $PT_{PS(P)}$ and verifies that P is active (if P is not active, `Deactivate` terminates). Then, the name $BS(P)$ of the current block server of P is read from $PT_{PS(P)}$, P is deleted from the table, and a reply message containing quantity $BS(P)$ is returned to M.

Copyright © 2000 John Wiley & Sons, Ltd.

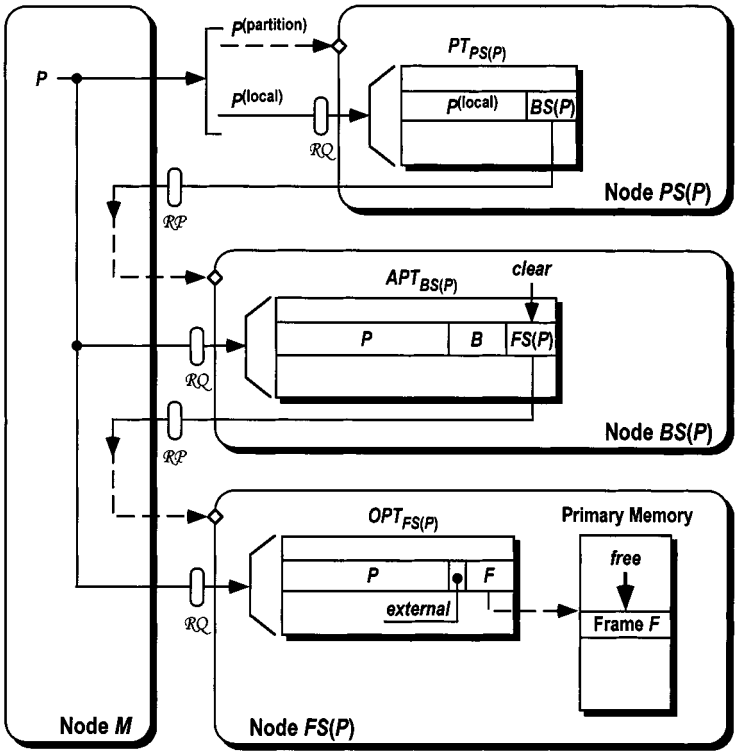


Figure 9. Actions involved in the execution of the `CloseExternal(P)` operation.

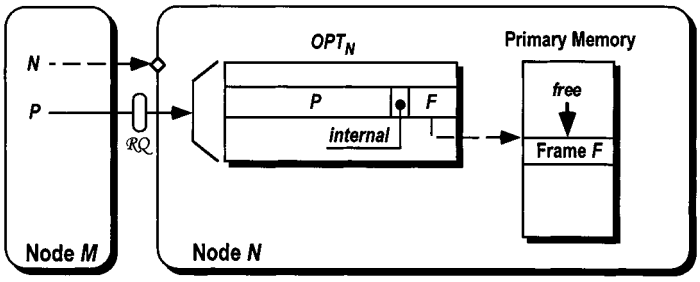


Figure 10. Actions involved in the execution of the `CloseInternal(P, N)` operation.

Copyright © 2000 John Wiley & Sons, Ltd.

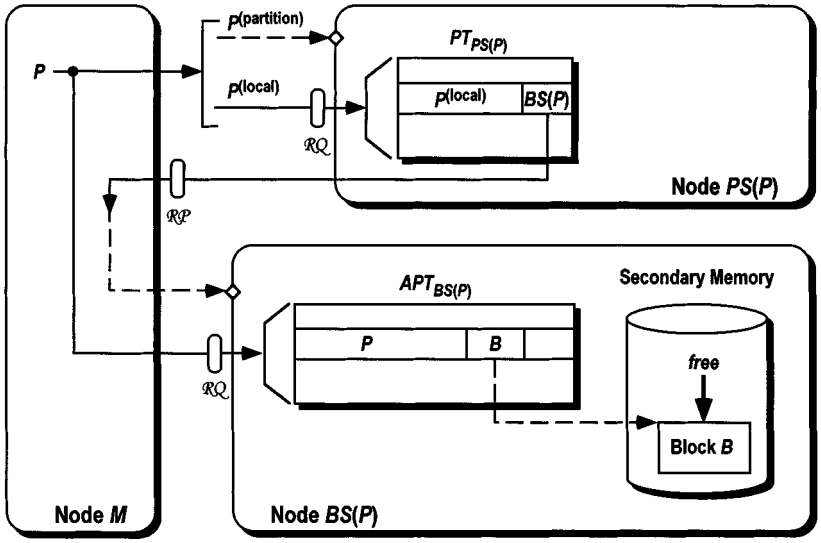


Figure 11. Actions involved in the execution of the `Deactivate(P)` operation.

2. Node M sends a request message containing quantity P to $BS(P)$. On receiving this message, $BS(P)$ accesses the active page table $APT_{BS(P)}$ and ascertains whether P is open in the external mode (in this case, `Deactivate` fails). Then, the name B of the block reserved for P is read from $APT_{BS(P)}$, block B is made free, and P is deleted from the table.

4.7. Changing the block server

The `ChangeBlockServer(P, N)` operation changes the block server of a given page P; the new block server will be node N. This effect is obtained by moving P from the secondary memory of its current block server $BS(P)$ to the secondary memory of N. The execution of this operation in node M is as follows (Figure 12):

1. Node M sends a request message containing the local identifier $P^{(local)}$ and quantity N to the partition server $PS(P)$. On receiving this message, $PS(P)$ accesses the partition table $PT_{PS(P)}$ and verifies that P is active (if P is not active, `ChangeBlockServer` fails). Then, the name $BS(P)$ of the current block server of P is read from $PT_{PS(P)}$, a reply message containing this block server name is assembled, quantity N is inserted into $PT_{PS(P)}$ and the reply message is returned to M.
2. Node M sends a request message containing quantities P and N to $BS(P)$. On receiving this message, $BS(P)$ reads the name B of the block storing P and the name of the frame server $FS(P)$ of P from the active page table $APT_{BS(P)}$. A page message Π is assembled, including

Copyright © 2000 John Wiley & Sons, Ltd.

