CHAPTER 9

Sharing of Data and Code in Main Memory

The engineering of large software systems is greatly simplified if individual modules can be constructed separately and later linked together. This also permits the reuse of modules developed previously, and the use of libraries and other modules built by others, thus reducing the total software development cost. To facilitate the necessary cooperation, a flexible linking mechanism should be provided that does not require individual modules to be recompiled each time they are being included in a user address space. Important additional benefits are obtained if the same copy of a module can be linked into more than one address space at the same time, essentially *sharing* a single copy of the module in main memory among different processes. In addition to sharing code, many concurrent programs cooperate and must share common data areas to exchange partial results or other forms of information with each other.

The first part of the chapter presents the main techniques for the sharing of code and data in centralized architectures. Because the shared memory model has proven to be such a convenient abstraction, it has also been implemented on top of distributed systems. In the last sections of the chapter, we introduce the basic ideas underlying distributed shared memory.

9.1 SINGLE-COPY SHARING

Our general focus is on sharing a **single copy** of code or data in memory. We are not concerned with the sharing of software components for the purposes of reuse. The latter form of sharing can be accomplished easily by giving each process its own private copy of the shared object. The copy can be incorporated, either manually or by the linker, into other programs.

9.1.1 Reasons for Sharing

There are two main reasons for single-copy sharing. The first, which involves the sharing of **data**, is that some processes are designed to communicate, cooperate, and compete

for resources with each other through shared memory. This is a reason similar to that underlying the need for critical sections. For example, producer and consumer processes typically share a common buffer area through which data flows from one to the other. Other applications include those that select work from a common pool of tasks and those that access a common data structure representing the current state of the computation. For example, multiple processes could be engaged in a search or a divide-and-conquer computation, where a shared tree structure records the global state. Various systems resources also are accessed in a shared manner by multiple concurrent processes. File directories are notable examples of such common resources. To prevent inconsistencies due to concurrent updates, only a single copy must be in memory at any given time. A similar need for sharing arises with various system status databases, for example, those recording which resources are free and which are busy. Such information is continually being searched and updated by different OS processes, and must be maintained as a single consistent copy.

The second main reason for single-copy sharing is to better utilize main memory. This involves both **code** and **data**. It is frequently the case that several active processes use the same code or data. For example, many users in a time-sharing system may be running the same editor or debugger. If each process had its own copy, the need for main memory would increase dramatically. This, in turn, would reduce the number of users able to share the system at the same time. It also would increase the I/O overhead in loading the excess copies into memory; and in systems with demand paging, it would increase the page fault rate and thus the risk of thrashing. For these reasons, using a single copy of any heavily used software component is highly desirable. Such components generally include OS kernel routines, for example, the I/O drivers, which must be shared by all processes; and various utilities and system services, such as debuggers, file manipulation routines, and linkers and loaders. Finally, popular text editors and language processors, i.e., compilers, assemblers, and interpreters, are often maintained as a single shared copy in main memory.

Sharing of *user* level code and data also may be beneficial. An example is a shared-memory multiprocessing application where several processes execute the same code, such as the body of a parallel loop, with different data sets; here, a single copy of the common code could be shared in memory by the different processes.

9.1.2 Requirements for Sharing

Designating Resources as Shared

To allow some portion of code or data to be shared among processes, the system must provide some way of **expressing** what is to be shared and under what conditions or constraints. This can be done *statically* at the time of linking or loading, or *dynamically*, at runtime.

In the simplest case, all resources to be shared are known a priori. This is usually the approach taken when **systems** resources are shared, such as parts of the OS kernel, compilers, editors, and other widely used software components. These are designated as single-copy shared implicitly, i.e., by convention, at the time of systems design or initialization. Since all sharable resources are known, the OS can keep track of them and only load a copy if this does not already exist in memory. In some cases, a resource can be assigned permanently to a specific portion of memory. This is usually done with heavily used parts of the OS kernel, such interrupt handlers or I/O drivers. Other, less frequently

"runall"
2002/9/24
page 276

used resources, such as compilers or editors, can be loaded on demand whenever one or more process need to use them.

In systems that allow single-copy sharing at the **applications** level, the sharable resources are not known ahead of time. In this case, the system must provide some way for users to express which portions of their programs or data are to be shared and by which processes. To support this in its full generality would require a very complex set of commands and is rarely attempted. But some systems support various limited forms of application-level sharing.

CASE STUDY: SHARING MEMORY IN UNIX

UNIX allows a process to create a section of memory of a specified number of bytes using the command *shmget()* (shared memory get). The command returns a nonnegative integer representing a unique identifier of the shared memory area. Other processes can then map this portion of memory into their space with the command *shmat()* (shared memory attach). This command takes the identifier of the shared memory as a parameter and returns the starting address of the segment in the caller's virtual memory. Alternately, the caller can specify the starting address as another parameter. The original creator of the shared area and all processes that subsequently attach to it may read and write this area (subject to access restrictions specified as part of the *shmget* or *shmat* commands) to share arbitrary unstructured data.

Reentrant Code

When **code** is to be shared among processes, some precautions must be taken to ensure that multiple processes can execute different parts of the same code concurrently. In the early days of computing, instructions could be treated as data and vice versa, which allowed programs to modify their own code. This technique was necessary to implement loops and function calls. The invention of index and base registers made instruction self-modification unnecessary. Today's modern programming languages draw a sharp distinction between code and data, and although code modification is still possible with some languages, it is unnecessary and generally considered poor programming practice. It makes program logic very obscure and difficult to follow. It also makes it impossible to share code because the code keeps changing unpredictably at runtime.

To permit single-copy code sharing, all functions must be **reentrant** or **pure**. Such functions do not modify their own instructions and can be kept as read-only segments. All variable data associated with a process executing a pure function must be stored in separate areas private to the process, including, for example, its *stack* and its *data* area. The process stack stores parameters, return addresses, and local variables. The data area stores global variables and the *heap*, which is used for dynamically allocated memory.

The main challenge in code sharing is to guarantee that addresses generated by the shared code are translated correctly, depending on which process is currently executing the segment. Specifically, references to the segment itself (e.g., branch instructions within the shared code) must translate to the *same* physical addresses, regardless of which process runs. However, references to the stack, data area, or to other code sections must

"runall"
2002/9/24
page 277

all translate to *different* physical address, corresponding to the areas belonging to the process under which the shared code currently executes.

EXAMPLE: Simple Code Sharing

Figure 9-1 shows a code segment shared by two processes, p_1 and p_2. The target address of the *branch* instruction must be translated to the same physical location under both processes. The *load* instruction, on the other hand, must access data from different data areas, depending on which process is currently executing the shared code.

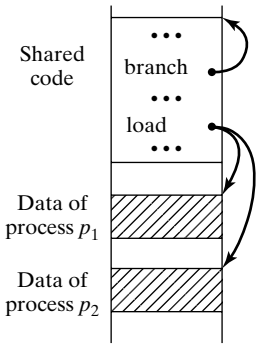


FIGURE 9-1. Addresses in shared code.

9.1.3 Linking and Sharing

Issues of sharing are closely related to linking, which is the process of combining several separately translated programs (object modules) into one load module. The main tasks are the relocation of individual object modules and the binding of external references to reflect the starting addresses assumed by the linker. In systems that do not support segmentation, all modules are linked into *one* contiguous load module. The address space assumed by the linker when generating this load module can be the actual physical address space, a contiguous relocatable space, or a paged virtual space, depending on the memory management scheme supported by the system. But from the linker's point of view, the task in all three cases is the same. For each external reference (M, w), where M is the name of an external module and w is an offset within that module, the linker determines the starting address, m, of M within the load module and adds the offset w to it. The resulting address $m + w$ then replaces all references (M, w).

In systems with segmentation, linking is conceptually simpler. Each segment to be linked is first assigned a segment number. All external references (S, w) to a segment named S are then simply replaced by a pair (s, w), where s is the segment number assigned to S and w is the offset within that segment.

If the linking process is completed, i.e., all symbolic external references are replaced by their corresponding load module addresses, before execution begins, the linking is referred to as *static*. If the linking is performed when execution is already in progress, it is called *dynamic*.

"runall"
2002/9/24
page 278

Let us now consider sharing. This may be viewed as the linking of the *same* copy of a module into *two* or more address spaces. If the links are resolved prior to execution, the sharing is called *static*. This is typically done by the *loader* just prior to program execution. The reason is that the linker does not know whether a shared component is already in memory or still on disk. Furthermore, the component's location can change before the new load module is loaded. Alternately, sharing can be done *dynamically*, where the external links are resolved at runtime. The remainder of this section discusses how sharing may be accomplished under different memory management schemes.

