



Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001

Chapter 16 Exercise Solutions

16.1 Explain in which respects DSM is suitable or unsuitable for client-server systems.

16.1 Ans.

DSM is unsuitable for client-server systems in that it is not conducive to heterogeneous working. Furthermore, for security we would need a shared region per client, which would be expensive.

DSM may be suitable for client-server systems in some application domains, e.g. where a set of clients share server responses.

16.2 Discuss whether message passing or DSM is preferable for fault-tolerant applications.

16.2 Ans.

Consider two processes executing at failure-independent computers. In a message passing system, if one process has a bug that leads it to send spurious messages, the other may protect itself to a certain extent by validating the messages it receives. If a process fails part-way through a multi-message operation, then transactional techniques can be used to ensure that data are left in a consistent state.

Now consider that the processes share memory, whether it is physically shared memory or page-based DSM. Then one of them may adversely affect the other if it fails, because now one process may update a shared variable without the knowledge of the other. For example, it could incorrectly update shared variables due to a bug. It could fail after starting but not completing an update to several variables.

If processes use middleware-based DSM, then it may have some protection against aberrant processes. For example, processes using the Linda programming primitives must explicitly request items (tuples) from the shared memory. They can validate these, just as a process may validate messages.

16.3 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?

16.3 Ans.

The middleware calls can include marshalling and unmarshalling procedures. In a page-based implementation, pages would have to be marshalled and unmarshalled by the kernels that send and receive them. This implies maintaining a description of the layout and types of the data, in the DSM segment, which can be converted to and from the local representation.

A machine that takes a page fault needs to describe which page it needs in a way that is independent of the machine architecture. Different page sizes will create problems here, as will data items that straddle page boundaries, or items that straddle page boundaries when unmarshalled.

A solution would be to use a 'virtual page' as the unit of transfer, whose size is the maximum of the page sizes of all the architectures supported. Data items would be laid out so that the same set of items occurs in each virtual page for all architectures. Pointers can also be marshalled, as long as the kernels know the layout of data, and can express pointers as pointing to an object with a description of the form "Offset o in data item i ", where o and i are expressed symbolically, rather than physically.

This activity implies huge overheads.

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

16.4 Ans.

A user-level DSM implementation facilitates application-specific memory (consistency) models and protocol options.

We require the kernel to export interfaces for (a) handling page faults from user level (in UNIX, as a signal) and (b) setting page protections from user level (see the UNIX *mmap* system call).

16.5 How would you implement a semaphore using a tuple space?

16.5 Ans.

We implement a semaphore with standard *wait* and *signal* operations; the style of semaphore that counts the number of blocked processes if its count is negative;

The implementation uses a tuple $\langle \text{"count"}, \text{int} \rangle$ to maintain the semaphore's integer value; and tuples $\langle \text{"blocked"}, \text{int} \rangle$ and $\langle \text{"unblocked"}, \text{int} \rangle$ for each process that is blocked on the semaphore.

The *wait* operation is implemented as follows:

```

<"count", count> = ts.take(<"count", int>);
if (count > 0)
    count := count - 1;
else
{
    // Use ts.read(<"blocked", int>) to find the smallest b such that <"blocked", b> is not in ts
    ts.write(<"blocked", b>);
}
ts.write(<"count", count>);
ts.take(<"unblocked", b>; // blocks until a corresponding tuple enters ts

```

The *signal* operation is implemented as follows:

```

<"count", count> = ts.take(<"count", int>);
if (there exists any b such that <"blocked", b> is in ts)
{
    ts.take(<"blocked", b>);
    ts.write("unblocked", b); // unblocks a process
}
else
    count := count + 1;
ts.write("count", count);

```

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

```

P1:      R(x)1; R(x)2; W(y)1
P2:      W(x)1; R(y)1; W(x)2

```

16.6 Ans.

P1 reads the value of x to be 2 before setting y to be 1. But P2 sets x to be 2 only after it has read y to be 1 (y was previously zero). Therefore these two executions are incompatible, and the memory is not sequentially consistent.

16.7 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?

16.7 Ans.

The execution of Exercise 16.6 is coherent – the reads and writes for each variable are consistent with both program orders – but it is not sequentially consistent, as we showed.

A sequentially consistent memory is consistent with program order; therefore the sub-sequences of operations on each individual variable are consistent with program order – the memory is coherent.

