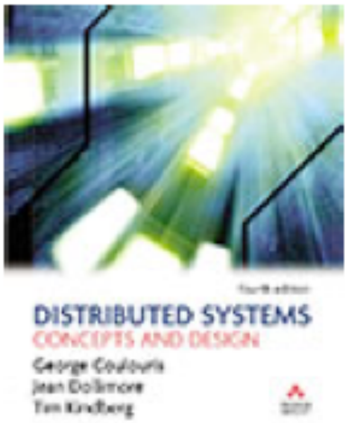


Archive material from *Edition 4* of Distributed Systems: Concepts and Design

CHAPTER CDK4–18

Originally published as pp. 749-781 of Coulouris, Dolllimore and Kindberg, Distributed Systems, Edition 4, 2005.

DISTRIBUTED SHARED MEMORY

This chapter describes distributed shared memory (DSM), an abstraction used for sharing data between processes in computers that do not share physical memory. The motivation for DSM is that it allows a shared memory programming model to be employed, which has some advantages over message-based models. For example, programmers do not have to marshal data items in DSM.

A central problem in implementing DSM is how to achieve good performance that is retained as systems scale to large numbers of computers. Accesses to DSM involve potential underlying network communication. Processes competing for the same or neighbouring data items may cause large amounts of communication to occur. The amount of communication is strongly related to the consistency model of a DSM – the model that determines which of possibly many written values will be returned when a process reads from a DSM location.

The chapter discusses DSM design issues such as the consistency model and implementation issues such as whether copies of the same data item are invalidated or updated when one copy is written. It goes on to discuss invalidation protocols in more detail. Finally, it describes release consistency – a relatively weak consistency model that is adequate for many purposes and relatively cheap to implement.

18.1 Introduction

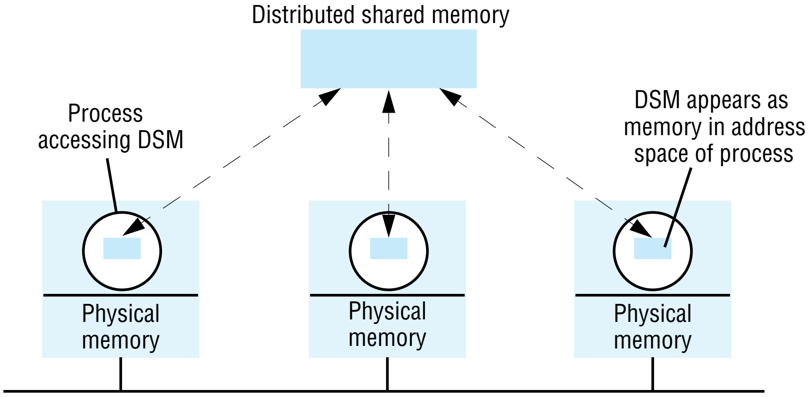
Distributed shared memory (DSM) is an abstraction used for sharing data between computers that do not share physical memory. Processes access DSM by reads and updates to what appears to be ordinary memory within their address space. However, an underlying runtime system ensures transparently that processes executing at different computers observe the updates made by one another. It is as though the processes access a single shared memory, but in fact the physical memory is distributed (see Figure 18.1).

The main point of DSM is that it spares the programmer the concerns of message passing when writing applications that might otherwise have to use it. DSM is primarily a tool for parallel applications or for any distributed application or group of applications in which individual shared data items can be accessed directly. DSM is in general less appropriate in client-server systems, where clients normally view server-held resources as abstract data and access them by request (for reasons of modularity and protection). However, servers can provide DSM that is shared between clients. For example, memory-mapped files that are shared and for which some degree of consistency is maintained are a form of DSM. (Mapped files were introduced with the MULTICS operating system [Organick 1972].)

Message passing cannot be avoided altogether in a distributed system: in the absence of physically shared memory, the DSM runtime support has to send updates in messages between computers. DSM systems manage replicated data: each computer has a local copy of recently accessed data items stored in DSM, for speed of access. The problems of implementing DSM are related to those discussed in Chapter 15, as well as those of caching shared files discussed in Chapter 8.

One of the first notable examples of a DSM implementation was the Apollo Domain file system [Leach *et al.* 1983], in which processes hosted by different workstations share files by mapping them simultaneously into their address spaces. This example shows that distributed shared memory can be persistent. That is, it may outlast the execution of any process or group of processes that accesses it and be shared by different groups of processes over time.

Figure 18.1 The distributed shared memory abstraction


Distributed shared memory
Process accessing DSM
DSM appears as memory in address space of process
Physical memory
Physical memory
Physical memory

The significance of DSM first grew alongside the development of shared-memory multiprocessors (see Section 6.3). Much research has gone into investigating algorithms suitable for parallel computation on these multiprocessors. At the hardware architectural level, developments include both caching strategies and fast processor-memory interconnections, aimed at maximizing the number of processors that can be sustained while achieving fast memory access latency and throughput [Dubois *et al.* 1988]. Where processes are connected to memory modules over a common bus, the practical limit is in the order of 10 processors before performance degrades drastically due to bus contention. Processors sharing memory are commonly constructed in groups of four, sharing a memory module over a bus on a single circuit board. Multiprocessors with up to 64 processors in total are constructed from such boards in a *Non-Uniform Memory Access* (*NUMA*) architecture. This is a hierarchical architecture in which the four-processor boards are connected using a high-performance switch or higher-level bus. In a NUMA architecture, processors see a single address space containing all the memory of all the boards. But the access latency for on-board memory is less than that for a memory module on a different board – hence the name of this architecture.

In *distributed memory multiprocessors* and clusters of off-the-shelf computing components (see Section 6.3), the processors do not share memory but are connected by a very high-speed network. These systems, like general-purpose distributed systems, can scale to much greater numbers of processors than a shared-memory multiprocessor's 64 or so. A central question that has been pursued by the DSM and multiprocessor research communities is whether the investment in knowledge of shared memory algorithms and the associated software can be directly transferred to a more scalable distributed memory architecture.

18.1.1 Message passing versus DSM

As a communication mechanism, DSM is comparable with message passing rather than with request-reply-based communication, since its application to parallel processing, in particular, entails the use of asynchronous communication. The DSM and message passing approaches to programming can be contrasted as follows:

Programming model: Under the message passing model, variables have to be marshalled from one process, transmitted and unmarshalled into other variables at the receiving process. By contrast, with shared memory the processes involved share variables directly, so no marshalling is necessary – even of pointers to shared variables – and thus no separate communication operations are necessary. Most implementations allow variables stored in DSM to be named and accessed similarly to ordinary unshared variables. In favour of message passing, on the other hand, is that it allows processes to communicate while being protected from one another by having private address spaces, whereas processes sharing DSM can, for example, cause one another to fail by erroneously altering data. Furthermore, when message passing is used between heterogeneous computers, marshalling takes care of differences in data representation; but how can memory be shared between computers with, for example, different integer representations?

Synchronization between processes is achieved in the message model through message passing primitives themselves, using techniques such as the lock server implementation discussed in Chapter 13. In the case of DSM, synchronization is via normal constructs for shared-memory programming such as locks and semaphores (although these require different implementations in the distributed memory environment). Chapter 6 briefly discussed such synchronization objects in the context of programming with threads.

Finally, since DSM can be made persistent, processes communicating via DSM may execute with non-overlapping lifetimes. A process can leave data in an agreed memory location for the other to examine when it runs. By contrast, processes communicating via message passing must execute at the same time.

Efficiency: Experiments show that certain parallel programs developed for DSM can be made to perform about as well as functionally equivalent programs written for message passing platforms on the same hardware [Carter *et al*. 1991] – at least in the case of relatively small numbers of computers (ten or so). However, this result cannot be generalized. The performance of a program based on DSM depends upon many factors, as we shall discuss below – particularly the pattern of data sharing (such as whether an item is updated by several processes).

There is a difference in the visibility of costs associated with the two types of programming. In message passing, all remote data accesses are explicit and therefore the programmer is always aware of whether a particular operation is in-process or involves the expense of communication. Using DSM, however, any particular read or update may or may not involve communication by the underlying runtime support. Whether it does or not depends upon such factors as whether the data have been accessed before and the sharing pattern between processes at different computers.

There is no definitive answer as to whether DSM is preferable to message passing for any particular application. DSM is a promising tool whose ultimate status depends upon the efficiency with which it can be implemented.

18.1.2 Implementation approaches to DSM

Distributed shared memory is implemented using one or a combination of specialized hardware, conventional paged virtual memory or middleware:

Hardware: Shared-memory multiprocessor architectures based on a NUMA architecture (for example, Dash [Lenoski *et al*. 1992] and PLUS [Bisiani and Ravishankar 1990]) rely on specialized hardware to provide the processors with a consistent view of shared memory. They handle memory LOAD and STORE instructions by communicating with remote memory and cache modules as necessary to store and retrieve data. This communication is over a high-speed interconnection which is analogous to a network. The prototype Dash multiprocessor has 64 nodes connected in a NUMA architecture.

Paged virtual memory: Many systems, including Ivy [Li and Hudak 1989], Munin [Carter *et al*. 1991], Mirage [Fleisch and Popek 1989], Clouds [Dasgupta *et al*. 1991] (see www.cdk4.net/oss), Choices [Sane *et al*. 1990], COOL [Lea *et al*. 1993] and Mether [Minnich and Farber 1989], implement DSM as a region of virtual memory occupying the same address range in the address space of every participating process.

Figure 18.2 Mether system program

```
  #include "world.h"
  struct shared { int a, b; };

Program Writer:
    main()
    {
        struct shared *p;
        methersetup();                      /* Initialize the Mether runtime */
        p = (struct shared *)METHERBASE;
                                    /* overlay structure on METHER segment */
        p->a = p->b = 0;                    /* initialize fields to zero */
        while(TRUE){                        /* continuously update structure fields */
            p –>a = p –>a + 1;
            p –>b = p –>b - 1;
        }
    }

Program Reader:
    main()
    {
        struct shared *p;
        methersetup();
        p = (struct shared *)METHERBASE;
        while(TRUE){                        /* read the fields once every second */
            printf("a = %d, b = %d\n", p –>a, p –>b);
            sleep(1);
        }
    }
```

This type of implementation is normally only suited to a collection of homogeneous computers, with common data and paging formats.