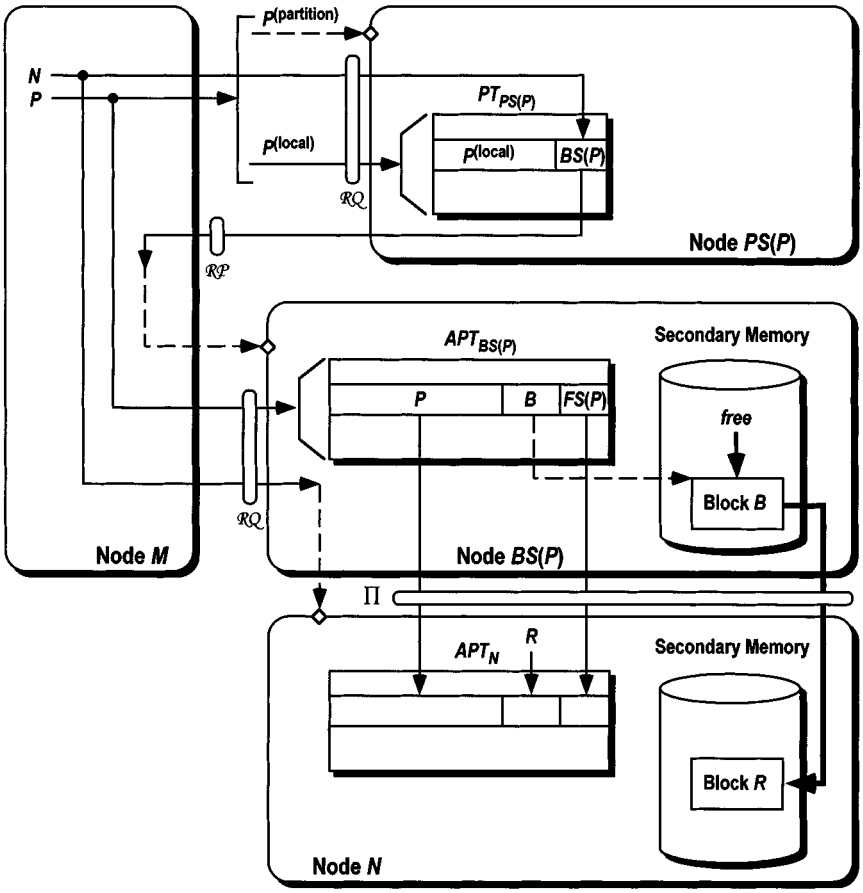


Figure 12. Actions involved in the execution of the `ChangeBlockServer(P, N)` operation.

quantities P, $FS(P)$, and the contents of block B. Then, B is made free, P is deleted from $APT_{BS(P)}$ and Π is sent to N.

3. On receiving message Π, node N searches its own secondary memory for a free block R and copies the contents of page P from Π into R (if no free block is available in N, `ChangeBlockServer` fails). Finally, an entry of the active page table APT_N is reserved for P and is filled with quantities R and $FS(P)$.

4.8. Changing the frame server

The `ChangeFrameServer(P, N)` operation changes the frame server of a given page P; the new frame server will be node N. This effect is obtained by moving P from the primary memory of its

Copyright © 2000 John Wiley & Sons, Ltd.

current frame server $FS(P)$ to the primary memory of N. The execution of this operation in node M is as follows (Figure 13):

1. Node M sends a request message containing the local identifier $P^{(\text{local})}$ to the partition server $PS(P)$. On receiving this message, $PS(P)$ accesses the partition table $PT_{PS(P)}$ and verifies that P is active (if P is not active, `ChangeFrameServer` fails). Then, the name $BS(P)$ of the block server of P is read from the table, and a reply message containing quantity $BS(P)$ is returned to M.
2. Node M sends a request message containing quantities P and N to $BS(P)$. On receiving this message, $BS(P)$ accesses the active page table $APT_{BS(P)}$ and verifies that P is open in the external mode (if this is not the case, `ChangeFrameServer` fails). The name $FS(P)$ of the current frame server of P is read from $APT_{BS(P)}$ and is replaced with quantity N. Then, a reply message containing quantity $FS(P)$ is returned to M.
3. Node M sends a request message containing quantities P and N to $FS(P)$. On receiving this message, $FS(P)$ accesses the open page table $OPT_{FS(P)}$ and reads the name F of the frame reserved for P. A page message Π containing quantity P and the contents of frame F is assembled, F is made free, P is deleted from $OPT_{FS(P)}$ and Π is sent to node N.
4. On receiving message Π, node N accesses the open page table OPT_N and ascertains whether P is open in the internal mode in N (in this case, `ChangeFrameServer` fails). Then, the contents of page P are copied from Π into a free frame T of the primary memory of N (if no free frame is available in N, `ChangeFrameServer` fails). Finally, an entry of OPT_N is reserved for P and is filled with quantity T and the specification of the external mode.

5. DISCUSSION

5.1. Single address space

In the traditional, multiple address space approach, each address space is confined within the boundaries of a single process and a single network node. The meaning of a primary memory address is lost if this address is transmitted to a different process or to a different node. Serious obstacles follow for information sharing, resulting in poor integration between applications operating on large, shared data structures, for instance [9,18,19]. Conversely, in a single address space environment, the meaning of an address is independent of both the process using this address and the present location of the process in the network. A data item created by a given process at a given virtual address can be referenced by different processes using this address even from different nodes. The sharing of a given information item between processes can be simply obtained by providing each of them with a pointer to this information item. Consequently, the movements of data between the nodes [20,21] are facilitated, and interactions between remote applications are favoured.

Let us refer to the classical concept of a remote procedure call, for instance [22,23]. In a multiple address space environment, if the arguments of a given remote procedure include a pointer, the entire data structure referenced by this pointer must be transmitted by the client node to the procedure in the server node as part of the remote call protocol. At the client side, a *stub* (piece of code) is responsible for a *marshalling* activity converting the arguments into a linear form suitable for transmission across

Copyright © 2000 John Wiley & Sons, Ltd.