9.2 SHARING IN SYSTEMS WITHOUT VIRTUAL MEMORY

In the absence of any dynamic relocation mechanisms, or if there is only a single relocation register for the entire process, memory must be allocated to a process as a contiguous region at the time of linking or loading. In virtually all such systems, a sharp distinction is made between systems and user programs. Since space is contiguous, the only way for two processes to share any user components would be to partially overlap their memory spaces. Since this is very restrictive, sharing of user code or data is rarely supported.

Sharing of system components is more important and can be supported to some extent. Those system components that must be shared must be assigned specific *agreed-upon* memory addresses. The linker then uses these addresses when preparing a load module. It replaces each reference to a shared system module by its known memory address prior to loading the program. The main problem with this simple scenario is that the shared system component cannot easily differentiate between the different processes that invoke it. The identity of the invoking process or any other process-specific data can be made available to the shared component only via registers, which are saved and reloaded during each context switch.

Most processors provide more than one relocation register for the running process. Typically, a separate register is used for the code, the stack, and the data. This greatly simplifies the problem of sharing. To share the same code among two or more processes (system or user level), all that must be done is to load the code base register of any sharing process with the same address—the starting address of the shared code. The stack and data base registers point to different regions for each process, thus keeping them private.

EXAMPLE: Sharing of Code Using Relocation Registers

The two processes, p_1 and p_2, of Figure 9-2 share the same code but each has its own stack and data. Any reference to the (shared) code would use the code base register (*CBR*). For example, a branch instruction could have the form *branch x(CBR)* where x is the offset relative to CBR. Similarly, any reference to the stack, e.g., to access a local variable, or to the data segment, e.g., to access a variable on the heap, would use the current stack base register (*SBR*) or the data base register (*DBR*). For example, a load instruction could have the form *load x(SBR)* or *load x(DBR)*, where x is the offset for the desired variable relative to the given register. On a context switch to another process executing the same shared code, the content of CBR remains unchanged but the contents of the other base registers are changed to point to the stack and the data of the *new* process.

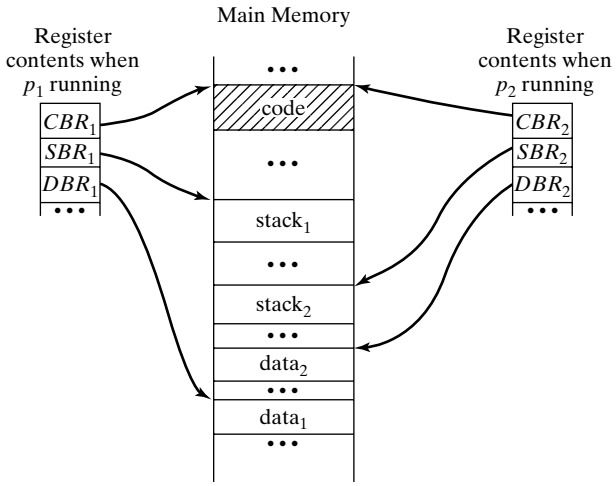


FIGURE 9-2. Sharing of code using relocation registers.

The sharing of data among processes is possible in a similar manner, assuming that additional base registers exist and are accessible by the sharing processes. Setting the base register of two or more processes to point to the same data address allows these processes to access the same region in memory.

9.3 SHARING IN PAGING SYSTEMS

Paging permits a noncontiguous allocation of main memory. Under a standard implementation, each page is referenced by an entry in the process's page table.

9.3.1 Sharing of Data

Sharing among several processes may be accomplished on a page basis by pointing to the same page frame from the different page tables. If the shared pages contain only *data* and no addresses—either to themselves or to other areas—the linker can assign arbitrary page numbers to the shared pages for each process and adjust the page tables to point to the appropriate page frames.

EXAMPLE: Sharing in Paged Systems

Consider two processes p_1 and p_2, whose virtual address spaces are represented by the two page tables PT_1 and PT_2, respectively (Fig. 9-3). Assume that page 0 of each process contains a reference to a shared data page. Under p_1, the shared page is assigned the number $n1$ and hence the corresponding reference is adjusted to $(n1, w)$, where w is the offset within the page. Under p_2, the same shared page is assigned the number $n2$ and thus the reference in p_2's page 0 is adjusted to $(n2, w)$. The page numbers $n1$ and $n2$ need not be the same as long as the shared page contains only data, i.e., the shared page is only read or written, but never executed. The memory mapping mechanism *(address_map)* guarantees that at run time the correct page table is used.

"runall"
2002/9/24
page 280

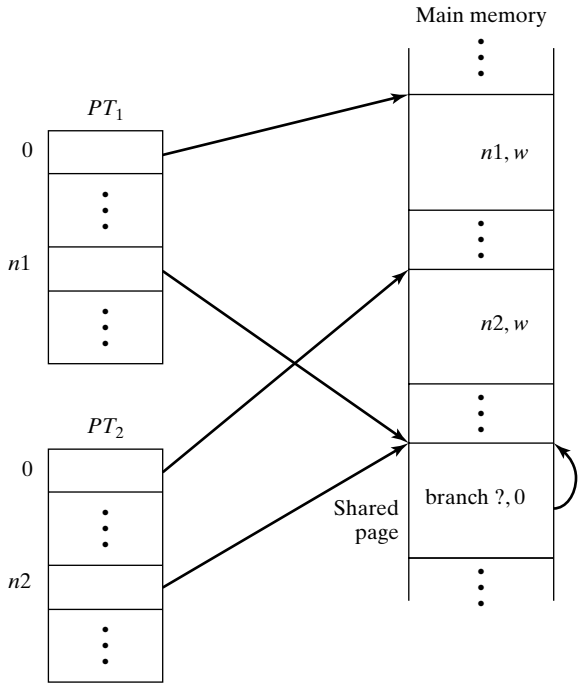


FIGURE 9-3. Sharing in paged systems.

CASE STUDIES: SHARING OF MAIN MEMORY DATA

1. *Linux*. Consider again the *shmget* and *shmat* primitives presented in Section 9.1.2, which allows two or more processes to dynamically declare a shared memory area. When a process performs a *shmget* operation, the system assigns the necessary number of page table entries to the shared area (based on the size of this area). When the process accesses the shared area, free memory pages are assigned and their addresses are entered into the page table. When another process attaches to the same area, using *shmat*, the system designates the same number of page table entries to this area; the caller may choose (using a parameter for *shmat*), whether the same page numbers must be assigned to the shared area as in the creator's page table. Regardless of which page table entries are selected, they are filled with the starting addresses of the same shared pages.

One difficulty with having different page tables pointing to shared pages is swapping. When a shared page is to be removed from memory, the system would have to find all page tables pointing to this page and mark the entries accordingly. To avoid this overhead, Linux keeps track of how many processes are currently attached to a given shared area. Only when a single process is attached is the page actually removed from memory and the (single) page table entry updated.

2. *Windows 2000*. Windows 2000 uses the concept of **section objects** to support sharing of memory among processes. A section object is similar to the shared area obtained with *shmget/shmat* in Linux or UNIX. It is created by one process, which obtains a handle for it. The creator can pass this handle to other processes, which can open and access it. A section object has a size, access protection information, and a designation whether it must be mapped to the same virtual address for all processes sharing it or whether different addresses may be used.

9.3.2 Sharing of Code

Sharing of code requires more care since instructions contain addresses referring to other instructions or data. For shared code pages to execute correctly under different processes, they must be assigned the *same* page numbers in *all* virtual address spaces. To illustrate why this is necessary, consider again the example in Figure 9-3. Assume that the shared page is a function containing a branch instruction to some other location within itself, say to the instruction 0. This instruction would have the form *branch n,0*, where n is the page number of the shared function and 0 is the offset. The problem is that n depends on the process executing the function. When executing under process p_1, the target address must be $(n1, 0)$, and under p_2, it must be $(n2, 0)$. Since a memory cell can hold only one value at any time, the two page numbers $n1$ and $n2$ must be the same. In other words, the shared page must be assigned the same entries in *both* page tables. The potential page number conflicts and wasted table space that can result when a new process must share several already loaded modules limits the ability to share code in paged systems.