Middleware: Some languages such as Orca [Bal *et al.* 1990], and middleware such as Linda [Carriero and Gelernter 1989] and its derivatives JavaSpaces [Bishop and Warren 2003] and TSpaces [Wyckoff *et al.* 1998], support forms of DSM without any hardware or paging support, in a platform-neutral way. In this type of implementation, sharing is implemented by communication between instances of the user-level support layer in clients and servers. Processes make calls to this layer when they access data items in DSM. The instances of this layer at the different computers access local data items and communicate as necessary to maintain consistency.

This chapter concentrates on the use of software to implement DSM on standard computers. Even with hardware support, high-level software techniques may be used to minimize the amount of communication between components of a DSM implementation.

The page-based approach has the advantage of imposing no particular structure on the DSM, which appears as a sequence of bytes. In principle, it enables programs designed for a shared-memory multiprocessor to run on computers without shared memory, with little or no adaptation. Microkernels such as Mach and Chorus provide native support for DSM (and other memory abstractions – the Mach virtual memory facilities are described in www.cdk4.net/mach). Page-based DSM is more usually implemented largely at user level to take advantage of the flexibility that that provides. The implementation utilizes kernel support for user-level page fault handlers. UNIX and some variants of Windows provide this facility. Microprocessors with 64-bit address spaces widen the scope for page-based DSM by relaxing constraints on address space management [Bartoli *et al*. 1993].

The example in Figure 18.2 is of two C programs, *Reader* and *Writer*, which communicate via the page-based DSM provided by the Mether system [Minnich and Farber 1989]. *Writer* updates two fields in a structure overlaid upon the beginning of the Mether DSM segment (beginning at address *METHERBASE*) and *Reader* periodically prints out the values it reads from these fields.

The two programs contain no special operations; they are compiled into machine instructions that access a common range of virtual memory addresses (starting at *METHERBASE*). Mether ran over conventional Sun workstation and network hardware.

The middleware approach is quite different to the use of specialized hardware and paging in that it is not intended to utilize existing shared-memory code. Its significance is that it enables us to develop higher-level abstractions of shared objects, rather than shared memory locations.

18.2 Design and implementation issues

This section discusses design and implementation options concerning the main features that characterize a DSM system. These are the structure of data held in DSM; the synchronization model used to access DSM consistently at the application level; the DSM consistency model, which governs the consistency of data values accessed from different computers; the update options for communicating written values between computers; the granularity of sharing in a DSM implementation; and the problem of thrashing.

18.2.1 Structure

In Chapter 15, we considered systems that replicate a collection of objects such as diaries and files. Those systems enable client programs to perform operations upon the objects as though there was only one copy of each object, but in reality they may be accessing different physical replicas. The systems make guarantees about the extent to which the replicas of the objects are allowed to diverge.

A DSM system is just such a replication system. Each application process is presented with some abstraction of a collection of objects, but in this case the 'collection' looks more or less like memory. That is, the objects can be addressed in some fashion or other. Different approaches to DSM vary in what they consider to be an 'object' and in how objects are addressed. We consider three approaches, which view DSM as being composed respectively of contiguous bytes, language-level objects or immutable data items.

Byte-oriented ◊ This type of DSM is accessed as ordinary virtual memory – a contiguous array of bytes. It is the view illustrated above by the Mether system. It is also the view of many other DSM systems, including Ivy, which we discuss in Section 18.3. It allows applications (and language implementations) to impose whatever data structures they want on the shared memory. The shared objects are directly addressible memory locations (in practice, the shared locations may be multi-byte words rather than individual bytes). The only operations upon those objects are *read* (or LOAD) and *write* (or STORE). If x and y are two memory locations, then we denote instances of these operations as follows:

$R(x)a$ – a *read* operation that reads the value a from location x.

$W(x)b$ – a *write* operation that stores value b at location x.

An example execution is $W(x)1$, $R(x)2$. This process writes the value 1 to location x and then reads the value 2 from it. Some other process must have written the value 2 to that location meanwhile.

Object-oriented ◊ The shared memory is structured as a collection of language-level objects with higher-level semantics than simple *read/write* variables, such as stacks and dictionaries. The contents of the shared memory are changed only by invocations upon these objects and never by direct access to their member variables. An advantage of viewing memory in this way is that object semantics can be utilized when enforcing consistency. Orca views DSM as a collection of shared objects and automatically serializes operations upon any given object.

Immutable data ◊ Here DSM is viewed as a collection of immutable data items that processes can read, add to and remove from. Examples include Agora [Bisiani and Forin 1988] and, more significantly, Linda and its derivatives, TSpaces and JavaSpaces.

Linda-type systems provide the programmer with collections of tuples called a *tuple space* (see Section 16.3.1). Tuples consist of a sequence of one or more typed data fields such as <"fred", 1958>, <"sid", 1964> and <4, 9.8, "Yes">. Any combination of types of tuples may exist in the same tuple space. Processes share data by accessing the same tuple space: they place tuples in tuple space using the *write* operation and read or extract them from tuple space using the *read* or *take* operation. The *write* operation adds a tuple without affecting existing tuples in the space. The *read* operation returns the value of one tuple without affecting the contents of the tuple space. The *take* operation also returns a tuple, but in this case it also removes the tuple from the tuple space.

When reading or taking a tuple from tuple space, a process provides a tuple specification and the tuple space returns any tuple that matches that specification – this is a type of associative addressing. To enable processes to synchronize their activities, the *read* and *take* operations both block until there is a matching tuple in the tuple space. A tuple specification includes the number of fields and the required values or types of the fields. For example, *take*(<String, integer>) could extract either <"fred", 1958> or <"sid", 1964>; *take*(<String, 1958>) would extract only <"fred", 1958> of those two.

In Linda, no direct access to tuples in tuple space is allowed and processes have to replace tuples in the tuple space instead of modifying them. Suppose, for example, that a set of processes maintains a shared counter in tuple space. The current count (say 64) is in the tuple <“counter”, 64>. A process must execute code of the following form in order to increment the counter in a tuple space *myTS*:

```
<s, count> := myTS.take(<“counter”, integer>);
myTS.write(<“counter”, count+1>);
```

The reader should check that race conditions cannot arise, because *take* extracts the counter tuple from tuple space.

18.2.2 Synchronization model

Many applications apply constraints concerning the values stored in shared memory. This is as true of applications based on DSM as it is of applications written for shared-memory multiprocessors (or indeed for any concurrent programs that share data, such as operating system kernels and multi-threaded servers). For example, if *a* and *b* are two variables stored in DSM, then a constraint might be that $a = b$ always. If two or more processes execute the following code:

```
a := a + 1;
b := b + 1;
```

then an inconsistency may arise. Suppose *a* and *b* are initially zero and that process 1 gets as far as setting *a* to 1. Before it can increment *b*, process 2 sets *a* to 2 and *b* to 1. The constraint has been broken. The solution is to make this code fragment into a critical section: to synchronize processes to ensure that only one may execute it at a time.

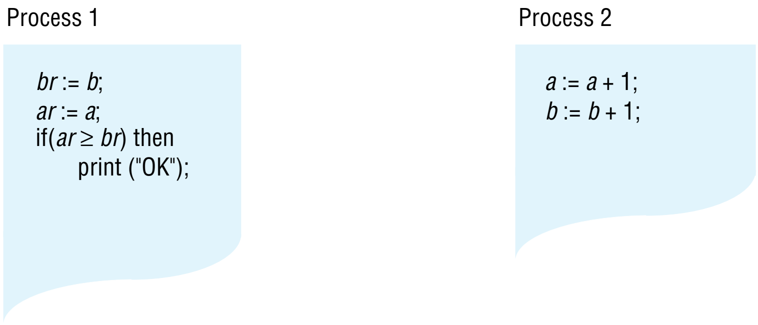
In order to use DSM, then, a distributed synchronization service needs to be provided, which includes familiar constructs such as locks and semaphores. Even when DSM is structured as a set of objects, the implementors of the objects have to be concerned with synchronization. Synchronization constructs are implemented using message passing (see Chapter 13 for a description of a lock server). Special machine instructions such as *testAndSet*, which are used for synchronization in shared-memory multiprocessors, are applicable to page-based DSM, but their operation in the distributed case may be very inefficient. DSM implementations take advantage of application-level synchronization to reduce the amount of update transmission. The DSM then includes synchronization as an integrated component.

18.2.3 Consistency model

As we described in Chapter 15, the issue of consistency arises for a system such as DSM, which replicates the contents of shared memory by caching it at separate computers. In the terminology of Chapter 15, each process has a local replica manager, which holds cached replicas of objects. In most implementations, data is read from local replicas for efficiency, but updates have to be propagated to the other replica managers.

The local replica manager is implemented by a combination of middleware (the DSM runtime layer in each process) and the kernel. It is usual for middleware to perform the majority of DSM processing. Even in a page-based DSM implementation, the kernel usually provides only basic page mapping, page-fault handling and communication

Figure 18.3 Two processes accessing shared variables



Process 1

```
br := b;
ar := a;
if(ar ≥ br) then
      print ("OK");
```

Process 2

```
a := a + 1;
b := b + 1;
```

mechanisms and middleware is responsible for implementing the page-sharing policies. If DSM segments are persistent, then one or more storage servers (for example, file servers) will also act as replica managers.

In addition to caching, a DSM implementation may buffer updates and thus amortize communication costs by spreading them over multiple updates. We saw a similar approach to amortizing communication costs in the gossip architecture of Chapter 15.