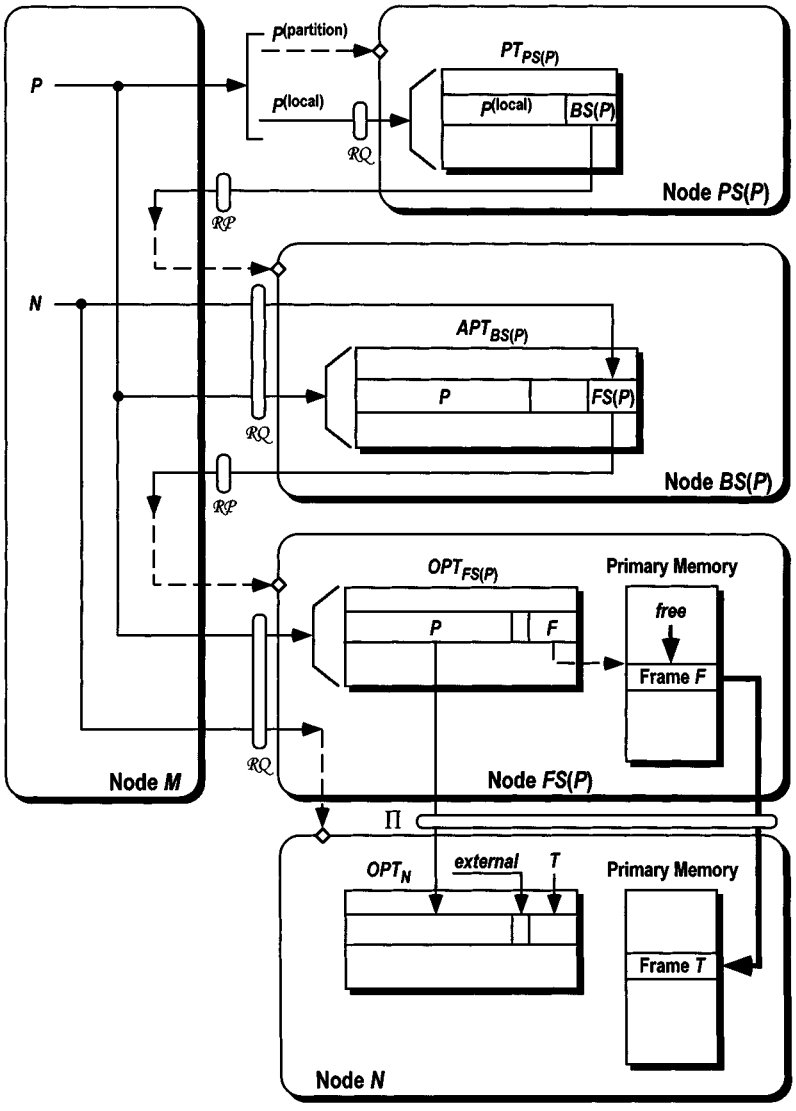


Figure 13. Actions involved in the execution of the `ChangeFrameServer(P, N)` operation.

Copyright © 2000 John Wiley & Sons, Ltd.

the network [24]. At the server side, an *unmarshalling* activity will convert the arguments back to the original form before being processed by the called procedure. If an argument takes the form of a complex data structure containing pointers, this data structure is flattened in the client as part of the marshalling operations, and is reconstructed in the server as part of the unmarshalling operations. High processing time costs are connected with these conversion activities. On the other hand, in a single address space environment, an arbitrarily complex data structure can be transmitted to a remote procedure by simply transmitting a pointer to this data structure. The called procedure will use this pointer to gain access to those components of the data structure that are actually involved in the procedure execution, by opening the page or pages storing these components.

As a further example, let us refer to migration of a process running in a given node to a different node [21,25]. In a traditional memory environment, the separation of the address spaces of the two nodes implies that, after migration, the memory area reserved for the process state in the first node can no longer be accessed from the new node. If a portion of this area is needed after migration, its contents must be tramsmitted to the new node as part of the process migration protocol. Conversely, in a single address space environment, when a process migrates we need to move only those components of the process state that are strictly necessary to resume execution of the process in the new node. Actual movement of any other component will be delayed till the occurrence of the first access to this component in the new node. The access will be preceded by the opening of the corresponding page in the new node. Significant performance advantages may follow from this load-on-demand strategy with respect to an overall copy of the process state [26]. A portion of the process state not needed in the new node will never migrate. This will be especially the case if a new process migration takes place, back to the original node, for instance.

Of course, if the memory requirements of all the processes must be satisfied within the framework of a single address space, the underlying machine architecture must support a large address size, or unacceptable memory limitations will follow for each process (indeed, the possibility to extend the total space of all the existing processes far beyond the limit of the processor address size perhaps was the main motivation for the introduction of private address spaces). The recent appearance of 64-bit machines, e.g. Sun SPARC [27], MIPS R10000 [28] and Digital Equipment Corporation Alpha [29], has provided an important opportunity for a re-examination of the single address space paradigm [30,31].

5.2. Uniform, persistent storage

In a classical memory environment, programs reflect the physical separation between the primary memory and the secondary memory. These two store forms correspond to different high level language constructs, e.g. arrays and structures for the primary memory and files for the secondary memory. A data item generated by a program in the primary memory must be converted to file form for permanent storage in the secondary memory. Conversion is also required for a large data structure not suitable for being stored in the primary memory, as its size exceeds the primary memory size. A new conversion will be required to the original form when transferring data from the secondary memory back to the primary memory. These conversion activities have a high cost; Atkinson *et al.* [32] report that they typically account for 30% of the application program code, for instance.

In this storage model, a pointer takes the form of an address in the *primary* memory and, as such, it loses its meaning if stored in the secondary memory [10]. Consequently, the pointers in a given data

Copyright © 2000 John Wiley & Sons, Ltd.

structure must be eliminated when this data structure is moved to the secondary memory. They will be restored to the original form when that data structure is moved back to the primary memory. A process is modelled as an independent activity that transforms an input stream into an output stream. In fact, input and output operations give support to two otherwise unrelated concepts, the permanent storage of data in the secondary memory and the exchange of information with the external environment.

Conversely, in a distributed single address space system incorporating the notion of a uniform, persistent storage, pointers reference specific locations of the *virtual* space, and consequently, their meaning is preserved even in the secondary memory. It follows that the movements of a given data item between the primary memory and the secondary memory require no conversion of the representation of this data item [33]. Arbitrarily complex data organizations retain their structure even in the secondary memory. No translation to file form is required to move data structures incorporating pointers to the secondary memory. A translation of this type is only required occasionally, to transmit data outside the single address space, across the network boundaries, for instance. The traditional view of a file as a stream of bytes to be read from and written to disk no longer holds [34]. Input/output now concerns only source/sink devices such as keyboards, screens and printers.