The problem can be avoided if the total set of sharable modules is known to the system ahead of time. In that case, their page numbers can be permanently reserved to eliminate conflicts, but such a priori knowledge is rarely available.

To make sharing of code more general, we must avoid using page numbers in the shared code. Assume that the shared code is self-contained, i.e., it contains no external references. Under this assumption, we can avoid the use of page numbers in the code by using only base registers. All branch address are compiled relative to the code base register. Similarly, all addresses of read/write instructions are compiled relative to the data or stack register. A single copy of such a code area can be linked into any address space, and it will execute correctly for all processes sharing it.

The only remaining problem is how to link a single copy of a code area into multiple address spaces. First, the linker must assign page numbers to the area and replace every reference to the area by the address (p, w), where p is the first page of the shared area and w is the offset where execution should start. The linker must then fill the corresponding page table entries with addresses of the pages belonging to the shared code area. It is this second task that makes sharing difficult: the linker does not know whether the shared area is still on disk or already used by some other process and resident in memory. But even if the linker could find this information, it would not help, because by the time the load module is loaded into memory, the location of the shared module could have already changed many times.

To solve this problem, external references can be resolved by the *loader* just prior to execution. But this task can be quite cumbersome, especially when the process specifies a large number of potentially shared routines, most of which are never used. This is

"runall"
2002/9/24
page 282

typically the case when a process includes various system libraries but only uses a small fraction of them during execution. To reduce the overhead, the system can postpone the assignment of page numbers and locating the routines until *runtime*, when a particular function is actually being invoked. However, doing this for every external reference repeatedly would be wasteful. Instead, each external reference should be resolved *only once*, when it is first used. Subsequent references should use the actual address assigned.

To implement this form of **dynamic linking**, the linker sets up a special area within the process address space, called the **transfer vector**. Each entry of the transfer vector corresponds to a references to a shared code segment. All references to a given code segment point to the corresponding entry in the transfer vector. Initially, each entry contains a piece of code, called the **stub**. When a reference is made to a shared segment for the first time, it executes the stub. This checks if the desired segment is currently resident in memory. If not, it loads it. Once the segment is loaded and its address is known, the stub replaces itself by a direct reference to this address. Thus, all future executions of this part of the code will go (indirectly via the transfer vector) to the specified library, without involving the OS.

EXAMPLE: Dynamic Linking Through Transfer Vector

Figure 9-4 illustrates the basic idea. Figure 9-4a shows the virtual memory of a process, where the instruction *bri* (branch indirectly) corresponds to a reference to a shared routine. It points to an entry in the transfer vector, which contains a stub. Figure 9-4b shows the memory after the first *bri* instruction executed. The stub found the corresponding routine in memory and replaced itself by a reference to this routine. Thus the next *bri* instruction, referring to the same shared routine, is able to invoke it without the overhead of linking.

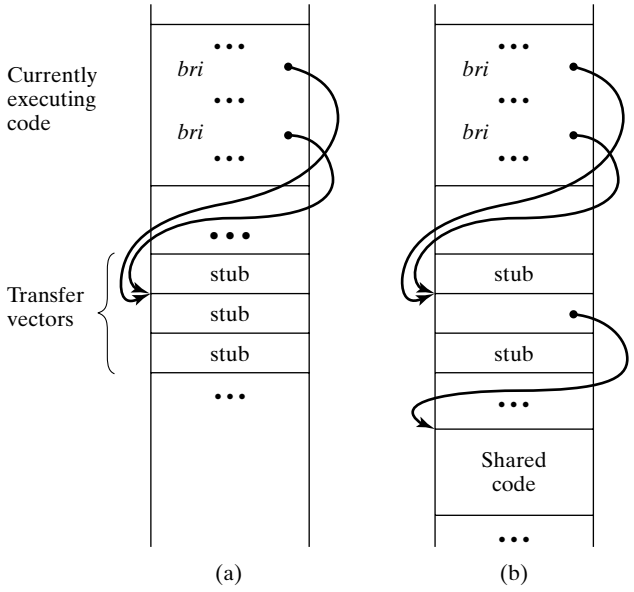


FIGURE 9-4. Dynamic linking through transfer vector.

"runall"
2002/9/24
page 283

CASE STUDY: DYNAMIC LINK LIBRARIES

Windows 2000 uses dynamic linking to implement all its application program interfaces to the operating system, such as Win32 or POSIX. Each interface consists of code files, called **DLLs** (dynamic link libraries). There are several hundred different DLLs, each containing hundreds of library functions. Jointly, the DLLs provide the functionality of the system, from low-level operations on I/O devices to user-level functions to manipulate windows, menus, icons, fonts, cryptography, and many others. Because of the large numbers of DLLs each process could potentially include in its address space, the linker only sets up the transfer vector for the referenced DLLs, but the actual code is loaded and linked to the processes dynamically, only when actually needed.

9.4 SHARING IN SEGMENTED SYSTEMS

In general, sharing of segments is simpler and more elegant than sharing of pages because segments represent natural logical entities such as functions or data files, whereas page boundaries have no correlation with the program structure. The sharing of a segment in memory by two or more processes may be accomplished by pointing to that segment from entries in the different segment tables. These entries point to the absolute locations of the shared segments or, if the memory is also paged, to *shared* page tables.

9.4.1 Sharing of Code and Data

As in the case of paging, sharing of segments containing only *data* presents no serious problems. A shared data segment can have a different segment number assigned to it in each process segment table. The situation is analogous to that of Figure 9-3, but using segments instead of pages. With *code* segments, we face the same problem as with paging, where the shared segment must be able to reference itself correctly, while referring to different stack and data segments, depending on the process under which the segment runs.

We can use the same basic approaches as with paging. The first is to require that all shared function segments have the *same* segment numbers in all virtual spaces. Different variants of this solution have been implemented in several contemporary systems.

CASE STUDY: SHARING OF SYSTEM SEGMENTS

The Intel Pentium processor supports a limited form of segment sharing based on designating a set of segment numbers as shared. The segment table of each process is divided into two halves, called LDT (local descriptor table) and GDT (global descriptor table). The entries in LDT corresponds to segments private to the process, whereas the entries in the GDT are shared among all processes in the system. The latter contain the OS and other sharable components. At runtime, one of the bits of the 14-bit segment number is used to choose between the two tables. The remaining 13 bits then select one of the 8K entries of the chosen table.

The second approach uses again different base registers. A code base register contains the segment number of the currently running process. All self-references by the

"runall"
2002/9/24
page 284

shared segment are then relative to the current segment base register. For example, all branch addresses could have the form $w(CBR)$, where CBR designates the code base register, rather than (s, w), where s is the actual segment number. The offset w remains the same in both cases. In this way, the segment number is part of the invoking process state, rather than part of the code. Consequently, different processes may use different segment numbers for the same shared function.

Unfortunately, this still does not solve the sharing problem in its full generality. In particular, we must make the same assumption as with paging, namely that the shared segments are fully *self-contained*, i.e., they contain no references to other segments. To illustrate this restriction, consider the situation where a shared code segment, $C1$, must refer to another code segment, $C2$. Regardless of whether $C2$ also is shared or is private, $C1$ will refer to it by a segment number. This segment number must become part of the code since a base register cannot be used for an external reference. Consequently, all processes sharing $C1$, while able to use different segment numbers for $C1$, will have to use the same segment number for $C2$. This essentially limits the level of sharing to segments that are not nested, i.e., segments that do not invoke or access any other segments of their own.

9.4.2 Unrestricted Dynamic Linking

Unlike pure paging, segmentation lends itself to the implementation of a fully general scheme of *dynamic linking and sharing*, where *any* two processes (user or system) can share *any* portion of their spaces. Such a scheme was pioneered in the MULTICS operating system (Daley and Dennis 1968), and has subsequently been implemented in many different variations by other systems.

One of the main principles of dynamic linking is to give each segment its own *private* section, called the **linkage section**. This is similar to the transfer vector explained earlier, because it records the segment numbers of dynamically linked segments. Thus, no segment numbers appear directly in the code. This allows different processes to refer to the same segment using different segment numbers. But there is an important difference between linkage sections and the transfer vector: Each segment has its own linkage section while only a single transfer vector is provided for the entire process. The vector is set up statically by the linker; it contains an entry for every external segment (library) the process can ever access. In contrast, the linkage section of a segment records the external references for only that segment. Thus, as new segments are linked to a process, the process incrementally learns about new segments it may have to link to at future references. This allows shared segments to references other segments of their own, which was not possible with the single static transfer vector.