A *memory consistency* model [Mosberger 1993] specifies the consistency guarantees that a DSM system makes about the values that processes read from objects, given that they actually access a replica of each object and that multiple processes may update the objects. Note that this is different from the higher-level, application-specific notion of consistency discussed under the heading of application synchronization above.

Cheriton [1985] describes how forms of DSM can be envisaged for which a considerable degree of inconsistency is acceptable. For example, DSM might be used to store the loads of computers on a network in order that clients can select the least-loaded computers for running applications. Since such information is by its nature liable to become inaccurate on relatively small timescales, it would be a waste of effort to keep it consistent at all times for all computers in the system.

Most applications do, however, have stronger consistency requirements. Care must be taken to give programmers a model that conforms to reasonable expectations of the way memory should behave. Before describing memory consistency requirements in more detail, it is helpful first to look at an example.

Consider an application in which two processes access two variables, *a* and *b* (Figure 18.3), which are initialized to zero. Process 2 increments *a* and *b*, in that order. Process 1 reads the values of *b* and *a* into local variables *br* and *ar*, in that order. Note that there is no application-level synchronization. Intuitively, process 1 should expect to see one of the following combinations of values, depending upon the points at which the read operations applied to *a* and *b* (implied in the statements $br := b$ and $ar := a$) occur with respect to process 2's execution: $ar = 0$, $br = 0$; $ar = 1$, $br = 0$; $ar = 1$, $br = 1$. In other words, the condition $ar \geq br$ should always be satisfied and process 1 should print 'OK'. However, a DSM implementation might deliver the updates to *a* and *b* out of order to the replica manager for process 1, in which case the combination $ar = 0$, $br = 1$ could occur.

The reader's immediate reaction to the example just given is probably that the DSM implementation, which reverses the order of two updates, is incorrect. If process 1 and process 2 execute together at a single-processor computer, we would assume that the memory subsystem was malfunctioning. However, it may be a correct implementation, in the distributed case, of a consistency model that is weaker than what many of us would intuitively expect, but that nonetheless can be useful and is relatively efficient.

Mosberger [1993] delineates a range of models that have been devised for shared-memory multiprocessors and software DSM systems. The main consistency models that can be practically realized in DSM implementations are sequential consistency and models that are based on weak consistency.

The central question to be asked in order to characterize a particular memory consistency model is this: when a read access is made to a memory location, which write accesses to the location are candidates whose values could be supplied to the read? At the weakest extreme, the answer is: any write that was issued before the read. This model would be obtained if replica managers could delay propagating updates to their peers indefinitely. It is too weak to be useful.

At the strongest extreme, all written values are instantaneously available to all processes: a read returns the most recent write at the time that the read takes place. This definition is problematic in two respects. First, neither writes nor reads take place at a single point in time, so the meaning of 'most recent' is not always clear. Each type of access has a well-defined point of issue, but they complete at some later time (for example, after message passing has taken place). Second, Chapter 11 showed that there are limits to how closely clocks can be synchronized in a distributed system. So it is not always possible to determine accurately whether one event occurred before another.

Nonetheless, this model has been specified and studied. The reader may already have recognized it: it is what we called linearizability in Chapter 15. Linearizability is more usually called *atomic consistency* in the DSM literature. We now restate the definition of linearizability from Chapter 15.

A replicated shared object service is said to be linearizable if *for any execution* there is some interleaving of the series of operations issued by all the clients that satisfies the following two criteria:

L1: The interleaved sequence of operations meets the specification of a (single) correct copy of the objects.

L2: The order of operations in the interleaving is consistent with the real times at which the operations occurred in the actual execution.

This definition is a general one that applies to any system containing shared replicated objects. We can be more specific now, since we know that we are dealing with a shared memory. Consider the simple case where the shared memory is structured as a set of variables that may be read or written. The operations are all reads and writes, which we introduced a notation for in Section 18.2.1: a read of value a from variable x is denoted $R(x)a$; a write of value b to variable x is denoted $W(x)b$. We can now express the first criterion L1 in terms of variables (the shared objects) as follows:

L1': The interleaved sequence of operations is such that if $R(x)a$ occurs in the sequence, then either the last write operation that occurs before it in the interleaved sequence is $W(x)a$, or no write operation occurs before it and *a* is the initial value of *x*.

This criterion states our intuition that a variable can only be changed by a write operation. The second criterion for linearizability, L2, remains the same.

Sequential consistency ◊ Linearizability is too strict for most practical purposes. The strongest memory model for DSM that is used in practice is *sequential consistency* [Lamport 1979], which we introduced in Chapter 15. We adapt Chapter 15's definition for the particular case of shared variables as follows.

A DSM system is said to be sequentially consistent if *for any execution* there is some interleaving of the series of operations issued by all the processes that satisfies the following two criteria:

SC1: The interleaved sequence of operations is such that if $R(x)a$ occurs in the sequence, then either the last write operation that occurs before it in the interleaved sequence is $W(x)a$, or no write operation occurs before it and *a* is the initial value of *x*.

SC2: The order of operations in the interleaving is consistent with the program order in which each individual client executed them.

Criterion SC1 is the same as L1'. Criterion SC2 refers to program order rather than temporal order, which is what makes it possible to implement sequential consistency.

The condition can be restated as follows: there is a virtual interleaving of all the processes' *read* and *write* operations against a single virtual image of the memory; the program order of every individual process is preserved in this interleaving and each process always reads the latest value written within the interleaving.

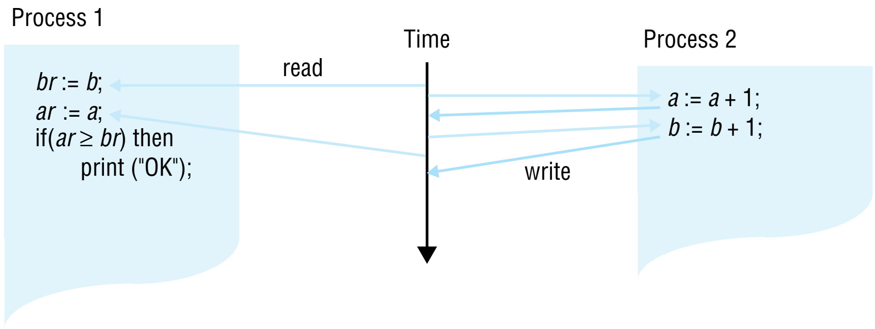
In an actual execution, memory operations may be overlapped and some updates may be ordered differently at different processes, as long as the definition's constraints are not thereby broken. Note that memory operations upon the entire DSM have to be taken into account to satisfy the conditions of sequential consistency – and not just the operations on each individual location.

The combination $ar = 0$, $br = 1$ in the above example could not occur under sequential consistency, because process 1 would be reading values that conflict with process 2's program order. An example interleaving of the processes' memory accesses in a sequentially consistent execution is shown in Figure 18.4. Once more, while this shows an actual interleaving of the read and write operations, the definition only stipulates that the execution should take place *as though* such a strict interleaving takes place.

Sequentially consistent DSM could be implemented by using a single server to hold all the shared data and by making all processes perform reads or writes by sending requests to the server, which globally orders them. This architecture is too inefficient for a DSM implementation and practical means of achieving sequential consistency are described below. Nonetheless, it remains a costly model to implement.

Coherence ◊ One reaction to the cost of sequential consistency is to settle for a weaker model with well-defined properties. *Coherence* is an example of a weaker form of consistency. Under coherence, every process agrees on the order of *write* operations to the same location, but they do not necessarily agree on the ordering of *write* operations

Figure 18.4 Interleaving under sequential consistency



to different locations. We can think of coherence as sequential consistency on a location-by-location basis. Coherent DSM can be implemented by taking a protocol for implementing sequential consistency and applying it separately to each unit of replicated data – for example, each page. The saving comes from the fact that accesses to two different pages are independent and need not delay one another, since the protocol is applied separately to them.

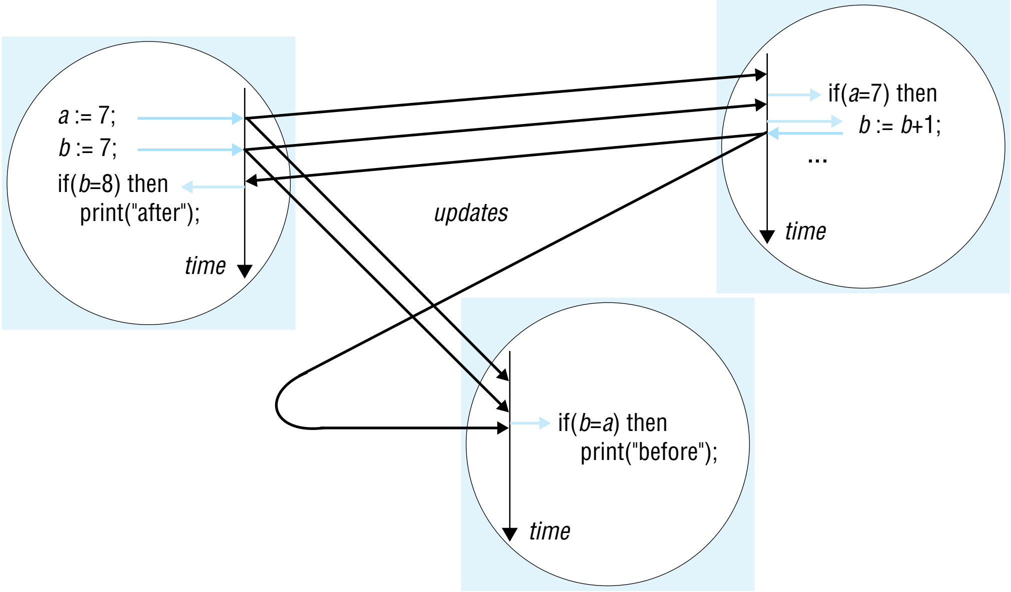
Weak consistency ◊ Dubois *et al.* [1988] developed the weak consistency model in an attempt to avoid the costs of sequential consistency on multiprocessors, while retaining the *effect* of sequential consistency. This model exploits knowledge of synchronization operations in order to relax memory consistency, while appearing to the programmer to implement sequential consistency (at least, under certain conditions that are beyond the scope of this book). For example, if the programmer uses a lock to implement a critical section, then a DSM system can assume that no other process may access the data items accessed under mutual exclusion within it. It is therefore redundant for the DSM system to propagate updates to these items until the process leaves the critical section. While items are left with 'inconsistent' values some of the time, they are not accessed at those points; the execution appears to be sequentially consistent. Adve and Hill [1990] describe a generalization of this notion called weak ordering: '(A DSM system) is weakly ordered with respect to a synchronization model if and only if it appears sequentially consistent to all software that obeys the synchronization model.' Release consistency, which is a development of weak consistency, is described in Section 18.4.