A relationship exists between the single address space paradigm and the *distributed shared memory* (*DSM*) paradigm [35–37]. In a DSM environment, application programs have a view of the primary memory resources distributed among the network nodes that is the same as the normal local memory. In fact, the DSM concept is the result of an effort to combine the two classical concepts, a tightly coupled multiprocessor featuring a shared memory accessible to all processors and a distributed system composed of nodes interconnected through a high-speed local network. With respect to a message-passing environment relying on communication primitives for data transmission among the nodes, an important motivation for a DSM environment is the ease of program writing. The simple underlying abstraction model allows an easy transition from sequential to distributed programming while preserving the scalability property typical of a network system. The single address space paradigm can be conceived as a result of the addition of the properties of a uniform and persistent storage to the DSM paradigm. In a single address space environment, the independence of the address of a given information item from the present storage location of this information item is extended to include not only the primary memory, but also the secondary memory.

5.3. Memory management controlled by applications

In the classical, *automatic* memory allocation paradigm, the running program has no direct control over the management of the system storage. Instead, a memory management system, developed independently, observes the program behaviour in memory and attempts to predict the future program memory requirements from an analysis of its past requirements. In doing this, the memory management system uses a model of memory usage that is part of the system itself, and is largely independent of the application. If the memory behaviour of the running program does not match the system model, we shall observe heavy degradations in the utilization of the system resources connected with storage management.

Let us refer to the page replacement algorithm, used to select a page to be swapped out of the primary memory when a new page must be loaded from the secondary memory and a free frame is lacking. A well-known example is the least recently used (LRU) algorithm, selecting the page that has been referenced least recently. With reference to a relational database system, consider a procedure that

Copyright © 2000 John Wiley & Sons, Ltd.

implements the natural join, for instance. It can be easily shown that in the execution of this procedure we shall obtain fewer page faults by replacing the page that has been referenced *most* recently [38]. This means that we are likely to suffer for poor performance if we use the LRU algorithm for the natural join. The most recently used (MRU) page replacement algorithm is a more appropriate choice.

As a second example, let us refer to a producer process using a buffer to send data elements to a consumer process. The implementation of the buffer will reserve a primary memory area for storage of the first few elements. Initially, the size of this area is kept to a minimum, e.g. a single frame. More frames will be added later if the producer inserts new elements and free space is lacking in the buffer. If space must be made free in the primary memory, a buffer portion is transferred back to the secondary memory. In a situation of this type, the elements inserted into the buffer most recently are good candidates for replacement. In fact, these elements will be referenced again only when they are read by the consumer process, and this will happen after the reading of every other buffer element. In this case, too, poor performance is likely to follow from usage of the LRU algorithm.

As a final example, let us refer to the code portion of the process address space. In a classical, automatic memory management environment, a page containing a given program fragment is loaded into the primary memory on demand, at the first reference to an instruction in this page. If primary memory space is lacking and a page must be selected for replacement, we have to face contrasting requirements. Most program components exhibit a high degree of locality of reference. For these components, the LRU algorithm is the correct choice. However, this is not the case for a program routine implementing a rarely used program feature, for instance. A long time interval is likely to separate two consecutive calls of the routine, and this is contrary to the implications of LRU. Thus, different page replacement algorithms should be used for different program portions. Behaviour of this type can hardly be achieved in the framework of automatic memory management.

Our memory management system defines a small, powerful set of operations, the page operations, that allow application programs to exert explicit control over memory management activities at three different levels: (i) allocation and deallocation of the virtual pages in the physical storage, obtained by using the `Activate` and `Deactivate` page operations; (ii) page movements between the secondary memory and the primary memory, made possible by the `OpenExternal`, `OpenInternal`, `CloseExternal`, `CloseInternal` and `Save` operations; and (iii) code and data migrations across the network, controlled by the `ChangeBlockServer` and `ChangeFrameServer` operations at the granularity level of a single page.

Application programs add appropriate calls to the page operations to exploit knowledge of their own memory reference pattern and implement a form of *manual* memory management. For a given program component, we take advantage of the page replacement algorithm that is more appropriate for this component [34,39,40]. For instance, with reference to the data area, we select the page containing the data items that are no longer in use. Then, we call `Save` and `CloseExternal` to save the page contents to the secondary memory and close this page. A subsequent call to `OpenExternal` is used to load the page containing the data items that will be referenced next. Similarly, for the code area, we load a given program module into the primary memory just before execution of this module, by using `OpenInternal` to open the page or pages storing the module. We unload the module when it is no longer in use by using `CloseInternal`, thereby freeing primary memory space. In this way, we keep the program working set to a minimum. The programmer (compiler) will insert the calls to `OpenInternal` and `CloseInternal` by taking the relationships between the program modules into account.

Copyright © 2000 John Wiley & Sons, Ltd.

Table II. Memory requirements for storage of the page tables.

Page size	Table	Entry size, bits	Table size, MB	Memory overhead, %
4 KBytes	PT	52	14	0.17
	APT	85	22	0.27
	OPT	68	0.281	0.22
32 KBytes	PT	49	1.75	0.02
	APT	79	2.5	0.03
	OPT	62	0.031	0.02

5.4. Storage requirements

Let us consider a medium-scale local area network featuring up to 4096 nodes whose processors generate 64-bit virtual addresses. In a system of this type, the virtual space is divided into 4096 partitions, node names and partition identifiers are codified in 12 bits, and each partition has a memory capacity of 2^{52} bytes. We shall refer to a node configuration representative of typical workstations, with a primary memory size of 128 MBytes and a secondary memory size of 8 GBytes. We shall consider two page sizes, 4 KBytes and 32 KBytes.

If the page size is 4 KBytes, the primary memory features 2^{15} frames and the secondary memory, 2^{21} blocks. Each virtual space partition is divided into 2^{40} pages, the size of a full page identifier P is 52 bits (12 bits for the partition identifier $P^{(\text{partition})}$ and 40 bits for the local page identifier $P^{(\text{local})}$), the size of a block name is 21 bits and that of a frame name, 15 bits. In a configuration of this type, a partition table entry occupies 52 bits, an active page table entry occupies 85 bits and an open page table entry, 68 bits (Table II). If the page table entries are byte-aligned, the total size of the memory management information for an open page is 27 bytes. If we define the memory overhead for storage of a given page table as the ratio between the size of an entry of this table and the page size, then we have memory overheads of 0.17% for the partition table, 0.27% for the active page table, and 0.22% for the open page table. Globally, the overhead for the three tables is 0.66%.