To implement private linkage sections for each segment, base registers must be used for every process: A *code base register* always points to the beginning of the currently executing segment. All internal references (self-references) of the currently executing code segment are interpreted as offsets relative to this base register. A *linkage section base register* always points to the linkage section of the currently executing code segment. All external references are resolved using the contents of this linkage section. The contents of both of these registers change whenever the process transfers control to another code segment.

A *stack base register* is used for all references to the stack segment. For references to global data, a *data base register* could be used. Alternately, each code segment can

"runall"
2002/9/24
page 285

reference various data segments using external references. Note that the contents of the base and data registers change only when the stack or data segments are relocated in memory (swapped out and reloaded at a different location) but otherwise remain unchanged for the lifetime of the process.

CASE STUDY: UNRESTRICTED DYNAMIC LINKING AND SHARING

The MULTICS system developed and implemented dynamic linking in its full generality. It allows code segments to reference other segments while avoiding the involvement of the OS at each external reference after it has been resolved for the first time.

Figure 9-5a illustrates the basic principles of this scheme. It shows a process, represented by its segment table. Segment i is the currently executing code segment C, and segment j is the linkage section associated with C. The linkage section was created originally by the compiler when segment C was compiled, and a private copy was made at the time the process linked to the segment C. The linkage section base register, *LBR*, points to this private copy, while the code base register, *CBR*, points to the code.

All self-references (i.e., branch instructions to within the current segment C) are relative to *CBR*. References to the stack segment are relative to *SBR* (not shown in the figure).

All external references by a segment remain in their *symbolic form* until they are actually needed during execution. At that time, they must be replaced by the corresponding virtual address of the referenced segment. The external references are kept in their symbolic form in a *symbol table*. In Figure 9-5a, we assume that the code C references another segment, S, at an offset W. Thus, the symbolic reference (S, W) is kept in C's symbol table.

The instruction that references this segment is set up by the compiler to point to the external reference via the linkage section. That means, the load instruction within C specifies the offset d. The l flag indicates that d is to be taken relative to the LBR, instead of one of the other base registers, and the star indicates that this is an indirect reference.

The entries in the linkage section consist of two parts: a reference to the symbolic name within the symbol table ((S, W) in the example), and a **trap bit**, initially set to one. When the load instruction is reached for the first time, it finds the trap bit set. The system traps to the OS. The OS invokes the dynamic linker which resolves the external reference as follows. It locates the segment using its symbolic name, S, and loads it into memory. Next it finds a free entry in the process segment table and assigns the corresponding segment number, s, to the new segment. It also translates the symbolic offset, W, into the corresponding numeric offset, w, using the process symbol table. The linker then replaces the entry in the linkage section with the virtual address (s, w) and turns off the trap bit.

Figure 9-5b shows the result of the above transformations. At this point, the segment C is ready to resume execution. The load instruction points to the same location within the linkage section as before. However, this location now contains a valid virtual address (s, w). Since the trap bit is off, the address is resolved by the normal

"runall"
2002/9/24
page 286

CASE STUDY: UNRESTRICTED DYNAMIC LINKING AND SHARING (*continued*)

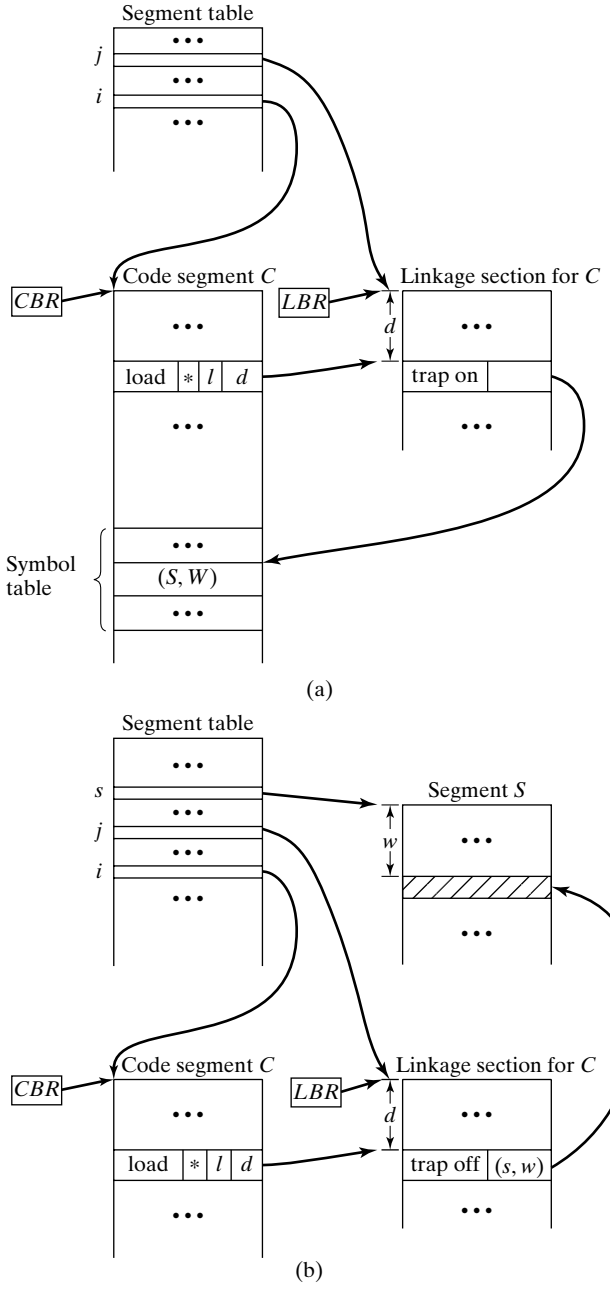


FIGURE 9-5. Dynamic linking/sharing: (a) before external references is executed; and (b) after external references is executed.

"runall"
2002/9/24
page 287

address translating mechanism using the segment table, which yields the desired location w within the segment s.

Note that the indirect addressing required by dynamic linking results in an additional memory access for every external reference. This is the price we must pay for supporting unrestricted sharing of segments among any subset of processes.

9.5 PRINCIPLES OF DISTRIBUTED SHARED MEMORY

The simplest computer architecture consists of a single processor and a single memory module. The previous sections of this chapter all assumed this type. The next level of complexity for a computer system is a multiprocessor, consisting of more than one processor, all sharing the same physical memory. The ability to run multiple programs at the same time complicates the scheduling and other tasks performed by the OS. It also results in a more complicated programming model, because individual applications can take advantage of multiprocessing to speed up their execution. But since they all share the same memory, programming for physical parallelism is no more difficult than programming for logical parallelism. In both the uniprocessor and multiprocessor cases, programs cannot make any assumptions about the sequence in which the various parallel components will be executed, and thus, must include adequate synchronization constructs to assure that CS and other dependencies are obeyed.

The level of complexity rises dramatically when the different processes do not share the same physical memory. In addition to synchronization, the user must now worry about the *location* of code and data in the different memory modules. In particular, any program or process must have all data it needs for its execution in the local memory of the processor on which it is executing. Since some pieces of data are needed by more than one process, the data must be sent between the processors when requested at runtime. This requires the use of message-passing, such as send/receive primitives, which is significantly more difficult and slower than just reading and writing memory directly.

Distributed Shared Memory (DSM) was proposed initially by Li (1986) and later refined by Li and Hudak (1989). The main objective of DSM is to alleviate the burden on the programmer by hiding the fact that physical memory is distributed and not accessible in its entirety to all processors. DSM creates the illusion of a single shared memory, much like a virtual memory creates the illusion of a memory that is larger than the available physical memory. This is accomplished in the following way. When a processor generates an address that maps into its own physical space, the referenced item is accessed locally. When it maps into a memory module belonging to another processor, the OS first transfers the needed data to the requesting processor's memory where it is then accessed locally. In general, the programmer does not have to be aware that data is being passed around using messages. However, given the great overhead of message-passing, sending individual words of data at each memory access would render this scheme too slow to be of any practical value.

To make the basic idea of DSM viable, the underlying data transfers must be done much more efficiently. Two possibilities have been suggested. One is to try to *optimize* the implementation, for example, by exploiting locality of reference or by using

"runall"
2002/9/24
page 288

data replications. The other approach is to *restrict* the full generality of the DSM at the user level. Specifically, we can permit the sharing of only certain portions of the memory space, or we can relax the semantics of the DSM such that it does not mimic the behavior of a physically shared memory in its full generality. The following section discusses these options and trade-offs that make DSM a practical concept. It is convenient to introduce them from the viewpoint of the user or programmer.