18.2.4 Update options

Two main implementation choices have been devised for propagating updates made by one process to the others: write-update and write-invalidate. These are applicable to a variety of DSM consistency models, including sequential consistency. In outline, the options are as follows:

Write-update: The updates made by a process are made locally and multicast to all other replica managers possessing a copy of the data item, which immediately modify the data read by local processes (Figure 18.5). Processes read the local copies of data items, without the need for communication. In addition to allowing multiple readers,

Figure 18.5 DSM using write-update



several processes may write the same data item at the same time; this is known as *multiple-reader/multiple-writer sharing*.

The memory consistency model that is implemented with write-update depends on several factors, mainly the multicast ordering property. Sequential consistency can be achieved by using multicasts that are totally ordered (see Chapter 12 for a definition of totally ordered multicast), which do not return until the update message has been delivered locally. All processes then agree on the order of updates. The set of reads that take place between any two consecutive updates is well defined and their ordering is immaterial to sequential consistency.

Reads are cheap in the write-update option. However, Chapter 12 showed that ordered multicast protocols are relatively expensive to implement in software. Orca uses write-update and employs the Amoeba multicast protocol [Kaashoek and Tanenbaum 1991] (see www.cdk4.net/coordination), which uses hardware support for multicast. Munin supports write-update as an option. A write-update protocol is used with specialized hardware support in the PLUS multiprocessor architecture.

Write-invalidate: This is commonly implemented in the form of multiple-reader/single-writer sharing. At any time, a data item may either be accessed in read-only mode by one or more processes, or it may be read and written by a single process. An item that is currently accessed in read-only mode can be copied indefinitely to other processes. When a process attempts to write to it, a multicast message is first sent to all other copies to invalidate them and this is acknowledged before the write can take place; the other processes are thereby prevented from reading stale data (that is, data that are not up to date). Any processes attempting to access the data item are blocked if a writer exists. Eventually, control is transferred from the writing process, and other accesses may take place once the update has been sent. The effect is to process all accesses to the item on a first-come, first-served basis. By the proof given by Lamport [1979], this scheme achieves sequential consistency. We shall see in Section 18.4 that invalidations may be delayed under release consistency.

Under the invalidation scheme, updates are only propagated when data are read and several updates can take place before communication is necessary. Against this must be placed the cost of invalidating read-only copies before a write can occur. In the multiple-reader/single-writer scheme described, this is potentially expensive. But if the read/write ratio is sufficiently high, then the parallelism obtained by allowing multiple simultaneous readers offsets this cost. Where the read/write ratio is relatively small, a single-reader/single-writer scheme can be more appropriate: that is, one in which at most one process may be granted read-only access at a time.

18.2.5 Granularity

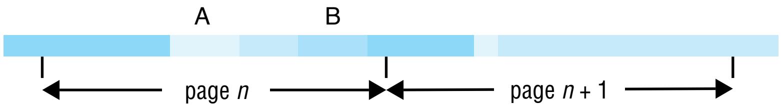
An issue that is related to the structure of DSM is the granularity of sharing. Conceptually, all processes share the entire contents of a DSM. As programs sharing DSM execute, however, only certain parts of the data are actually shared and then only for certain times during the execution. It would clearly be very wasteful for the DSM implementation always to transmit the entire contents of DSM as processes access and update it. What should be the unit of sharing in a DSM implementation? That is, when a process has written to DSM, which data does the DSM runtime send in order to provide consistent values elsewhere?

We focus here on page-based implementations, although the granularity issue does arise in other implementations (see Exercise 18.11). In a page-based DSM, the hardware supports alterations to an address space efficiently in units of pages – essentially by the placement of a new page frame pointer in the page table (see, for example, Bacon [2002] for a description of paging). Page sizes can typically range up to 8 kilobytes, so this is an appreciable amount of data that must be transmitted over a network to keep remote copies consistent when an update occurs. By default, the price of the whole page transfer must be paid whether the entire page has been updated, or just one byte of it.

Using a smaller page size – 512 bytes or 1 kilobyte say – does not necessarily lead to an improvement in overall performance. First, in cases where processes do update large amounts of contiguous data, it is better to send one large page rather than several smaller pages in separate updates, because of the fixed software overheads per network packet. Second, using a small page as the unit of distribution leads to a large number of units that must be administered separately by the DSM implementation.

To complicate matters further, processes tend to contend more for pages when the page size is large, because the likelihood that the data they access will lie within the same page increases with the page size. Consider, for example, two processes, one of which accesses only data item *A* while the other accesses only data item *B*, which lie within the same page (Figure 18.6). For the sake of concreteness, let us assume that one process reads *A* and the other updates *B*. There is no contention at the application level. However, the entire page must be transmitted between the processes, since the DSM runtime does not by default know which locations in the page have been altered. This phenomenon is known as *false sharing*: two or more processes share parts of a page, but

Figure 18.6 Data items laid out over pages



only one in fact accesses each part. In write-invalidate protocols, false sharing can lead to unnecessary invalidations. In write-update protocols, when several writers falsely share data items they may cause them to be overwritten with older versions.

In practice, the choice of the unit of sharing has to be made based on the physical page sizes available, although a unit of several contiguous pages may be taken if the page size is small. The layout of data with respect to page boundaries is an important factor in determining the number of page transfers made when a program executes.

18.2.6 Thrashing

A potential problem with write-invalidate protocols is thrashing. Thrashing is said to occur where the DSM runtime spends an inordinate amount of time invalidating and transferring shared data compared with the time spent by application processes doing useful work. It occurs when several processes compete for the same data item, or for falsely shared data items. If, for example, one process repeatedly reads a data item that another is regularly updating, then this item will be constantly transferred from the writer and invalidated at the reader. This is an example of a sharing pattern for which write-invalidate is inappropriate and write-update would be better. The next section describes the Mirage approach to thrashing, in which computers ‘own’ pages for a minimum period; Section 18.4 describes how Munin allows the programmer to declare access patterns to the DSM system so that it can choose appropriate update options for each data item and avoid thrashing.

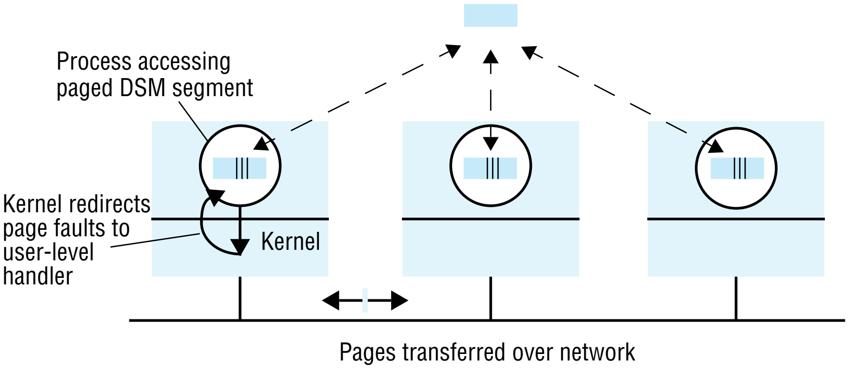
18.3 Sequential consistency and Ivy case study

This section describes methods for implementing sequentially consistent, page-based DSM. It draws upon Ivy [Li and Hudak 1989] as a case study.

18.3.1 The system model

The basic model to be considered is one in which a collection of processes shares a segment of DSM (Figure 18.7). The segment is mapped to the same range of addresses in each process, so that meaningful pointer values can be stored in the segment. The processes execute at computers equipped with a paged memory management unit. We shall assume that there is only one process per computer that accesses the DSM segment. There may in reality be several such processes at a computer. However, these could then share DSM pages directly (the same page frame can be used in the page tables used by the different processes). The only complication would be to coordinate fetching and

Figure 18.7 System model for page-based DSM


Process accessing paged DSM segment
Kernel redirects page faults to user-level handler
Kernel
Pages transferred over network

propagating updates to a page when two or more local processes access it. This description ignores such details.

Paging is transparent to the application components within processes; they can logically both read and write any data in DSM. However, the DSM runtime restricts page access permissions in order to maintain sequential consistency when processing reads and writes. Paged memory management units allow the access permissions to a data page to be set to *none*, *read-only* or *read-write*. If a process attempts to exceed the current access permissions, then it takes a read or write page fault, according to the type of access. The kernel redirects the page fault to a handler specified by the DSM runtime layer in each process. The page fault handler – which runs transparently to the application – processes the fault in a special way, to be described below, before returning control to the application. In the original DSM systems such as Ivy, the kernel itself performs much of the processing that we describe here. We shall speak of the processes themselves performing page-fault handling and communication handling. In actuality, some combination of the DSM runtime layer in the process and the kernel performs these handling functions. Usually, the in-process DSM runtime contains the most significant functionality in order than this can be reimplemented and fine-tuned without the problems associated with altering a kernel.

This description will ignore the page-fault processing that takes place as part of the normal virtual memory implementation. Apart from the fact that DSM segments compete with other segments for page frames, the implementations are independent.