If the page size is 32 KBytes, the primary memory features 2^{12} frames and the secondary memory, 2^{18} blocks. Each virtual space partition is divided into 2^{37} pages, the size of a full page identifier is 49 bits (12 bits for the partition identifier and 37 bits for the local page identifier), the size of a block name is 18 bits and that of a page name, 12 bits. A partition table entry occupies 49 bits, an active page table entry occupies 79 bits and an open page table entry, 62 bits. Thus, the total size of the memory management information for a page is 25 bytes. The memory overhead is 0.02% for the partition tables, 0.03% for the active page tables, and 0.02% for the open page tables. Globally, the overhead for the three tables is 0.07%.

The memory requirements for page table storage can be evaluated by hypothesizing that pages are activated uniformly across the nodes. In this hypothesis, if the page size is 4 KBytes, a partition table features up to 2^{18} entries with a maximum size of 14 MBytes, an active page table features up to 2^{18}

Copyright © 2000 John Wiley & Sons, Ltd.

entries with a maximum size of 22 MBytes, and an open page table features up to 2^{12} entries with a maximum size of 288 KBytes. If the page size is 32 KBytes, then we have a partition table size of up to 1.75 MBytes, an active page table size of up to 2.5 MBytes, and an open page table size of up to 32 KBytes. These figures indicate that the open page table of a given node can be entirely stored in the primary memory of this node, whereas the partition table and the active page table will be stored in the secondary memory.

We shall now compare these memory requirements with those that characterize two significant, alternative system organizations, i.e. a distributed system concentrating all the information for memory management in a single node that we shall call the *master* node, and a single processor machine. In the first case, if pages are associated with nodes according to their identifiers in the same way as in our system, then for each open page P the master node will maintain (i) the local identifier $P^{(\text{local})}$ of this page, (ii) the name $BS(P)$ of the node reserving a secondary memory block B for P, and (iii) the name $FS(P)$ of the node reserving a primary memory frame F for P. Moreover, each node will maintain the information for the management of the memory resources of this node, in the form of an active page table and an open page table. For page P, the active page table of node $BS(P)$ will contain an entry storing pair $\{P, B\}$, and the open page table of node $FS(P)$ will contain an entry storing pair $\{P, F\}$. The resulting total size of this information is 27 bytes for 4-KByte pages and 25 bytes for 32-KByte pages. These figures indicate that no memory overhead increase follows from our full distribution of the memory management information across the nodes with respect to a configuration with a master node.

In a single processor machine, for each open page the system will maintain the full identifier P of this page together with the names B and F of the block and the frame reserved for page storage. The total size of this information is 11 bytes for 4-KByte pages and 10 bytes for 32-KByte pages. In comparison with our distributed system, the resulting increase of the global memory overhead for page table storage is 0.39% for 4-KByte pages and 0.05% for 32-KByte pages.

We may conclude that the memory requirements for page table storage are only a negligible fraction of the overall memory resources of the network nodes. The cost of distribution is low even if compared with a single-processor machine featuring a small memory capacity, equal to that of a single node.

5.5. Messages

Table III summarizes the network costs connected with the execution of the page operations. The costs are expressed in terms of the number of page messages and control messages transmitted between the nodes in the case of successful operation termination (of course, the cost of a failing operation may be significantly lower than indicated, and depends on the execution stage at which failure is detected). A salient property is that the number of messages required to accomplish a given page operation is independent of the number of nodes that form the network. This means that, for memory management, the increase of the network size does not result in any performance degradation.

The figures of Table III apply if the given page operation is issued in a node M which is not a server of the page P involved in that operation. This is a worst-case situation. For instance, most operations use a request message and a reply message to obtain the name of the block server $BS(P)$ from the partition server $PS(P)$. Of course, if M is the partition server of P, these two control messages will be saved. An important point is that, if a page operation is issued in a node that is both the partition server, the block server and the frame server of the page involved, execution of that page operation generates

Copyright © 2000 John Wiley & Sons, Ltd.

Table III. Messages exchanged in the execution of the page operations.

	Messages			
Operation	Page	Control	Control, master node	Control, page table caches
`P = Activate(S)`	0	2	2	2
`OpenExternal(P, N)`	1	6	6	3
`OpenInternal(P, N)`	1	6	6	3
`Save(P)`	1	8	6	3
`CloseExternal(P)`	0	8	5	5
`CloseInternal(P, N)`	0	2	2	2
`Deactivate(P)`	0	5	5	5
`ChangeBlockServer(P, N)`	1	6	6	6
`ChangeFrameServer(P, N)`	1	9	6	6

no message exchange at all. Thus, as far as memory management is concerned, if a process activates and opens pages in a given node, the execution of this process on that node produces no performance loss in comparison with the execution of the same process on a single processor machine.

Once again, let us consider a distributed system concentrating the information for memory management in a single master node S, and suppose that a page operation involving page P is issued in a node M different from S. Table III gives the number of control messages exchanged in a situation of this type (of course, the number of page messages is the same as in the fully distributed case). These figures can be easily derived from the description of the actions caused by the execution of each page operation, contained in Section 4. Of course, in the presence of a master node, no message will be exchanged with the block server to access the active page table if this access is only aimed at retrieving the name of the frame server, as this information is immediately available in S. In three operations, namely `Save`, `CloseExternal` and `ChangeFrameServer`, this fact leads to a reduction of the number of messages. In the other operations, the presence of a master node has no influence on the network costs.

5.6. Page table caches

As seen in Section 4, the execution of a page operation in a given node uses information items concerning page allocation in memory that may be stored in the page tables of the other nodes. In a situation of this type, remote table accesses take place, and each of these accesses implies the exchange of one request message and one reply message, with negative effects on the overall system performance. This problem can be mitigated by caching the distributed page table information, as follows. In each given node M, two *page tables caches* – the *partition table cache* PTC_M and the *active page table cache* $APTC_M$ – will be maintained at the software level. These caches contain portions of the partition tables and of the active page tables of all the nodes which have been used recently in that node.