9.5.1 The User's View of Distributed Shared Memory

Section 8.2 described the two principal schemes for presenting main memory to a user: The first is a single, linear address region, implemented as contiguous space in physical memory or as a collection of pages. The second method provides multiple segments to better reflect the program structure. The choice of memory structure has important implications for sharing. As was discussed in Section 9.4, sharing of segments is generally easier than the sharing of pages.

Since a DSM is by definition concerned with sharing, the choice of structure becomes even more important. The main questions are:

- Which portions of memory are shared and which portions are private among the different processors?
- Is the sharing fully transparent, i.e., does the logically shared portion of the memory behave exactly as if it was shared physically, or must the user do anything special to accomplish the sharing?

The choices are between a fully shared, unstructured memory and a partially shared, structured one.

Unstructured Distributed Shared Memory

The basic philosophy of this approach is to simulate the behavior of a single, physically shared memory. That means, from the user's point of view, there is only a single linear address space, and it is accessed by all processors in the system concurrently. Figure 9-6 illustrates this choice of memory structure. The shared memory is the union of the individual physical memories, MMi, belonging to the different processors, Pi. Under this scheme,

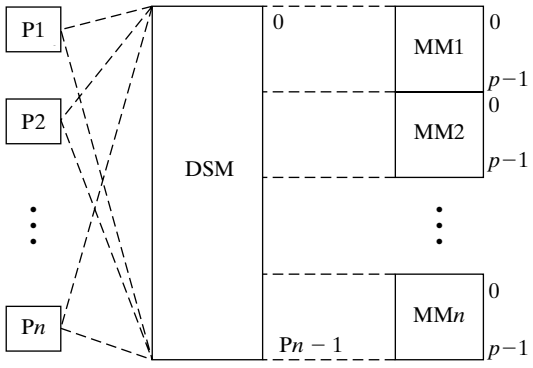


FIGURE 9-6. Unstructured DSM.

"runall"
2002/9/24
page 289

each processor can generate addresses in the range from 0 to $pn - 1$, where p is the size of each physical memory module (assumed all the same size here) and n is the number of modules. The underlying implementation translates these addresses to the appropriate physical memories. This creates the illusion of a virtual memory that is larger than the local physical memory, much like virtual memory implemented with paging. The main, and very important, difference between paging and this form of DSM is that the virtual space is not private with DSM but is accessible by all the other processors in the system.

The principal advantage of this unstructured organization is that the existence of physically disjoint memory modules is *fully transparent* to the user. That means, a concurrent program written for a system with a single physical memory can execute on a system using this form of DSM without any modifications. Concurrent computations also can communicate and coordinate their operations using this DSM. For example, semaphores can be used to synchronize access to CSs, and shared memory areas can be used to exchange data; there is no need to use any form of message-passing.

One drawback of such a fully transparent DSM, where the entire space is shared, is *efficiency*. Since each processor can access any address in the entire DSM, every instruction fetch or operand read/write potentially generates an access to a nonlocal memory module. Transferring each data item individually and on demand through the network is obviously not a viable approach. Section 9.6.1 outlines various ways to make this form of DSM practical.

Structured Distributed Shared Memory

Most programs are structured internally as collections of functions and data structures (variables). Those parts of a program to be shared with other objects usually coincide with the program's logical decomposition. In general, only specific functions and data structures must be shared. Since it is the programmer who decides what is to be shared and by whom, it is not unreasonable to require that the programmer specify this information explicitly. This can be done at the programming language level by defining some additional notations to indicate those functions or data structures that are private and those that are sharable.

Figure 9-7 illustrates this shift of paradigm. It shows a system of multiple processors, each with its own private physical memory. The logical address space available to the processor can be equal to the physical memory, or, if virtual memory is implemented locally, the logical space can be larger than the physical memory. The important fact is that some portion(s) of each of the logical spaces are shared among *all* processors. This shared portion is the DSM. When any of the processors writes into this portion of memory, the change becomes visible automatically and transparently by all the other processors, just as if the shared portion was implemented as a physically shared memory module.

This organization essentially segregates shared and nonshared variables, and is the first step toward a more efficient implementation of DSM. But it still requires too many data transfers between the processors. Thus, most systems impose additional restrictions on the use of the shared variables. The standard technique is to allow access to any shared variable only within CSs, which must be declared explicitly. Since CSs are by definition mutually exclusive, propagating the modifications of any of the shared variables can be limited to two well-defined points: either at the *beginning* or at the *end* of the CS. This idea leads to new forms of memory consistency, treated in Section 9.6.2, and results in greatly improved performance of the DSM.

"runall"
2002/9/24
page 290

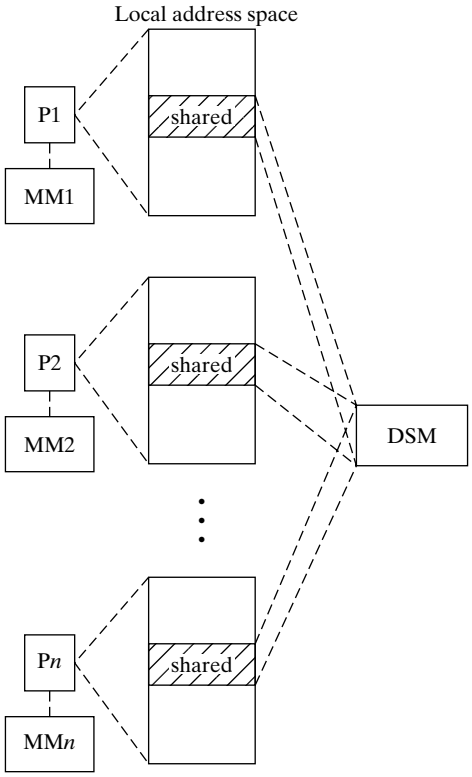


FIGURE 9-7. Structured DSM.

An attractive variant of the basic structured DSM is to use *objects* instead of just passive variables. Object-oriented programs are structured as collections of objects, whose functions are invoked from a main function or from other objects. This logical structuring provides a very natural basis for sharing. Whenever two processes execute the functions of a given object, they manipulate the same copy of the data; i.e., they are sharing the object.

This form of object sharing can be implemented transparently in an architecture where multiple processors do not share any physical memory. The resulting object-based DSM hides the fact that objects are distributed over multiple disjoint memory modules by providing the necessary mechanisms to access the remote objects.

9.6 IMPLEMENTATIONS OF DISTRIBUTED SHARED MEMORY

9.6.1 Implementing Unstructured Distributed Shared Memory

When a process references a data item that is not present in its local physical memory, it causes a fault. In response, the DSM runtime system must locate the missing data item and transfer it to the faulting process' local memory. There are several fundamental questions that lead to different schemes for building unstructured DSM. The key issues are the granularity of data transfers, replication of data, memory consistency, and keeping track of the data throughout the system.

"runall"
2002/9/24
page 291

Granularity of Data Transfers

It would be grossly inefficient to move each referenced data item individually through a network. Due to program locality, it is likely that a process will reference other data items in the vicinity of the currently accessed item. Thus, transferring a sequential block of data surrounding a missing data item is more efficient than transferring the data by itself.

The *size* of the data block is critical to performance, and its selection follows the same logic of that for choosing a page size (Section 8.3.4). In a memory system that uses paging, a natural choice for a DSM implementation is to use the existing page size as the granularity for transferring referenced data between processors. However, although transferring data through local networks is generally faster than from even local disks, the startup cost of a network transfer is very high. To amortize this overhead, some DSM systems choose multiple pages as the basic unit of transfer whenever a page fault occurs. Just like a larger page size, this increases network traffic. But, the more serious drawback of such increased granularity is a phenomenon called **false sharing**. It occurs when two unrelated variables, each accessed by a different process, end up on the same page or set of pages being transferred between memories. This causes a situation analogous to thrashing, where the pages are transferred back and forth as the two processes reference their respective variables. Obviously, the larger the unit of transfer, the greater the chance for unrelated variables to become transferred together.