The problem of write-update ◊ The previous section outlined the general implementation alternatives of write-update and write-invalidation. In practice, if the DSM is page-based, then write-update is used only if writes can be buffered. This is because standard page-fault handling is unsuited to the task of processing every single write update to a page.

To see this, suppose that every update has to be multicast to the remaining replicas. Suppose that a page has been write-protected. When a process attempts to write upon the page, it takes a page fault and a handler routine is called. This handler could, in principle, examine the faulting instruction to determine the value and address being written and multicast the update before restoring write access and returning to complete the faulting instruction.

But now that write access has been restored, subsequent updates to the page will not cause a page fault. To make every write access produce a page fault, it would be necessary for the page fault handler to set the process into TRACE mode, whereby the processor generates a TRACE exception after each instruction. The TRACE exception handler would turn off write permissions to the page and turn off TRACE mode once more. The whole exercise would be repeated when a write fault next occurred. It is clear that this method is liable to be very expensive. There would be many exceptions caused during the execution of a process.

In practice, write-update is used with page-based implementations, but only where the page is left with write permissions after an initial page fault and several writes are allowed to occur before the updated page is propagated. Munin uses this write-buffering technique. As an extra efficiency measure, Munin tries to avoid propagating the whole page – only a small part of which may have been updated. When a process first attempts to write to the page, Munin handles the page fault by taking a copy of the page and setting the copy aside before enabling write access. Later, when Munin is ready to propagate the page, it compares the updated page with the copy that it took and encodes the updates as a set of differences between the two pages. The differences often take up much less space than an entire page. Other processes generate the updated page from the pre-update copy and the set of differences.

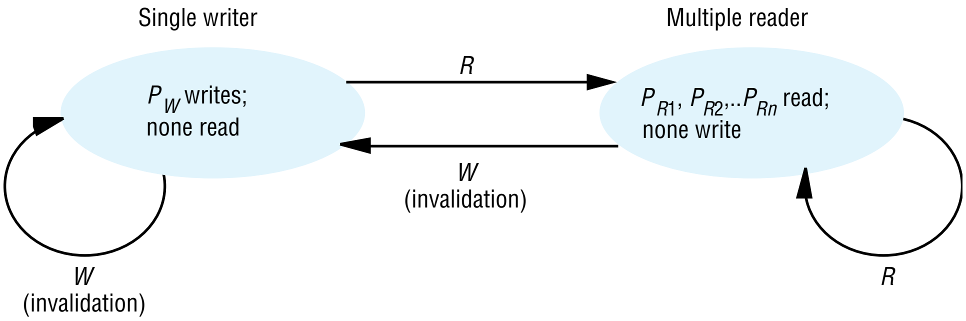
18.3.2 Write invalidation

Invalidation-based algorithms use page protection to enforce consistent data sharing. When a process is updating a page, it has read and write permissions locally; all other processes have no access permissions to the page. When one or more processes are reading the page, they have read-only permission; all other processes have no access permissions (although they may acquire read permissions). No other combinations are possible. A process with the most up-to-date version of a page p is designated as its *owner* – referred to as *owner*(p). This is either the single writer, or one of the readers. The set of processes that have a copy of a page p is called its *copy set* – referred to as *copyset*(p).

The possible state transitions are shown in Figure 18.8. When a process P_w attempts to write a page p to which it has no access or read-only access, a page fault takes place. The page-fault handling procedure is as follows:

- The page is transferred to P_w, if it does not already have an up-to-date read-only copy.
- All other copies are invalidated: the page permissions are set to no access at all members of *copyset*(p).
- *copyset*(p) := { P_w }.
- *owner*(p) := P_w.
- The DSM runtime layer in P_w places the page with read-write permissions at the appropriate location in its address space and restarts the faulting instruction.

Figure 18.8 State transitions under write-invalidation



Note: R = read fault occurs; *W* = write fault occurs.

Note that two or more processes with read-only copies may take write faults at more or less the same time. A read-only copy of a page may be out-of-date when ownership is eventually granted. To detect whether a current read-only copy of a page is out-of-date, each page can be associated with a sequence number, which is incremented whenever ownership is transferred. A process requiring write access encloses the sequence number of its read-only copy, if it possesses one. The current owner can then tell whether the page has been modified and therefore needs to be sent. This scheme is described by Kessler and Livny [1989] as the 'shrewd algorithm'.

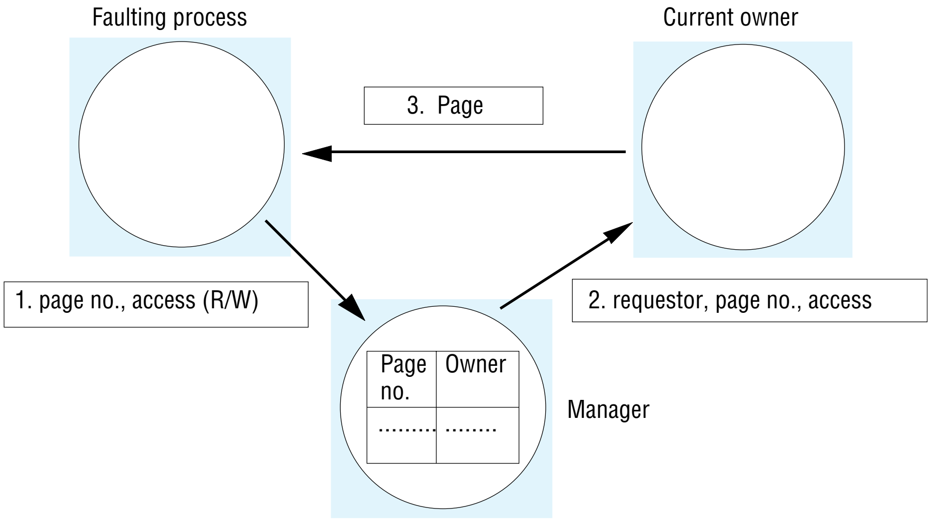
When a process P_R attempts to read a page *p* for which it has no access permissions, a read page fault takes place. The page-fault handling procedure is as follows:

- The page is copied from *owner*(*p*) to P_R.
- If the current owner is a single writer, then it remains as *p*'s owner and its access permission for *p* is set to read-only access. Retaining read access is desirable in case the process attempts to read the page subsequently – it will have retained an up-to-date version of the page. However, as the owner it will have to process subsequent requests for the page even if it does not access the page again. So it might turn out to have been more appropriate to reduce permission to no access and transfer ownership to P_R.
- $copyset(p) := copyset(p) \cup \{P_R\}$.
- The DSM runtime layer in P_R places the page with read-only permissions at the appropriate location in its address space and restarts the faulting instruction.

It is possible for a second page fault to occur during the transition algorithms just described. In order that transitions take place consistently, any new request for the page is not processed until after the current transition has completed.

The description just given has only explained *what* must be done. The problem of *how* to implement page fault handling efficiently is now addressed

Figure 18.9 Central manager and associated messages



18.3.3 Invalidation protocols

Two important problems remain to be addressed in a protocol to implement the invalidation scheme:

1. How to locate *owner*(*p*) for a given page *p*.
2. Where to store *copyset*(*p*).

For Ivy, Li and Hudak [1989] describe several architectures and protocols that take varying approaches to these problems. The simplest we shall describe is their improved centralized manager algorithm. In it, a single server called a manager is used to store the location (transport address) of *owner*(*p*) for every page *p*. The manager could be one of the processes running the application, or it could be any other process. In this algorithm, the set *copyset*(*p*) is stored at *owner*(*p*). That is, the identifiers and transport addresses of the members of *copyset*(*p*) are stored.

As shown in Figure 18.9, when a page fault occurs the local process (which we shall refer to as the *client*) sends a message to the manager containing the page number and the type of access required (read or read-write). The client awaits a reply. The manager handles the request by looking up the address of *owner*(*p*) and forwarding the request to the owner. In the case of a write fault, the manager sets the new owner to be the client. Subsequent requests are thus queued at the client until it has completed the transfer of ownership to itself.

The previous owner sends the page to the client. In the case of a write fault, it also sends the page's copy set. The client performs the invalidation when it receives the copy set. It sends a multicast request to the members of the copy set, awaiting acknowledgement from all the processes concerned that invalidation has taken place. The multicast need not be ordered. The former owner need not be included in the list of destinations, since it invalidates itself. The details of copy set management are left to the reader, who should consult the general invalidation algorithms given above.

The manager is a performance bottleneck and a critical point of failure. Li and Hudak suggested three alternatives that allow the load of page management to be divided between computers: fixed distributed page management, multicast-based distributed management and dynamic distributed management. In the first, multiple managers are used, each functionally equivalent to the central manager just described, but the pages are divided statically between them. For example, each manager could manage just those pages whose page numbers hash to a certain range of values. Clients calculate the hash number for the needed page and use a predetermined configuration table to look up the address of the corresponding manager.

This scheme would ameliorate the problem of load in general, but it has the disadvantage that a fixed mapping of pages to managers may not be suitable. When processes do not access the pages equally, some managers will incur more load than others. We now describe multicast-based and dynamic distributed management.

Using multicast to locate the owner ◊ Multicast can be used to eliminate the manager completely. When a process faults, it multicasts its page request to all the other processes. Only the process that owns the page replies. Care must be taken to ensure correct behaviour if two clients request the same page at more or less the same time: each client must obtain the page eventually, even if its request is multicast during transfer of ownership.

Consider two clients C_1 and C_2, which use multicast to locate a page owned by O. Suppose that O receives C_1's request first and transfers ownership to it. Before the page arrives, C_2's request arrives at O and at C_1. O will discard C_2's request because it no longer owns the page. Li and Hudak pointed out that C_1 should defer processing C_2's request until after it has obtained the page – otherwise it would discard the request because it is not the owner and C_2's request would be lost altogether. However, a problem still remains. C_1's request has been queued at C_2 meanwhile. After C_1 has eventually given C_2 the page, C_2 will receive and process C_1's request – which is now obsolete!