16.8 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.)

16.8 Ans.

```
P1:   W(x)1;           // x is updated immediately at P1, and multicast elsewhere
      R(x)1;
      R(y)0;           // the multicast from P2 has not arrived yet

P2:   W(y)1;
      R(x)0;           // the multicast from P1 has not arrived yet
      R(y)1;           // y was updated immediately and multicast to P1
```

P1 would say that *x* was updated before *y*; *P2* would say that *y* was updated before *x*. So the memory is not sequentially consistent when we use an asynchronous totally ordered multicast, and if we update the local value immediately.

If the multicast was synchronous (that is, a writer is blocked until the update has been delivered everywhere) and totally ordered, then it is easy to see that all processes would agree on a serialization of their updates (and therefore of all memory operations).

We could allow the totally-ordered multicast to be asynchronous, if we block any subsequent read operation until all outstanding updates made by the process have been assigned their total order (that is, the corresponding multicast messages have been delivered locally). This would allow writes to be pipelined, up to the next read.

- 16.9 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.

16.9 Ans.

One could implement a coherent memory by using a multicast that totally ordered writes to each individual location, but which did not order writes to different locations. For example, one could use different sequencers for different location. Updates for a location are sequenced (totally ordered) by the corresponding sequencer; but updates to different locations could arrive in different orders at different locations.

- 16.10 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.

16.10 Ans.

Assume that two processes update different words within a shared page, and that the whole page is sent when delivering the update to other processes (that is, they falsely share the page). There is a danger that the unmodified words in one process's updated page will overwrite another's modifications to the falsely shared words.

To get around this problem, each process sends only the differences it has made to the page (see the discussion of Munin's write-shared data items on page 540). That way, updates to falsely shared words will be applied only to the affected words, and will not conflict. To implement this, it is necessary for each process to keep a copy of the page before it updates it.

A simple encoding of the differences between the page before and after it was modified is to create a series of tuples:

<pageOffset, size, changedBytes>

– which store in *changedBytes* a run of *size* bytes to change, starting at *pageOffset*. The list of tuples is created by comparing the pages byte-for-byte. Starting at the beginning, when a difference is encountered, we create a new tuple and record the byte's offset. We then copy the bytes from the modified page into *changedBytes*, until we reach a run of bytes that are the same in both pages and whose length is greater than a certain minimum value *M*. The encoding procedure then continues until the whole page has been encoded.

In judging such an algorithm, we are mindful of the processing time and the storage space taken up by the encoded differences. The minimum length *M* is chosen so that the processing time and storage space taken up by creating a new tuple is justified against the overhead of copying bytes that have not been modified. It is

likely, for example, that it is cheaper to store and copy a run of four unmodified bytes than to create an extra tuple.

The reader may care to improve further upon this algorithm.

- 16.11 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?

16.11 Ans.

We refer the reader to the discussion of granularity on pages 648-649.

In a page-based implementation, the minimum granularity is fixed by the hardware. In Exercise 16.10 we have seen a way of updating smaller quantities of data than a page, but the expense of this technique makes it not generally applicable.

In middleware-based DSM, the granularity is up to the implementation, and may be as small as one byte. If a process updates one field in a data structure, then the implementation may choose to send just the field or the whole data structure in its update.

Consider a DSM such as Linda's Tuple Space, which consists of immutable data items. Suppose a process needs to update one element in a tuple containing a million-element array. Since tuples are immutable the process must extract the tuple, copying the whole array into its local variables; then it modifies the element; then it writes the new tuple back into Tuple Space. Far more data has been transferred than was needed for the modification. If the data had been stored as separate tuples, each containing one array element, then the update would have been much cheaper. On the other hand, a process wishing to access the whole array would have to make a million accesses to tuple space. Because of latency, this would be far more expensive than accessing the whole array in one tuple.

False sharing is described on pages 648-649. It does not of itself lead to incorrect executions, but it lowers the efficiency of a DSM system.

- 16.12 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)?

16.12 Ans.

When a kernel wishes to replace a page belonging to a DSM segment, it can choose between pages that are read-only, pages that are read-only but which the kernel owns, and pages that the kernel has write access to (and has modified). Of these options, the least cost is associated with deleting the unowned read-only page (which the kernel can always obtain again if necessary); if the kernel deletes a read-only page that it owns, then it has lost a potential advantage if write access is soon required; and if it deletes the modified page then it must first transfer it elsewhere over the network or onto a local disk. So the kernel would prefer to delete pages in the order given. Of course it can discriminate between pages with equal status by choosing, for example, the least recently accessed.