Copyright © 2000 John Wiley & Sons, Ltd.

In detail, each entry of PTC_M can contain a full page identifier and a block server name. Whenever the name $BS(P)$ of the block server of a given page P is determined in node M, for instance, in the execution of a page operation, the pair $\{P, BS(P)\}$ is written into a free entry of PTC_M or, if no free entry is available, into an entry selected for replacement [41]. If execution of a page operation requires the name of the block server of a given page, a search is carried out in PTC_M for an entry for this page. If this search is successful, the block server name is extracted from the cache; otherwise an active page table access will take place. Similarly, each entry of $APTC_M$ can contain a full page identifier and a frame server name. Whenever the name $FS(P)$ of the block server of a given page P is determined in node M, the pair $\{P, FS(P)\}$ is written into $APTC_M$. If the execution of a page operation requires the name of the frame server of a given page, $APTC_M$ is accessed first, and, if a match is found, the frame server name is extracted from the cache.

Consider, for instance, the execution in node M of the `OpenExternal(P, N)` operation (see sub-Section 4.3). The first execution step is aimed at obtaining the name $BS(P)$ of the current block server of the page P involved in the operation. This step requires a remote access to the partition table $PT_{PS(P)}$ stored in the partition server $PS(P)$. In the presence of the page table caches, this step is preceded by a search in PTC_M for an entry for page P, and, if a match is found, the block server name is taken from the cache. Of course, if this cache access produces a miss, the remote access to $PT_{PS(P)}$ becomes mandatory. In this case, the block server name resulting from this remote access will be inserted into PTC_M, making it immediately available for any subsequent page operation involving P.

In sub-Section 4.1, we took advantage of the order imposed by the logical chain on the accesses to the page table entries relevant to the same given page to enforce sequentiality on these accesses, and this result was obtained by associating a lock flag with each partition table entry. However, when a search in a partition table cache produces a hit, the logical chain is accessed starting from the active page table rather than from the partition table. It follows that, in the presence of the page table caches, we must be able to lock the entries of the active page tables as well, and this requires a lock flag for each of these entries (of course, no lock flag is required for the entries of the open page tables, as these entries occupy the last position in the logical chain). The form of resource partitioning into three ordered classes (the entries of the partition tables, of the active page tables and of the open page tables) that is introduced by the logical chain prevents deadlock. Ordering follows from the fact that an access to a partition table entry always precedes any access to the active page tables, and an access to an active page table entry always precedes any access to the open page tables.

An important point is that, as a consequence of possible changes of the block server or of the frame server, the information contained in the page table caches may no longer be valid. Thus, the page operations must be able to comply with possible failures resulting from outdated cache read results. Recovery from a failure of this type implies one or more steps back in the logical chain, to reach the page server containing the correct value of the desired information item.

For instance, let us suppose that, while executing `OpenExternal(P, N)` in node M, the block server name $BS'(P)$ of page P contained in PTC_M is no longer valid, as a consequence of a block server change taking place after the writing of this information into the cache. In this hypothesis, the result of the cache read performed at execution step 1 will be erroneous. This lack of coherency between the cache contents and the real memory situation will be detected at step 2, when the active page table $APT_{BS'(P)}$ of node $BS'(P)$ is accessed to find the name of the block storing P. Of course, $APT_{BS'(P)}$ no longer reserves any entry for P, and the table access fails. Recovery from a situation of this type implies the repetition of execution step 1. The correct name $BS(P)$ of the current block server of P

Copyright © 2000 John Wiley & Sons, Ltd.

will be now retrieved in the partition table $PT_{PS(P)}$ of the partition server $PS(P)$. Quantity $BS(P)$ will then be used to update PTC_M.

Of course, a page table cache can be used to save an access to a page table entry only if this access does not modify the contents of this entry. Let us consider the execution in node M of the `ChangeFrameServer(P, N)` operation, for instance (see sub-Section 4.8). The first execution step is aimed at finding the name $BS(P)$ of the current block server of page P. This step can be accomplished in the partition table cache PTC_M, thereby saving two control messages. However, the subsequent access to the active page table $APT_{BS(P)}$ at execution step 2 cannot be avoided by using $APTC_M$, as this access changes the contents of the active page table APT_M to insert the name of the new frame server of page P.

Table III gives the number of messages exchanged as a consequence of the execution of each page operation in the presence of the page table caches (of course, the presence of the caches does not reduce the number of page messages). These figures can easily be derived from the description of the actions causes by the execution of each page operation, given in Section 4, in the best-case hypotheses that: (i) every cache access produces a hit; and (ii) the cache contents always reflect the present situation of the page allocation in memory. Four operations, namely `Activate`, `CloseInternal`, `Deactivate` and `ChangeBlockServer`, take no advantage of the presence of the caches. This is a consequence of the fact that these operations access the page tables for write. On the other hand, the caches reduce the network cost of the `Save` operation from eight to three messages, for instance. It should be clear that positive effects follow not only on the network traffic, but also on node throughput, owing to the fewer context switches that are necessary to suspend execution of the page operation after sending a request and to resume execution later, when the reply eventually becomes available.

A final observation concerns the fact, mentioned in sub-Section 5.1, that owing to their sizes, the partition tables and the active page tables should be stored in the secondary memory. Of course, the portions of these tables that are currently in use must be stored in the primary memory for fast processor access. This result will be simply obtained by extending the functionality of the page table caches of a given node to the storage of local information items, taken from the page tables of that node.

6. CONCLUDING REMARKS

We have approached the problem of giving physical support to a uniform storage model in a distributed system context consisting of nodes connected through a local area network. We have presented the organization of a distributed memory management system supporting data persistence according to the single address space paradigm. In our system:

- The virtual space is partitioned into fixed-size pages that are the units of storage allocation.
- The physical memory resources of the network nodes support a two-layer storage hierarchy in which the distributed secondary memory stores the contents of the pages that at present hold valid information items, and the primary memory implements a form of caching of these information items, aimed at fast processor access.
- The memory management functionality is fully distributed among the nodes. A logical chain, linking the nodes that contain memory management information for the same given page, allows efficient access to this information from every node.