One solution is to use totally ordered multicast, so that clients can safely discard requests that arrive before their own (requests are delivered to themselves as well as to other processes). Another solution, which uses a cheaper unordered multicast but which consumes more bandwidth, is to associate each page with a vector timestamp, with one entry per process (see Chapter 11 for a description of vector timestamps). When page ownership is transferred, so is the timestamp. When a process obtains ownership, it increments its entry in the timestamp. When a process requests ownership, it encloses the last timestamp it held for the page. In our example, C_2 could discard C_1's request, because C_1's entry in the request's timestamp is lower than that which arrived with the page.

Whether an ordered multicast or unordered multicast is used, this scheme has the usual disadvantage of multicast schemes: processes that are not the owners of a page are interrupted by irrelevant messages, wasting processing time.

18.3.4 A dynamic distributed manager algorithm

Li and Hudak suggested the dynamic distributed manager algorithm, which allows page ownership to be transferred between processes but which uses an alternative to multicast as its method of locating a page's owner. The idea is to divide the overheads of locating pages between those computers that access them. Every process keeps, for every page *p*, a hint as to the page's current owner – the probable owner of *p*, or *probOwner*(*p*). Initially, every process is supplied with accurate page locations. In general, however, these values are *hints*, because pages can be transferred elsewhere at any time. As in previous algorithms, ownership is transferred only when a write fault occurs.

The owner of a page is located by following chains of hints that are set up as ownership of the page is transferred from computer to computer. The length of the chain – that is, the number of forwarding messages necessary to locate the owner – threatens to increase indefinitely. The algorithm overcomes this by updating the hints as more up-to-date values become available. Hints are updated and requests are forwarded as follows:

- When a process transfers ownership of page *p* to another process, it updates *probOwner*(*p*) to be the recipient.
- When a process handles an invalidation request for a page *p*, it updates *probOwner*(*p*) to be the requester.
- When a process that has requested read access to a page *p* receives it, it updates *probOwner*(*p*) to be the provider.
- When a process receives a request for a page *p* that it does not own, it forwards the request to *probOwner*(*p*) and resets *probOwner*(*p*) to be the requester.

The first three updates follow simply from the protocol for transferring page ownership and providing read-only copies. The rationale for the update when forwarding requests is that, for write requests, the requester will soon be the owner, even though it is not currently. In fact, in Li and Hudak's algorithm, assumed here, the *probOwner* update is made whether the request is for read access or write access. We return to this point shortly.

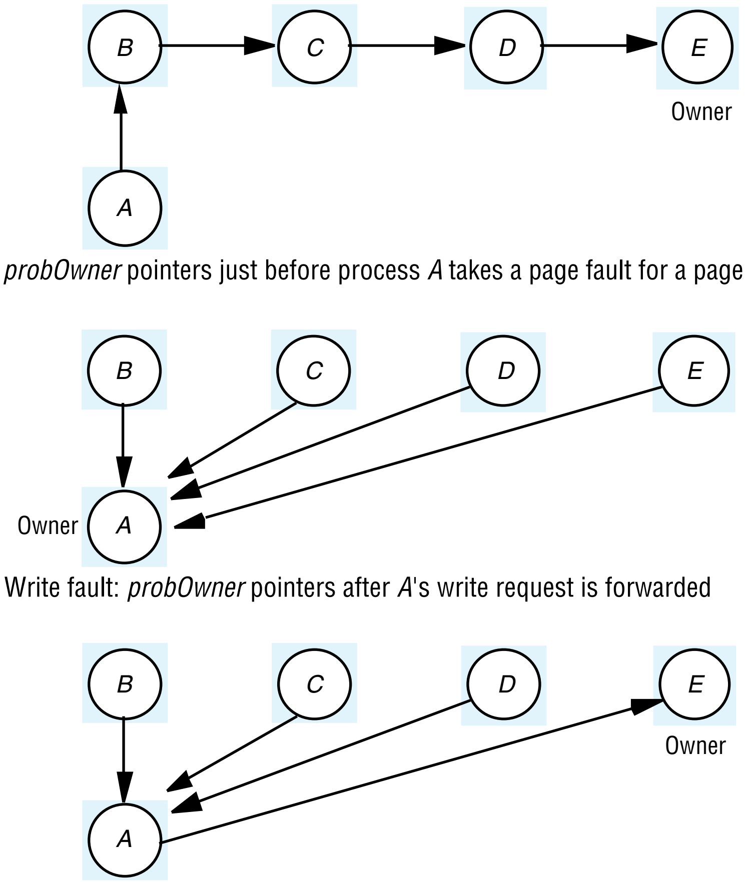
Figure 18.10 ((a) and (b)) illustrates *probOwner* pointers before and after process *A* takes a write page fault. *A*'s *probOwner* pointer for the page initially points to *B*. Processes *B*, *C* and *D* forward the request to *E* by following their own *probOwner* pointers; thereafter, all are set to point to *A* as a result of the update rules just described. The arrangement after fault handling is clearly better than that which preceded it: the chain of pointers has collapsed.

If, however, *A* takes a read fault, then process *B* is better off (two steps instead of three to *E*), *C*'s situation is the same as it was before (two steps), but *D* is worse off, with two steps instead of one (Figure 18.10(c)). Simulations are required to investigate the overall effect of this tactic on performance.

The average length of pointer chains can further be controlled by periodically broadcasting the current owner's location to all processes. This has the effect of collapsing all chains to length 1.

Li and Hudak describe the results of simulations that they carried out to investigate the efficacy of their pointer updates. With faulting processes chosen at random, for 1024 processors they found that the average number of messages taken to reach the owner of a page was 2.34 if broadcasts announcing the owner's location are made every 256 faults and 3.64 if broadcasts are made every 1024 faults. These figures are given only as illustrations: a complete set of results is given by Li and Hudak [1989].

Figure 18.10 Updating *probOwner* pointers



(a) *probOwner* pointers just before process *A* takes a page fault for a page owned by *E*

(b) Write fault: *probOwner* pointers after *A*'s write request is forwarded

(c) Read fault: *probOwner* pointers after *A*'s read request is forwarded

Note that a DSM system that uses a central manager requires two messages to reach the owner of a page.

Finally, Li and Hudak describe an optimization that potentially both makes invalidation more efficient and reduces the number of messages required to handle a read page fault. Instead of having to obtain a page copy from the owner of a page, a client can obtain a copy from any process with a valid copy. There is a chance that a client attempting to locate the owner will encounter such a process before the owner on the pointer chain.

This is done with the proviso that processes keep a record of clients that have obtained a copy of a page from them. The set of processes that possess read-only copies of a page thus forms a tree rooted at the owner, with each node pointing to the child nodes below, which obtained copies from it. The invalidation of a page begins at the owner and works down through the tree. On receiving an invalidation message, a node forwards it to its children in addition to invalidating its own copy. The overall effect is that some invalidations occur in parallel. This can reduce the overall time taken to invalidate a page – especially in an environment without hardware support for multicast.

18.3.5 Thrashing

It can be argued that it is the programmer's responsibility to avoid thrashing. The programmer could annotate data items in order to assist the DSM runtime in minimizing page copying and ownership transfers. The latter approach is discussed in the next section in the context of the Munin DSM system.

Mirage [Fleisch and Popek 1989] takes an approach to thrashing that is intended to be transparent to programmers. Mirage associates each page with a small time interval. Once a process has access to a page, it is allowed to retain access for the given interval, which serves as a type of timeslice. Other requests for the page are held off in the meantime. An obvious disadvantage of this scheme is that it is very difficult to choose the length of the timeslice. If the system uses a statically chosen length of time, it is liable to be inappropriate in many cases. A process might, for example, write a page only once and thereafter not access it; nonetheless, other processes are prevented from accessing it. Equally, the system might grant another process access to the page before it has finished using it.

A DSM system could choose the length of the timeslice dynamically. A possible basis for this is observation of accesses to the page (using the memory management unit's *referenced* bits). Another factor that could be taken into account is the length of the queue of processes waiting for the page.

18.4 Release consistency and Munin case study

The algorithms in the previous section were designed to achieve sequentially consistent DSM. The advantage of sequential consistency is that DSM behaves in the way that programmers expect shared memory to behave. Its disadvantage is that it is costly to implement. DSM systems often require the use of multicasts in their implementations, whether they are implemented using write-update or write-invalidation – although unordered multicast suffices for invalidation. Locating the owner of a page tends to be expensive: a central manager that knows the location of every page's owner acts as a bottleneck; following pointers involves more messages, on average. In addition, invalidation-based algorithms may give rise to thrashing.

Release consistency was introduced with the Dash multiprocessor, which implements DSM in hardware, primarily using a write-invalidation protocol [Lenoski *et al*. 1992]. Munin and Treadmarks [Keleher *et al.* 1992] have adopted a software implementation of it. Release consistency is weaker than sequential consistency and cheaper to implement, but it has reasonable semantics that are tractable to programmers.

The idea of release consistency is to reduce DSM overheads by exploiting the fact that programmers use synchronization objects such as semaphores, locks and barriers. A DSM implementation can use knowledge of accesses to these objects to allow memory to become inconsistent at certain points, while the use of synchronization objects nonetheless preserves application-level consistency.

18.4.1 Memory accesses

In order to understand release consistency – or any other memory model that takes synchronization into account – we begin by categorizing memory accesses according to their role, if any, in synchronization. Furthermore, we shall discuss how memory accesses may be performed asynchronously to gain performance and give a simple operational model of how memory accesses take effect.