Replication of Data

The second fundamental decision to be made when implementing a DSM concerns the possible **replication** of shared pages. That means, when a page fault occurs, should the requested page be actually *moved* from the remote to the requesting processor, or should only a *copy* of the page be transferred. The obvious advantages of maintaining multiple copies of a page are decreased network traffic, less thrashing, and reduced delays resulting from page faults, since any page only must be transferred once. The main problem is how to maintain consistency in the presence of multiple copies. Whenever a process modifies a shared page, the change must be made visible to all other processes using that page. The time at which the changes become visible is crucial to preserving the correctness of the programs. Figure 9-8 illustrates the problem. It shows two computations, p_1 and p_2, reading a shared variable x. Depending on the application, p_1 and p_2 could be independent processes or concurrent branches or threads of a common program.

At time $t1$, p_1 writes a 1 into x. At time $t2$, both processes read x. We would naturally expect that the values of $a1$ and $a2$ will both be 1, i.e., the most recently written value. Similarly, the values of $b1$ and $b2$ read by the processes at time $t4$ should both be 2, since this was the most recently written value written into x (by process p_1).

Ensuring that these expectations will be met in the case where each process is running on a different processor and the variable x is replicated in the corresponding local memories is a difficult problem. It is similar to the problem of maintaining cache

intial:	$x = 0$						
real time:		$t1$		$t2$	$t3$	$t4$	
$p1$:	{	$x = 1;$		$a1 = x;$	$x = 2;$	$b1 = x;$	}
$p2$:	{			$a2 = x;$		$b2 = x;$	}

FIGURE 9-8. Example of strict consistency.

"runall"
2002/9/24
page 292

Operation	Page Location	Page Status	Actions Taken Before Local Read/Write
read	local	read-only	
write	local	read-only	invalidate remote copies; upgrade local copy to writable
read	remote	read-only	make local read-only copy
write	remote	read-only	invalidate remote copies; make local writable copy
read	local	writable	
write	local	writable	
read	remote	writable	downgrade page to read-only; make local read-only copy
write	remote	writable	transfer remote writable copy to local memory

FIGURE 9-9. Protocol for handling DSM pages.

coherence in a multiprocessor system, where each processor keeps copies of the most recently accessed data items in its local cache. The solution to the cache coherence problem is to broadcast all write operations performed by any processor to all others. The receivers of the broadcast then either **update** all copies of the same data item held in any of the caches, or they **invalidate** these entries so that the updated value needs to be reloaded from memory.

The solutions to cache coherence require a hardware broadcast that performs the updates or invalidation atomically. Unfortunately, DSMs are implemented in software, generally on top of existing OSs; and thus performing an atomic broadcast each time a variable is modified would render the DSM hopelessly inefficient. One practical method differentiates between **read-only** and **writable** pages. Consistency is preserved if only a single copy of any writable page is maintained, while allowing read-only pages to be replicated freely. Unfortunately, it is not always known in advance which pages will be modified and which will only be read, and thus we must permit pages to change their state between *read-only* and *writable* during execution. There are different protocols to assure consistency during such transitions. One popular representative is based on the rules given in Figure 9-9. This specifies the actions that must be taken, based on the current operation (*read* or *write*), the location of the page (*local* or *remote*), and the state of the page (*read-only* or *writable*). The following example illustrates the use of these rules.

EXAMPLE: Structured DSM Operations

Figure 9-10 shows two processors accessing two pages, *A* and *B*, in the DSM. Page *A* is writable and resides in P1's physical memory MM1. Page *B* is read-only, and each processor has a copy in its own physical memory. The following table shows certain operations performed by a processor and the corresponding actions taken by the system.

P1 reads A	operation is done locally
P1 writes A	operation is done locally (page is writable)
P1 writes B	invalidate copy in MM2; upgrade copy in MM1 to writable
P2 reads A	downgrade page in MM1 to read only; make copy in MM2
P2 writes A	transfer page from MM1 to MM2

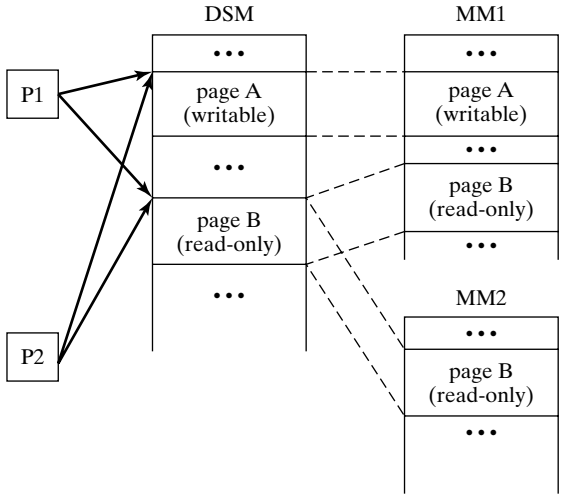


FIGURE 9-10. Example of page operations.

The above protocol treats the DSM coherence problem like the readers-writers problem in Section 3.3.1. That means, it allows multiple concurrent readers or a single writer to access the same data page. But in doing so, it must delay certain read/write operations until the intentions have been communicated to remote processors and the necessary changes prescribed by the rules of Figure 9-9 have been made. Specifically, when a read operation for a remote writable page has been issued, the current owner of the page will continue writing until it has been notified of the intended read. Consequently, the read operation will see a value that has been written later in real time. This could not occur if the memory was shared physically. Similarly, a write operation to a remote page will be delayed until all remote copies have been invalidated. In the meantime, the remote processes may be reading the current values. Again, a physically shared memory would automatically prevent any such delayed writes from occurring.

As a result, executing the same program on a system with physically shared memory could yield different results than executing it on a system with DSM. One way to make this behavior acceptable is to adopt a different model of memory consistency for the shared memory, that relaxes the requirements on how the DSM must behave with respect to real time yet still mirrors reality and intuition. We describe this **sequential consistency** model next.

Strict Versus Sequential Consistency

Most programs do not rely on real-time for their synchronization. When two branches of a program have been designated as concurrent, for example, using some of the constructs discussed in Chapter 2, the assumption is that these branches are independent. Consequently, they remain correct regardless of whether they are executed in parallel, in sequence (in either order), or by arbitrarily interleaving their instructions. When a particular ordering of instructions is required, it must be enforced explicitly by synchronization commands, such as semaphores or message-passing.

```
initial: x = 0
(a) p1:  {                          x = 1;    a1 = x;   x = 2;   b1 = x;}
    p2:  { a2 = x;      b2 = x;}

(b) p1:  {              x = 1;    a1 = x;   x = 2;    b1 = x;}
    p2:  { a2 = x;                b2 = x;}

(c) p1:  { x = 1;       a1 = x;   x = 2;    b1 = x;}
    p2:  {              a2 = x;                       b2 = x;}
```

FIGURE 9-11. Examples of possible instruction interleaving.

Consider the example of Figure 9-8. Except in very special situations of dedicated real-time systems, the two processes p_1 and p_2 should not rely on the fact that both perform the read operations at exactly the same point in real time. Thus, unless explicit synchronization constructs are included, the instructions of p_1 and p_2 could be interleaved in many different ways. Figure 9-11 shows examples of three possible interleavings. In all three cases, p_1 will see $a1 = 1$ and $b1 = 2$, since these were the values it wrote into x just prior to the reads. Any other behavior of the memory would not be acceptable. For p_2, however, the situation is different. In case (a), it will see $a2 = b2 = 0$ because p_1 has not yet overwritten the initial value. In case (b), p_2 will see $a2 = 0$ and $b2 = 1$. In case (c), it will see $a2 = 1$ and $b2 = 2$—the same values as read by p_1.

Based on the above observations, we can now formulate two different forms of memory consistency models for DSM (Lamport 1979):

- **Strict consistency:** A DSM is said to obey strict consistency if reading a variable x always returns the value written to x by the most recently executed write operation.
- **Sequential consistency:** A DSM is said to obey sequential consistency if the sequence of values read by the different processes corresponds to some sequential interleaved execution of the same processes.

To illustrate the difference, consider Figure 9-8 once again. Under strict consistency, both processes must see $x = 1$ with their first read operation and $x = 2$ with the second. The reason is that, for both processes, the first read operation (issued at real time $t2$) was executed after the write operation (issued at real time $t1$). In contrast, the same sequence of instructions executed under sequential consistency is considered correct as long as:

1. The two read operations of p_1 always return the values 1 and 2, respectively.
2. The two read operations of p_2 return any of the following combinations of values: 0 0, 0 1, 1 1, 1 2, 2 2.