Copyright © 2000 John Wiley & Sons, Ltd.

- A set of page operations makes it possible to control page allocation in the physical memory as well as the page movements across the memory hierarchy.

We have obtained the following results:

- The memory overhead due to the persistent memory system is low. We have analysed this overhead taking a node configuration into consideration, that is representative of a typical workstation. No memory overhead increase follows from full distribution of the memory management information among the nodes with respect to an organization concentrating this information in a single master node. The cost of distribution is low even if compared with a single-processor machine.
- The network traffic produced by the execution of the page operations is low. The absence of a master node has little influence on the network costs. Significant reductions of the network traffic can be obtained by introducing forms of caching of the memory management information. The caching activities can be effectively implemented at the software level, leading to no increase in hardware costs.
- The number of messages required to locate the physical memory areas reserved for a given page is independent of the past movements of the page in the distributed storage. Locating a page in memory never requires a message broadcast. Both these important benefits follow from the logical chain organization. In a different approach, the action of locating a page causes the sending of a number of messages that varies according to the past page ownership changes, and possibly implies the broadcast of a request across the network [12].
- The system is highly scalable with respect to the network size, as far as network traffic is concerned. The number of messages required to accomplish a given page operation does not increase if the network size increases as a consequence of the addition of new nodes.
- The page operations make it possible to profit from knowledge of the storage reference pattern of the application program in both the code and the data areas. The form of manual memory management that can be carried out by using these operations is in contrast with the demand-paged memory management often adopted in single address space systems, implemented at the level of the operating system kernel [11–13]. Forms of software control over the storage hierarchy have been proposed in the past with reference to single-processor architectures [39,42–44]. The system we have presented in this paper extends the application of this concept to a distributed framework.
- The `ChangeBlockServer` and `ChangeFrameServer` page operations allow application programs to control code and data distribution across the network. These operations can be used to implement both process migration and object migration [45]. The network cost of a page operation, expressed in terms of the number of messages exchanged, is independent of the present network location of the page involved in this operation. It follows that, as far as page management is concerned, redistribution of code and data among the network nodes leads to no increase in the network traffic.
- Application programs have no explicit control of the physical addresses in the memory devices. Let us refer to the secondary memory, for instance. In making a page active using the `Activate` page operation, the program states the node that will reserve a secondary memory block for this page; however, the program cannot assign a *specific* block to the new page. As seen in sub-Section 4.2, this assignment is made by `Activate` autonomously, as part of the actions

Copyright © 2000 John Wiley & Sons, Ltd.

involved in the execution of this page operation. In a different approach, programs have the ability to reserve specific portions of the physical storage [39]. In our opinion, the assignment of real addresses is a low-level activity requiring a degree of knowledge of the overall system state that is usually not available to the application program. We allow the program to take advantage of local optimizations of storage utilization at the level of the management of its own memory space. On the other hand, we consider global optimizations of memory usage to be a prerogative of the memory management system, at the level of the internal algorithms of the page operations.

A fully operational prototype of our memory management system has been implemented and tested [46,47]. Our distributed environment consists of a local area network that connects workstations running the Unix operating system. The prototype is written in the C++ programming language [48], and takes advantage of the Unix facilities for process management and network communication. The prototype allowed us to assess the coding effort for system development. We found an average size of the page operations of 133 lines of source code, the largest operation being `Save` (207 lines), and the smallest, `CloseInternal` (45 lines). The average code size increase due to the addition of the functionality of page table caching, as illustrated in sub-Section 5.6, is 4.2%. These figures show that development costs are moderate.

The idea of data persistence in a single address space context is certainly not new. In our opinion, it deserves careful consideration in distributed systems, where the advantages resulting from a uniform storage model are even more evident. Our work demonstrates that these advantages can be gained in a fully distributed environment with little performance penalty compared to alternative, centralized system configurations. In this respect, we hope that our work will have a significant impact in this direction.

ACKNOWLEDGEMENTS

The work described in this paper has been carried out under the financial support of the Italian Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the MOSAICO Project ('Design methodologies and tools of high performance systems for distributed applications').

REFERENCES

1. Pose RD. Capability based, tightly coupled multiprocessor hardware to support a persistent global virtual memory. *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, vol. II, Kailua-Kona, Hawaii, January 1989; pp. 36–45.
2. Appel AW, Li K. Virtual memory primitives for user programs. *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, April 1991. *Comput. Archit. News* 1991; **19**(2):96–107.
3. Hollins M, Rosenberg J, Hitchens M. Capability based protection in a persistent object-based programming language. *Advances in Modular Languages, Proceedings of the Joint Modular Languages Conference*, University of Ulm, Germany, September 1994; pp. 457–470.
4. Lindstrom A, Dearle A, di Bona R, Norris S, Rosenberg J, Vaughan F. Persistence in the Grasshopper kernel. *Proceedings of the Eighteenth Australasian Computer Science Conference*, Glenelg, South Australia, February 1995; pp. 329–338.
5. Millard BR, Dasgupta P, Rao S, Kuramkote R. Run-time support and storage management for memory-mapped persistent objects. *Proceedings of the Thirteenth International Conference on Distributed Computing Systems*, Pittsburgh, Pennsylvania, May 1993; pp. 508–515.

Copyright © 2000 John Wiley & Sons, Ltd.