As we said above, DSM implementations on general-purpose distributed systems may use message passing rather than shared variables to implement synchronization, for reasons of efficiency. But it may help to bear shared-variable-based synchronization in mind in the following discussion. The following pseudocode implements locks using the *testAndSet* operation on variables. The function *testAndSet* sets the lock to 1 and returns 0 if it finds it zero; otherwise it returns 1. It does this atomically.

```
acquireLock(var int lock):       // lock is passed by-reference
        while (testAndSet(lock) = 1)
            skip;
releaseLock(var int lock):       // lock is passed by-reference
        lock := 0;
```

Types of memory access ◊ The main distinction is between *competing* accesses and *non-competing* (*ordinary*) accesses. Two accesses are competing if:

- they may occur concurrently (there is no enforced ordering between them) and
- at least one is a *write*.

So two *read* operations can never be competing; a *read* and a *write* to the same location made by two processes that synchronize between the operations (and so order them) are non-competing.

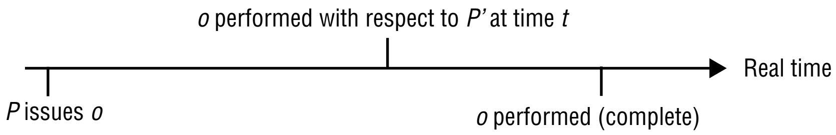
We further divide competing accesses into *synchronization* and *non-synchronization* accesses:

- synchronization accesses are *read* or *write* operations that contribute to synchronization;
- non-synchronization accesses are *read* or *write* operations that are concurrent but that do not contribute to synchronization.

The *write* operation implied by '*lock* := 0' in *releaseLock* (above) is a synchronization access. So is the *read* operation implicit in *testAndSet.*

Synchronization accesses are competing, because potentially synchronizing processes must be able to access synchronization variables concurrently and they must update them: *read* operations alone could not achieve synchronization. But not all competing accesses are synchronization accesses – there are classes of parallel algorithms in which processes make competing accesses to shared variables just to update and read one another's results and not to synchronize.

Figure 18.11 Timeline for performing a DSM *read* or *write* operation



Synchronization accesses are further divided into *acquire* accesses and *release* accesses, corresponding to their role in potentially blocking the process making the access, or in unblocking some other process.

Performing asynchronous operations ◊ As we saw when discussing implementations of sequentially consistent DSM, memory operations may incur significant delays. Several forms of asynchronous operation are available to increase the rate at which processes execute, despite these delays. First, *write* operations may be implemented asynchronously. A written value is buffered before being propagated and the effects of the *write* are observed later by other processes. Second, DSM implementations may pre-fetch values in anticipation of reading them, to avoid stalling a process at the time it needs the values. Third, processors may perform instructions out of order. While awaiting completion of the current memory access, they may issue the next instruction, as long as it does not depend on the current instruction.

In view of the asynchronous operation that we have outlined, we distinguish between the point at which a *read* or *write* operation is issued – when the process first commences execution of the operation – and the point when the instruction is *performed* or completed.

We shall assume that our DSM is at least coherent. As Section 18.2.3 explained, this means that every process agrees on the order of *write* operations to the same location. Given this assumption, we may speak unambiguously of the ordering of *write* operations to a given location.

In a distributed shared memory system, we can draw a timeline for any memory operation *o* that process *P* executes (see Figure 18.11).

We say that a *write* operation $W(x)v$ has *performed* with respect to a process *P* if from that point *P*'s *read* operations will return *v* as written by that *write* operation, or the value of some subsequent write to *x* (note that another operation may write the same value *v*).

Similarly, we say that a *read* operation $R(x)v$ has performed with respect to process *P* when no subsequent *write* issued to the same location could possibly supply the value *v* that *P* reads. For example, *P* may have pre-fetched the value it needs to *read*.

Finally, the operation *o* has *performed* if it has performed with respect to all processes.

18.4.2 Release consistency

The requirements that we wish to meet are:

- to preserve the synchronization semantics of objects such as locks and barriers;

- to gain performance, we allow a degree of asynchronicity for memory operations;
- to constrain the overlap between memory accesses in order to guarantee executions that provide the equivalent of sequential consistency.

Release-consistent memory is designed to satisfy these requirements. Gharachorloo *et al.* 1990 define release consistency as follows:

RC1: before an ordinary *read* or *write* operation is allowed to perform with respect to any other process, all previous *acquire* accesses must be performed.

RC2: before a *release* operation is allowed to perform with respect to any other process, all previous ordinary *read* and *write* operations must be performed.

RC3: *acquire* and *release* operations are sequentially consistent with respect to one another.

RC1 and RC2 guarantee that, when a release has taken place, no other process acquiring a lock can read stale versions of data modified by the process that performs the release. This is consistent with the programmer's expectation that a release of a lock, for example, signifies that a process has finished modifying data within a critical section.

The DSM runtime can only enforce the release consistency guarantee if it is aware of synchronization accesses. In Munin, for example, the programmer is forced to use Munin's own *acquireLock*, *releaseLock* and *waitAtBarrier* primitives. (A barrier is a synchronization object that blocks each of a set of processes until all have waited on it; all processes then continue.) A program must use synchronization to ensure that updates are made visible to other processes. Two processes that share DSM but never use synchronization objects may never see one another's updates if the implementation strictly applies the sole guarantee given above.

Note that the release consistency model does allow an implementation to employ some asynchronous operations. For example, a process need not be blocked when it makes updates within a critical section. Nor do its updates have to be propagated until it leaves the critical section by releasing a lock. Furthermore, updates can then be collected and sent in a single message. Only the final update to each data item need be sent.

Consider the processes in Figure 18.12, which acquire and release a lock in order to access a pair of variables *a* and *b* (*a* and *b* are initialized to zero). Process 1 updates *a* and *b* under conditions of mutual exclusion, so that process 2 cannot read *a* and *b* at the same time and so will find $a = b = 0$ or $a = b = 1$. The critical sections enforce consistency – equality of *a* and *b* – *at the application level*. It is redundant to propagate updates to the variables affected during the critical section. If process 2 tried to access *a*, say, outside a critical section, then it might find a stale value. That is a matter for the application writer.

Let us assume that process 1 acquires the lock first. Process 2 will block and not cause any activity related to DSM until it has acquired the lock and attempts to access *a* and *b*. If the two processes were to execute on a sequentially consistent memory, then process 1 would block when it updates *a* and *b*. Under a write-update protocol, it would block while all versions of the data are updated; under a write-invalidation protocol, it would block while all copies are invalidated.

Figure 18.12 Processes executing on a release-consistent DSM

```
Process 1:
        acquireLock();           // enter critical section
        a := a + 1;
        b := b + 1;
        releaseLock();           // leave critical section
Process 2:
        acquireLock();           // enter critical section
        print ("The values of a and b are: ", a, b);
        releaseLock();           // leave critical section
```

Under release consistency, process 1 will not block when it accesses *a* and *b*. The DSM runtime system notes which data have been updated but need take no further action at that time. It is only when process 1 has released the lock that communication is required. Under a write-update protocol, the updates to *a* and *b* will be propagated; under a write-invalidation protocol, the invalidations should be sent.

The programmer (or a compiler) is responsible for labelling read and write operations as *release*, *acquire* or *non-synchronization* accesses – other instructions are assumed to be ordinary. To label the program is to direct the DSM system to enforce the release consistency conditions.

Gharachorloo *et al*. [1990] describe the concept of a *properly labelled* program. They prove that such a program cannot distinguish between a release-consistent DSM and a sequentially consistent DSM.

18.4.3 Munin

The Munin DSM design [Carter *et al*. 1991] attempts to improve the efficiency of DSM by implementing the release consistency model. Furthermore, Munin allows programmers to annotate their data items according to the way in which they are shared, so that optimizations can be made in the update options selected for maintaining consistency. It is implemented upon the V kernel [Cheriton and Zwaenepoel 1985], which was one of the first kernels to allow user-level threads to handle page faults and manipulate page tables.

The following points apply to Munin's implementation of release consistency:

- Munin sends update or invalidation information as soon as a lock is released.
- The programmer can make annotations that associate a lock with particular data items. In this case, the DSM runtime can propagate relevant updates in the same message that transfers the lock to a waiting process – ensuring that the lock's recipient has copies of the data it needs before it accesses them.

Keleher *et al*. [1992] describe an alternative to Munin's *eager* approach of sending update or invalidation information at the time of a release. Instead, this *lazy* implementation does so only when the lock in question is next acquired. Furthermore, it sends this information only to the process acquiring the lock and piggy backs it onto the message granting the lock. It is unnecessary to make the updates visible to other processes until they in turn acquire the lock.

Sharing annotations ◊ Munin implements a variety of consistency protocols, which are applied at the granularity of individual data items. The protocols are parameterized according to the following options:

- whether to use a write-update or write-invalidate protocol;
- whether several replicas of a modifiable data item may exist simultaneously;
- whether or not to delay updates or invalidations (for example, under release consistency);
- whether the item has a fixed owner, to which all updates must be sent;
- whether the same data item may be modified concurrently by several writers;
- whether the data item is shared by a fixed set of processes;
- whether the data item may be modified.

These options are chosen according to the nature of the data item and the pattern of its sharing between processes. The programmer can make an explicit choice of which parameter options to use for each data item. However, Munin supplies a small, standard set of annotations for the programmer to apply to data items, each of which implies a convenient choice of the parameters, suitable for a variety of applications and data items. These are as follows:

Read-only: No updates may be made after initialization and the item may be freely copied.

Migratory: Processes typically take turns in making several accesses to the item, at least one of which is an update. For example, the item might be accessed within a critical section. Munin always gives both read and write access together to such an object, even when a process takes a read fault. This saves subsequent write-fault processing.

Write-shared: Several processes update the same data item (for example, an array) concurrently, but this annotation is a declaration from the programmer that the processes do not update the same parts of it. This means that Munin can avoid false sharing but must propagate only those words in the data item that are actually updated at each process. To do this, Munin makes a copy of a page (inside a write-fault handler) just before it is updated locally. Only the differences between the two versions are sent in an update.