The sequential consistency model is the most widely used model implemented for unstructured DSMs. It satisfies the intuitive expectations placed on memory for distributed programming. At the same time, it lends itself to a much more efficient implementation than the strict consistency model, since the results of write operations may be propagated to remote memories with delays and without compromising program correctness.

"runall"
2002/9/24
page 295

Finding Pages

When pages are allowed to migrate among the different processors, some mechanisms must be implemented to allow a processor to find the page it wishes to access. With replication, the problem is complicated further by the fact that multiple copies of the same page exists, all of which must be found when a write operation must invalidate them. The replication problem is usually solved by designating one of the processors as the **owner** of a given page and adopting the following policy.

When the page is writable, there is only one copy, and hence the processor currently holding the page is by default the owner. When a page is degraded to read-only, a copy is made for the requesting processor, but the ownership remains with the original owner. This owner keeps track of all the copies made of this page by maintaining a list, called the **copy set**, of all processors who requested a read-only copy. The page ownership is transferred to another processor only when a write request is received. In this case, all processors on the copy set are sent a message to invalidate their copies; this can be done by the original or the new owner. When this is completed, the new owner can write into the page. The owner also becomes responsible for maintaining the new copy set whenever another processor requests a read access to the page.

Several different schemes have been proposed for finding a page owner (Li and Hudak 1989). The simplest uses **broadcasting**. Whenever a processor must access a page that is currently not it its local memory, it broadcasts the request to all other processors in the system. The current owner then responds by transferring the page to the requesting processor. The main drawback of this method is the overhead resulting from the various broadcasts, each of which interrupts all processors.

Another simple technique designates one processor as a **central manager** to keep track of the current location of all pages. Unfortunately, the central manager is likely to become a bottleneck, rendering this solution not scalable. A **replicated manager** approach can alleviate the problem; here, several processors serve as managers, each responsible for a designated subset of the pages.

The last scheme is based on the concept of a **probable owner**. Each processor keeps track of all pages in the system using a data structure similar to a page table in a virtual memory system. For local pages, the corresponding entry in this table points to the page in local memory. When a page is transferred to another processor, say P, the table entry records P and the new owner of the page also becomes P. When the original processor must access the page again, it will send the request to the processor whose address it had recorded as the new owner. Unfortunately, the page may have migrated on to other processors in the meantime. Each migration would record the address of a new owner in its own table. A desired page can be found by following the chain of addresses, each pointing to the next probable owner. To make subsequent accesses to the same page less costly, the chain can be dissolved by making all probable-owner pointers point to the last node along the chain, i.e., the actual current owner.

Unstructured DSM implementations assume that all variables in the shared space are consistent at all times. This involves moving and/or invalidating pages whenever the access to a page changes from reading to writing or vice versa. Much network traffic can be generated, resulting in poor performance. The alternative, structured DSM, requires a new model of memory consistency.

9.6.2 Implementing Structured Distributed Shared Memory

Distributed Shared Memory with Weak Consistency

Weak consistency (Dubois et al. 1988) is based on the premise that the user knows and can identify those points in a distributed program where shared variables must be brought into a consistent state. To do this, a new type of variable, a **synchronization variable**, is introduced. Only when the concurrent processes access this shared variable are the effects of the various local write operations propagated to other processes, thereby bringing the shared memory into a consistent state. In other words, the DSM is guaranteed to be in a consistent state only immediately following the access to a synchronization variable.

EXAMPLE: Weak Memory Consistency

Figure 9-12 illustrates the principles using two concurrent processes p_1 and p_2, and a synchronization variable. Both processes independently modify a shared variable x, and subsequently read it. Since each process has its local copy of x (step 1 in Fig. 9-12), the modifications remain local until the shared memory is explicitly synchronized. Thus the value $a1$ (written by p_1) will be 1, whereas the value of $a2$ (written by p_2) will be 2. After both processes execute the synchronization S, the values of all shared variables are exchanged among the processes and a consistent state is established (step 2 in the figure). Therefore, the results of all read operations immediately following the synchronization will be identical in all processes. That means, both $b1$ and $b2$ will either contain the value 1 or the value 2, depending on the results of the consistency exchanges (step 2).

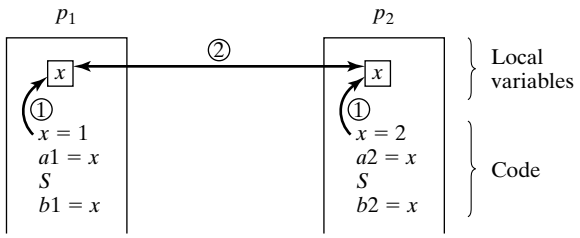


FIGURE 9-12. Weak memory consistency.

Distributed Shared Memory with Release or Entry Consistency

In many applications, the number of shared variables is small. By requiring the user to explicitly identify which variables are shared, we can dramatically reduce the amount of data that must be moved and/or invalidated by the DSM implementation. Furthermore, shared variables are typically accessed only within CSs. Since CSs are by definition mutually exclusive, only one process can read or write the shared data. Thus, there is no need to propagate any writes when the process is inside the CS. All we must guarantee is that such shared variables are consistent whenever a process is about to enter a CS. The exchanges can be performed at the **exit** or the **entry** point of a CS. The following example illustrates the first choice.

"runall"
2002/9/24
page 297

EXAMPLE: Release Memory Consistency

Figure 9-13 illustrates the first option. Assuming that p_1 enters the critical section before p_2 by executing the *lock* operation first (implemented, for example, using a spinning lock or a semaphore), it writes 1 into x (step 1) and then exits the critical section by executing *unlock*. The modified variables are exported to all other processes at the end of the CS (step 2). Then a is assigned the new value of 1 (step 3).

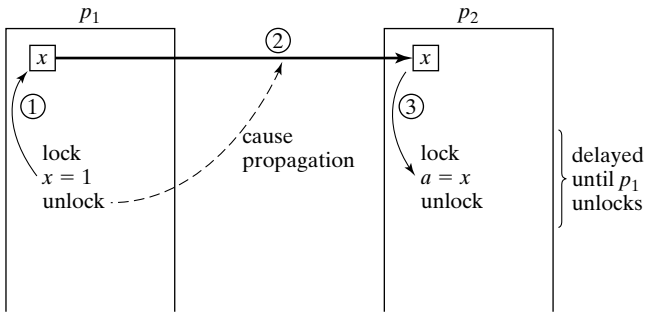


FIGURE 9-13. Release memory consistency.

This model, referred to as **release consistency** (Gharachorloo 1990) is straightforward but is wasteful in that the modifications are exported to all processes, regardless of which, if any, will be accessing them next. The unnecessary overhead can be eliminated by performing the exchange at the *entry* point to the CS, instead of the exit.

EXAMPLE: Entry Memory Consistency

Figure 9-14 illustrates this option. We again assume that p_1 executes the *lock* operation before p_2 and updates x (step 1). The *lock* operation of p_2 then causes the value of x to be brought up to date (step 2) prior to entering the CS and reading x (step 3).

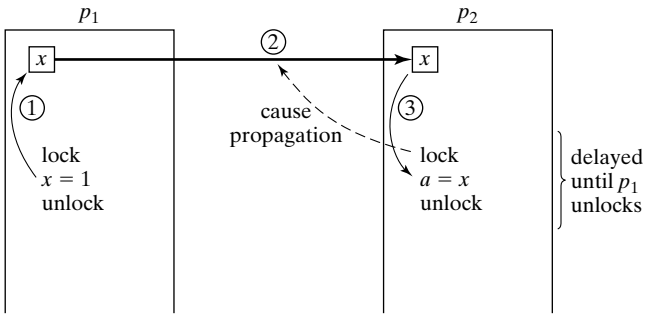


FIGURE 9-14. Entry memory consistency.

"runall"
2002/9/24
page 298

The above form of consistency comes in two forms: **lazy release consistency** (Keleher et al. 1992) or **entry consistency** (Bershad 1993). The difference between the two is that the former imports *all* shared variables whereas the latter associates each shared variable with a lock variable and imports only those shared variables pertaining to the current lock. In situations where a process repeatedly enters the same CS, no data is exchanged until another process wishes to enter the same CS.

Object-Based Distributed Shared Memory

Using especially designated variables as the basis for sharing reduces the amount of data that must be moved between processors. But for a shared variable to be accessible by a processor, it must be transferred or copied into the processor's local memory. With objects, the situation is different. There is no need to transfer a remote object into local memory to access the data it encapsulates. This is because the functions or methods that access the data are always part of the object. If the system provides the ability to perform *remote* methods invocation, similar to remote procedure calls as discussed in Chapter 3, then the DSM implementation can choose between moving an object, copying it, or accessing it remotely, to maximize performance.