6. Reitenspiess M. An architecture supporting security and persistent object storage. *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistance of Information*, Bremen, West Germany, May 1990; pp. 202–214.
7. Rosenberg J, Dearle A, Hulse D, Lindström A, Norris S. Operating system support for persistant and recoverable computations. *Commun. ACM* 1996; **39**(9):62–69.
8. Chase JS, Levy HM, Feeley MJ, Lazowska ED. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.* 1994; **12**(4):271–307.
9. Koldinger EJ, Chase JS, Eggers SJ. Architectural support for single address space operating systems. *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 1992. *SIGARCH Comput. Arch. News* 1992; **20**:175–186 (special issue).
10. Kotz D, Crow P. The expected lifetime of 'single-address-space' operating systems. *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Nashville, TN, May 1994. *Perform. Eval. Rev.* 1994; **22**(1):161–170.
11. Wilkinson T, Murray K. Evaluation of a distributed single address space operating system. *Proceedings of the Sixteenth International Conference on Distributed Computing Systems*, Hong Kong, May 1996; pp. 494–501.
12. Heiser G, Elphinstone K, Vochteloo J, Russell S, Liedtke J. The Mungi single-address-space operating system. *Softw.—Pract. Exp.* 1998; **28**(9):901–928.
13. Skousen A, Miller D. Using a single address space operating system for distributed computing and high performance. *Proceedings of the Eighteenth IEEE International Performance, Computing, and Communications Conference*, Piscataway, NJ, February 1999; pp. 8–14.
14. Jones R, Lins R. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*; Wiley: Chichester, 1996.
15. Huck J, Hays J. Architectural support for translation table management in large address space machines. *Proceedings of the Twelfth Annual International Symposium on Computer Architecture*, San Diego, California, May 1993; pp. 39–50.
16. Liedtke J, Elphinstone K. Guarded page tables on the MIPS R4600. TR-9503, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, November 1995. Available at ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/9503.ps.Z.
17. Talluri M, Hill MD, Khalidi YA. A new page table for 64-bit address spaces. *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, December 1995; pp. 215–231.
18. Chase JS, Levy HM, Lazowska ED, Baker-Harvey M. Lightweight shared objects in a 64-bit operating system. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, Canada, October 1992. *SIGPLAN Not.* 1992; **27**(10):397–413.
19. Koelbel C, Mehrotra P, van Rosendale J. Supporting shared data structures on distributed memory architectures. *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, Seattle, Washington, March 1990. *SIGPLAN Not.* 1990; **25**(3):177–186.
20. Lux W. Adaptable object migration: concept and implementation. *Oper. Syst. Rev.* 1995; **29**(2):54–69.
21. Schrimpf H. Migration of processes, files, and virtual devices in the MDX operating system. *Oper. Syst. Rev.* 1995; **29**(2):70–81.
22. Bershad BN, Anderson TE, Lazowska ED, Levy HM. Lightweight remote procedure call. *ACM Trans. Comput. Syst.* 1990; **8**(1):37–55.
23. Thekkath CA, Levy HM, Lazowska ED. Separating data and control transfer in distributed operating systems. *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1994; pp. 2–11.
24. Bartoli A. A novel approach to marshalling. *Softw.—Pract. Exp.* 1997; **27**(1):63–85.
25. Roush ET, Campbell RH. Fast dynamic process migration. *Proceedings of the Sixteenth International Conference on Distributed Computing Systems*, Hong Kong, May 1996; pp. 637–645.
26. Ramachandran U, Khalidi MYA. An implementation of distributed shared memory. *Softw.—Pract. Exp.* 1991; **21**(5):443–464.
27. Tremblay M, O'Connor JM. UltraSparc I: a four-issue processor supporting multimedia. *IEEE Micro* 1996; **16**(2):42–50.
28. Yeager KC. The MIPS R10000 superscalar microprocessor. *IEEE Micro* 1996; **16**(2):28–40.
29. Edmondson JH, Rubinfeld P, Preston R, Rajagopalan V. Superscalar instruction execution in the 21164 Alpha microprocessor. *IEEE Micro* 1995; **15**(2):33–43.
30. Bartoli A, Mullender SJ, van der Valk M. Wide-address spaces – exploring the design space. *Oper. Syst. Rev.* 1993; **27**(1):11–17.
31. Carter NP, Keckler SW, Dally WJ. Hardware support for fast capability-based addressing. *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 1994; pp. 319–327.
32. Atkinson MP, Bailey PJ, Chisholm KJ, Cockshott PW, Morrison R. An approach to persistent programming. *Comput. J.* 1983; **26**(4):360–365.

Copyright © 2000 John Wiley & Sons, Ltd.

33. Rosenberg J, Keedy JL, Abramson D. Addressing mechanisms for large virtual memories. *Comput. J.* 1992; **35**(4):369–375.
34. Bartoli A, Dini G, Lopriore L. Application-controlled memory management in a single address space environment. *MOSAICO Project Technical Report PI-DII/2/98*, June 1998. Available from the authors on request to lopriore@iet.unipi.it.
35. Carter JB, Bennett JK, Zwaenepoel W. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Trans. Comput. Syst.* 1995; **13**(3):205–243.
36. Protić J, Tomašević M, Milutinović V. Distributed shared memory: concepts and systems. *IEEE Parallel Distrib. Technol.* 1996; **4**(2):63–79.
37. Stumm M, Zhou S. Algorithms implementing distributed shared memory. *IEEE Computer* 1990; **23**(5):54–64.
38. Faloutsos C, Ng R, Sellis T. Flexible and adaptable buffer management techniques for database management systems. *IEEE Trans. Comput.* 1995; **44**(4):546–560.
39. Engler DR, Gupta SK, Kaashoek MF. AVM: application-level virtual memory. *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, Los Alamitos, CA, 1995; pp. 72–77.
40. Liedtke J. Toward real microkernels. *Commun. ACM* 1996; **39**(9):70–77.
41. Karedla R, Love JS, Wherry BG. Caching strategies to improve disk system performance. *Computer* 1994; **27**(3):38–46.
42. Corsini P, Lopriore L. An implementation of storage management in capability environments. *Softw.—Pract. Exp.* 1995; **25**(5):501–520.
43. Harty K, Cheriton DR. Application-controlled physical memory using external page-cache management. *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, Massachusetts, October 1992. *SIGPLAN Not.* 1992; **27**(9):187–197.
44. Lee PCH, Chang R-C, Chen MC. *Hipec*: a system for application-customized virtual-memory caching maangement. *Softw.—Pract. Exp.* 1997; **27**(5):547–571.
45. Lopriore L. Object and process migration in a single-address-space distributed system. *Microprocessors and Microsystems*, to appear.
46. Dini G, Lettieri G, Lopriore L. Recoverable-persistence in a distributed single address space. *Proceedings of the Seventeenth IASTED International Conference on Applied Informatics*, Innsbruck, Austria, February 1999; pp. 477–480.
47. Dini G, Lettieri G, Lopriore L. Implementing a distributed single address space in the presence of failures. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, June 1999; pp. 355–361.
48. Stroustrup B. *The C++ Programming Language* Third Edition; Addison-Wesley: Reading, Massachusetts, 1997.

Copyright © 2000 John Wiley & Sons, Ltd.