- 16.13 Prove that Ivy's write-invalidate protocol guarantees sequential consistency.

16.13 Ans.

A memory is not sequentially consistent if (and only if) there is an execution in which two processes disagree about the order in which two or more updates were made. In a write-invalidate protocol, updates to any particular page are self-evidently serialised (only one process may update it at a time, and readers are meanwhile excluded). So we may assume that the updates are to different variables, residing in different pages.

Suppose the variables are x and y , with initial values (without loss of generality) 6 and 7. Let us suppose, further that x is incremented to 16, and y is incremented to 17. We suppose again, without loss of generality, that two processes' histories contain the following evidence of disorder:

$P1: \quad R/W(x)16; \dots; R(y)7; \dots // x$ was incremented first

$P2: \quad R/W(y)17; \dots; R(x)6; \dots // y$ was incremented first

Since the write-invalidate protocol was used, $P1$ obtained access to x 's page, where it either read x or wrote x as 16. Subsequently, $P1$ obtained access to y 's page, where it read y to be 7. For $P2$ to have read or written y with the value 17, it must have obtained access to y 's page after $P1$ finished with it. Later still, it obtained

access to x 's page and found that x had the value 6. By *reductio ad absurdum*, we may assume that our hypothesis was false: no such executions can exist, and the memory is sequentially consistent.

Lamport [1979] gives a more general argument, that a memory is sequentially consistent if the following two conditions apply:

R1: Each processor (process) issues memory requests in the order specified by its program.

R2: Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of entering the request on this queue.

Write-invalidation satisfies these conditions, where we substitute 'page' for 'memory module'.

The reader is invited to generalise the argument we have given, to obtain Lamport's result. You will need to construct a partial order between memory requests, based upon the order of issue by a single process, and the order of request servicing at the pages.

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

16.14 Ans.

We refer the reader to the discussion on page 655.

16.15 Why is thrashing an important issue in DSM systems and what methods are available for dealing with it?

16.15 Ans.

Thrashing and the Mirage approach to it are discussed on page 657. The discussion on Munin (pages 661-663) describes how sharing annotations may be used to aid the DSM run-time in preventing thrashing. For example, migratory data items are always given read-and-write access, even if the first access is a read access. This is done in the expectation that a read is typically followed closely by a write. The producer-consumer annotation causes the run-time to use a write-update protocol, instead of an invalidation protocol. This is more appropriate, in that it avoids continually transferring access to the data item between the writer (producer) and readers (consumers).

16.16 Discuss how condition RC2 for release consistency could be relaxed. Hence distinguish between eager and lazy release consistency.

16.16 Ans.

Consider a process P that updates variables within a critical section. It may not be strictly necessary for another process to observe P 's updates until it enters the critical section, whereas RC2 stipulates that the updates should occur on P 's *release* operation. (Neither of these semantics is absolutely 'right' or 'wrong'; but the programmer has to be made aware of which is used.)

Implementations of eager release consistency propagate updates or invalidations upon the *release* operation; lazy ones propagate them when another process enters the critical section (issues an *acquire* operation).

16.17 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?

16.17 Ans.

A release-consistent DSM implementation is free never to propagate an update in the absence of synchronisation. To guarantee that the monitor process sees new values of the variable t , synchronisation operations must be used. Using the definition of release consistency on p. 660, only the sensor process needs to issue *acquire* and *release* operations.

16.18 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 : $R(b)2$; $R(a)0$

16.18 Ans.

In the following, we use subscripts to denote which process issued an operation.

$W_2(b)2$ writes-into $R_3(b)2$

Taking this together with P_3 's execution order implies the following order:

$W_2(b)2; R_3(b)2; R_3(a)0; W_1(a)1;$

But $W_1(a)1$ is causally before $W_2(b)2$ – so no causally consistent serialisation exists.

- 16.19 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?

16.19 Ans.

The DSM implementation can use the association to determine which variables' updates/invalidations need to be propagated with a lock (there may be multiple critical sections, used for different sets of variables).

The disadvantage of making the association explicit is the work that this represents to the programmer (who, moreover, may make inaccurate associations).