Producer-consumer: The data object is shared by a fixed set of processes, only one of which updates it. As we explained when discussing thrashing above, a write-update protocol is most suitable here. Moreover, updates may be delayed under the model of release consistency, assuming that the processes use locks to synchronize their accesses.

Reduction: The data item is always modified by being locked, read, updated and unlocked. An example of this is a global minimum in a parallel computation, which must be fetched and modified atomically if it is greater than the local minimum. These items are stored at a fixed owner. Updates are sent to the owner, which propagates them.

Result: Several processes update different words within the data item; a single process reads the whole item. For example, different 'worker' processes might fill in different elements of an array, which is then processed by a 'master' process. The point here is that the updates need only be propagated to the master and not to the workers (as would occur under the 'write-shared' annotation just described).

Conventional: The data item is managed under an invalidation protocol similar to that described in the previous section. No process may therefore read a stale version of the data item.

Carter *et al*. [1991] detail the parameter options used for each of the annotations we have given. This set of annotations is not fixed. Others may be created as sharing patterns that require different parameter options are encountered.

18.5 Other consistency models

Models of memory consistency can be divided into *uniform models*, which do not distinguish between types of memory access, and *hybrid models*, which do distinguish between ordinary and synchronization accesses (as well as other types of access).

Several uniform models exist that are weaker than sequential consistency. We introduced coherence in Section 18.2.3, in which the memory is sequentially consistent on a location-by-location basis. Processors agree on the order of all writes to a given location, but they may differ on the order of writes from different processors to different locations [Goodman 1989, Gharachorloo *et al*. 1990].

Other uniform consistency models include:

Causal consistency: Reads and writes may be related by the happened-before relationship (see Chapter 11). This is defined to hold between memory operations when either (a) they are made by the same process; (b) a process reads a value written by another process; or (c) there exists a sequence of such operations linking the two operations. The model's constraint is that the value returned by a read must be consistent with the happened-before relationship. This is described by Hutto and Ahamad [1990].

Processor consistency: The memory is both coherent and adheres to the pipelined RAM model (see below). The simplest way to think of processor consistency is that the memory is coherent and that all processes agree on the ordering of any two write accesses made by the same process – that is, they agree with its program order. This was first defined informally by Goodman [1989] and later formally defined by Gharachorloo *et al.* [1990] and Ahamad *et al*. [1992].

Pipelined RAM: All processors agree on the order of writes issued by any given processor [Lipton and Sandberg 1988].

In addition to release consistency, hybrid models include:

> *Entry consistency*: Entry consistency was proposed for the Midway DSM system [Bershad *et al.* 1993]. In this model, every shared variable is bound to a synchronization object such as a lock, which governs access to that variable. Any process that first acquires the lock is guaranteed to read the latest value of the variable. A process wishing to write the variable must first obtain the corresponding lock in 'exclusive' mode – making it the only process able to access the variable. Several processes may read the variable concurrently by holding the lock in non-exclusive mode. Midway avoids the tendency to false sharing in release consistency, but at the expense of increased programming complexity.
>
> *Scope consistency*: This memory model [Iftode *et al.* 1996] attempts to simplify the programming model of entry consistency. In scope consistency, variables are associated with synchronization objects largely automatically instead of relying on the programmer to associate locks with variables explicitly. For example, the system can monitor which variables are updated in a critical section.
>
> *Weak consistency*: Weak consistency [Dubois *et al.* 1988] does not distinguish between *acquire* and *release* synchronization accesses. One of its guarantees is that all previous ordinary accesses complete before *either* type of synchronization access completes.

Discussion ◊ Release consistency and some of the other consistency models weaker than sequential consistency appear to be the most promising for DSM. It does not seem to be a significant disadvantage of the release consistency model that synchronization operations need to be known to the DSM runtime – as long as those supplied by the system are sufficiently powerful to meet the needs of programmers.

It is important to realize that, under the hybrid models, most programmers are not forced to consider the particular memory consistency semantics used as long as they synchronize their data accesses appropriately. But there is a general danger in DSM designs of asking the programmer to perform many annotations to his or her program in order to make its execution efficient. This includes both annotations identifying data items with synchronization objects and the sharing annotations such as those of Munin. One of the advantages of shared-memory programming over message passing is supposed to be its relative convenience.

18.6 Summary

This chapter has described and motivated the concept of distributed shared memory as an abstraction of shared memory that is an alternative to message-based communication in a distributed system. DSM is primarily intended for parallel processing and data sharing. It has been shown to perform as well as message passing for certain parallel applications, but it is difficult to implement efficiently, and its performance varies with applications.

The chapter has concentrated on software implementations of DSM – particularly those using the virtual memory subsystem – but it has been implemented with hardware support.

The main design and implementation issues are the DSM structure, the means by which applications synchronize, the memory consistency model, the use of write-update or write-invalidation protocols, the granularity of sharing, and thrashing.

The DSM is structured either as a series of bytes, a collection of shared objects, or a collection of immutable data items such as tuples.

Applications using DSM require synchronization in order to meet application-specific consistency constraints. They use objects such as locks for this purpose, implemented using message passing for efficiency.

The most common strict type of memory consistency implemented in DSM systems is sequential consistency. Because of its cost, weaker consistency models have been developed, such as coherence and release consistency. Release consistency enables the implementation to exploit the use of synchronization objects to achieve greater efficiency without breaking application-level consistency constraints. Several other consistency models were outlined, including entry, scope and weak consistency, which all exploit synchronization.

Write-update protocols are those in which updates are propagated to all copies as data items are updated. These are usually implemented in hardware, although software implementations using totally ordered multicast exist. Write-invalidation protocols prevent stale data being read by invalidating copies as data items are updated. These are more suited to page-based DSM, for which write-update may be an expensive option.

The granularity of DSM affects the likelihood of contention between processes that falsely share data items because they are contained in the same unit of sharing (for example, page). It also affects the cost per byte of transferring updates between computers.

Thrashing may occur when write-invalidation is used. This is the repeated transfer of data between competing processes at the expense of application progress. This may be reduced by application-level synchronization, by allowing computers to retain a page for a minimum time, or by labelling data items so that both read and write access are always granted together.

The chapter has described Ivy's three main write-invalidate protocols for page-based DSM, which address the problems of managing the copy set and locating the owner of a page. These were the central manager protocol, in which a single process stores the current owner's address for each page; the protocol that uses multicast to locate the current owner of a page; and the dynamic distributed manager protocol, which uses forwarding pointers to locate the current owner of a page.

Munin is an example implementation of release consistency. It implements eager release consistency in that it propagates update or invalidation messages as soon as a lock is released. Alternative, lazy implementations exist, which propagate those messages only when they are required. Munin allows programmers to annotate their data items in order to select the protocol options that are best suited to them, given the way in which they are shared.

EXERCISES

18.1 Explain in which respects DSM is suitable or unsuitable for client-server systems. *page 750*

18.2 Discuss whether message passing or DSM is preferable for fault-tolerant applications. *page 751*

18.3 How would you deal with the problem of differing data representations for a middleware-based implementation of DSM on heterogeneous computers? How would you tackle the problem in a page-based implementation? Does your solution extend to pointers? *page 753*

18.4 Why should we want to implement page-based DSM largely at user-level, and what is required to achieve this? *page 754*

18.5 How would you implement a semaphore using a tuple space? *page 755*

18.6 Is the memory underlying the following execution of two processes sequentially consistent (assuming that, initially, all variables are set to zero)?

P_1: $R(x)1; R(x)2; W(y)1$

P_2: $W(x)1; R(y)1; W(x)2$ *page 759*

18.7 Using the *R*(), *W*() notation, give an example of an execution on a memory that is coherent but not sequentially consistent. Can a memory be sequentially consistent but not coherent? *page 759*

18.8 In write-update, show that sequential consistency could be broken if each update were to be made locally before asynchronously multicasting it to other replica managers, even though the multicast is totally ordered. Discuss whether an asynchronous multicast can be used to achieve sequential consistency. (Hint: consider whether to block subsequent operations.) *page 760*

18.9 Sequentially consistent memory can be implemented using a write-update protocol employing a synchronous, totally ordered multicast. Discuss what multicast ordering requirements would be necessary to implement coherent memory. *page 760*

18.10 Explain why, under a write-update protocol, care is needed to propagate only those words within a data item that have been updated locally.

Devise an algorithm for representing the differences between a page and an updated version of it. Discuss the performance of this algorithm. *page 760*

18.11 Explain why granularity is an important issue in DSM systems. Compare the issue of granularity between object-oriented and byte-oriented DSM systems, bearing in mind their implementations.

Why is granularity relevant to tuple spaces, which contain immutable data?

What is false sharing? Can it lead to incorrect executions? *page 762*

18.12 What are the implications of DSM for page replacement policies (that is, the choice of which page to purge from main memory in order to bring a new page in)? *page 763*

18.13 Prove that Ivy's write-invalidate protocol guarantees sequential consistency. *page 765*

18.14 In Ivy's dynamic distributed manager algorithm, what steps are taken to minimize the number of lookups necessary to find a page? *page 768*

18.15 Why is thrashing an important issue in DSM systems and what methods are available for dealing with it? *page 771*

18.16 Discuss how condition RC2 for release consistency could be relaxed. Hence distinguish between eager and lazy release consistency. *page 774*

18.17 A sensor process writes the current temperature into a variable *t* stored in a release-consistent DSM. Periodically, a monitor process reads *t*. Explain the need for synchronization to propagate the updates to *t*, even though none is otherwise needed at the application level. Which of these processes needs to perform synchronization operations? *page 773*

18.18 Show that the following history is not causally consistent:

P_1: $W(a)0;\ W(a)1$

P_2: $R(a)1;\ W(b)2$

P_3: $R(b)2;\ R(a)0$ *page 777*

18.19 What advantage can a DSM implementation obtain from knowing the association between data items and synchronization objects? What is the disadvantage of making the association explicit? *page 778*