When objects are replicated, maintaining consistency again becomes an issue. We can implement a scheme similar to that used for page-based DSMs, where read-only accesses can proceed concurrently while a write access invalidates all replicas of the corresponding object. Alternatively, because of the ability to invoke methods remotely, a write access can be sent and executed remotely at all replicas, rather than invalidating them. This corresponds to the update protocol implemented in some caches, but requires a reliable atomic broadcast capability, which is costly to implement.

CONCEPTS, TERMS, AND ABBREVIATIONS

The following concepts have been introduced in this chapter. Test yourself by defining and discussing each keyword or phrase.

Consistency in DSM	Replicated pages in DSM
Distributed shared memory (DSM)	Segmented systems sharing
Dynamic linking and sharing	Sequential consistency
Dynamic link libraries (DLL)	Single-copy sharing
Entry consistency	Static linking and sharing
False sharing	Strict consistency
Linkage section	Structured DSM
Object-based DSM	Unstructured DSM
Paged systems sharing	Weak consistency
Release consistency	

EXERCISES

1. Consider the problems of static sharing in a purely paged system. Assume a process p_a requires programs Q_1 and Q_2, temporary storage T_1, and data D_1, of size q_1, q_2, t_1, and d_1, respectively, in pages. These are combined and linked into a virtual space program of size $q_1 + q_2 + t_1 + d_1$ pages in the above order and loaded into memory.

While p_a is still active, another process p_b requests loading of information Q_1', Q_2, T_1', and D_1 with page sizes q_1', q_2, t_1', and d_1, respectively, where $q_1' \leq q_1$ and $t_1' > t_1$. Q_2 and D_1 are to be shared between p_a and p_b.

(a) Show the page tables for the two processes.

(b) What problem arises if a new process p_c now enters the system and wants to share Q_1 and Q_1'?

2. Consider two processes p_1 and p_2 in a purely segmented system. Their current segment tables contain the following entries:

	ST1
0	4000
1	6000
2	9000
3	2000
4	7000

	ST2
0	2000
1	6000
2	9000

(a) Which segments are shared?

(b) Which of the following virtual addresses would be illegal in the segment at location 6000: (0,0) (1,0) (2,0) (3,0) (4,0) (5,0)?

(c) The segment at location 7000 is swapped out and later reloaded at location 8000. Similarly, segment at location 2000 is swapped out and reloaded at 1000. Show the new segment tables.

(d) A third process p_3 wishes to share the segments at locations 2000, 4000, and 5000. Which of these must be data segments (rather than code segments) to make this possible?

3. Consider a system with segmentation (no paging). Two processes, p_1 and p_2, are currently running. Their respective segment tables have the following contents:

	ST1
0	1251
1	0
2	810
3	145
4	2601
5	380

	ST2
0	512
1	1020
2	1477
3	145
4	1251
5	380
6	3500

Make the following assumptions:

- p_1 is currently executing in its segment 2;
- p_2 is currently executing in its segment 1;
- base registers are used for all self-references (references to the current segment).

Assume that both processes references the following virtual addresses, where "*BR, n*" denotes a self-reference (an offset n relative to base register BR):

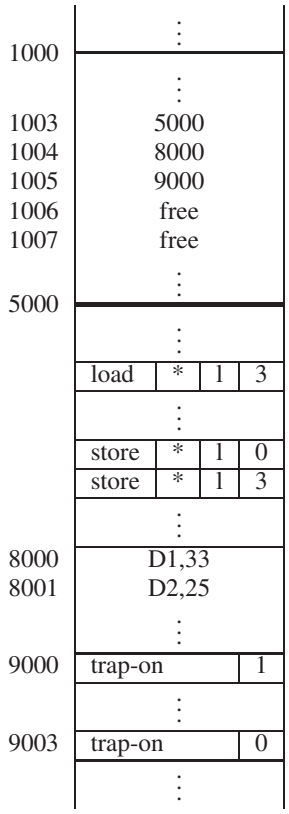
(3, 0), (5, 20), (4, 5), (6, 10), (BR, 0), (BR, 10)

Translate each virtual address into the corresponding physical address for both processes p_1 and p_2. If an address is illegal, indicate so.

"runall"
2002/9/24
page 300

4. Consider a system with unrestricted dynamic linking. The memory snapshot below shows the various segments of a process p_1. Make the following assumptions:

- The segment table of the process p_1 begins at address 1000. (The segment table is not paged.)
- The process is currently executing inside segment 3, which is a code segment.
- Segment 4 is the symbol table corresponding to segment 3.
- Segment 5 is the current linkage section.
- The segments named $D1$ and $D2$ are already in memory, residing at addresses 400 and 600, respectively.



(a) What are the current contents of the code base register (*CBR*), linkage pointer register (*LBR*), and segment table register (*STR*)?

(b) What other segments does the currently executing code segment reference? What are their symbolic names and entry point offsets?

(c) Show all changes to the memory after each of the three instructions (load, store, and store) are executed.

(d) Consider a second process p_2 wishing to share the code segment at address 5000 (and all other segments referenced by this segment). Assume that the segment table of p_2 begins at address 2000 and already contains 5 segment entries. Assume further that the linkage section of p_2 begins at address 3000. Show all changes to the memory *before* and *after* p_2 executes the code segment.

"runall"
2002/9/24
page 301

5. Consider two concurrent processes, p_1 and p_2, running on two different processors and sharing a two-dimensional array $A[n, n]$ kept in a page-based DSM. Make the following assumptions:

- Each processor has enough memory to hold the entire array A.
- Initially, the array is split evenly over the local memories of the two processors (processor 1 holds the first $n/2$ rows, processor 2 holds the remaining $n/2$ rows).
- The page size used by the memory system is n.
- The array is stored in row-major.
- Each process must access every elements of the array exactly once (reading or writing it).
- A page is transferred instantaneously (no overhead) as soon as any element of that page is referenced.

(a) Approximately how many pages will have to be transferred between the two processors in each of the following cases:

i. only one of the processes runs; it accesses A in row-major.

ii. both processes access A concurrently in row-major; both start with element $A[0, 0]$.

iii. both processes access A concurrently in row-major; both start with element $A[0, 0]$ but the execution of p_2 is delayed by n steps.

iv. both processes access A concurrently in row-major; p_1 starts with element $A[0, 0]$ while p_2 starts with element $A[n - 1, n - 1]$.

v. both processes access A concurrently in column-major; both start with element $A[0, 0]$.

How would your answers to the preceding questions change if:

(b) all accesses to the array A were read-only.

(c) a larger or smaller page size was used.

6. Consider a page-based DSM that obeys the protocol of Figure 9.9. Assume there are three processes, p_1, p_2, and p_3, all accessing a single shared page. Give the actions carried out by the DSM implementation for the following sequence of operations. (Assume the page p is currently writable in p_1's local memory.)

- p_1 writes p
- p_2 reads p
- p_3 reads p
- p_1 reads p
- p_2 writes p
- p_3 writes p

7. Consider two processes p_1 and p_2 sharing a single variable x.

(a) Give a sequence of read and write operations by the two processes that obeys sequential consistency but violates strict consistency.

(b) Is it possible to construct such a sequence under the following assumptions:

i. process p_1 only reads while process p_2 only writes

ii. both processes only read

iii. both processes only write

8. Consider two concurrent processes, p_1 and p_2, performing the following operations on two shared variables, x and y:

```
initial: x = 0; y = 0;
p1: { x = 1; a1 = y; b1 = x; }
p2: { y = 1; a2 = x; b2 = y; }
```

"runall"
2002/9/24
page 302

At the end of the execution, the value of each of the local variables a_1, b_1, a_2, and b_2 can be zero or one. For the 16 possible combinations or these values, determine which ones are valid under sequential consistency.

9. Consider two concurrent processes, p_1 and p_2, performing the following operations on two shared variables, x and y:

```
initial: x = 0; y = 0;
p1: { x = 1; y = 1; }
p2: { a1 = x; a2 = y; S; a3 = x; a4 = y; }
```

At the end of the execution, the value of each of the local variables a_1, a_2, a_3, and a_4 can be zero or one. For the 16 possible combinations or these values, determine which ones are valid under weak consistency, where S is the synchronization statement.