Tutorial: An Introduction to Shared Memory

William Sumner

Hewlett Packard Corporation
3000 Waterview Parkway
Richardson, Texas 75080

Voice: 972-497-4642
Fax: 972-497-4245
bill_sumner@hp.com

Introduction and Background

This paper presents a short tutorial on the use of shared memory and memory-mapped files in a multi-process or multi-threaded environment. It is meant to be an introduction to the topics and not an exhaustive nor an in-depth study. It presents the basic definitions for the terms used, introduces the system calls available in this area, and discusses the concepts necessary for using shared memory and memory-mapped files. Additionally it attempts to mention the key difficulties that may trip-up the first-time user.

To prepare for this discussion, it is helpful to start with a common set of definitions for the terms in use. The remainder of the talk will expand on each of these.

In this paper the term **UNIX** refers to the generic UNIX-style operating system while the term **HP-UX** refers to the Hewlett Packard implementation of UNIX.

A **Process** is the context for everything a user does in a UNIX system. It holds things like the userid, the group-id, a set of environment variables and the program that we are running. Of particular interest to this talk -- a process has a virtual address space and is a container for one or more threads of execution.

The **Virtual Address Space** is a range of addresses, unique to this process, into which we can map various items of storage. These items include an executable program, a private memory area for our stacks and heaps, and a jumptable through which we request operating system services, among others. This paper concentrates on two items which may be shared among two or more virtual address spaces -- a memory-mapped file, and a shared memory segment.

The term **Shared Memory Area** will be used in this paper to refer to either a memory-mapped file, or a shared memory segment without regard to which one it really is-- because most of the principles operate the same on both. There is only one physical copy of a given shared memory area regardless of how many processes have mapped it into their virtual address space. All of the processes read and write the same physical copy. When a process changes the value of a shared data item, the new value is seen immediately by all of the processes.

A **Shared Data Item** is any data item that is accessed (for read or write) by more than one thread. This paper only discusses shared data items in a shared memory area. Although many of the principles also apply to the private data area of multi-threaded applications, this paper will not address sharing data items in a process's private data area.

Data items within a shared data area may be allocated either statically or dynamically.
A typical static shared data item might be a:

- Lock
- Simple Data Item (char, int, ulong, etc.)
- Structure

- Vector of Structures or Simple Data Items (Buffer, Multi-dimensional array)
- Pointer

A typical dynamically allocated data item might be a:
- Lock
- Structure
- Vector of Structures or Simple Data Items
- Complex Organization of Structures
  - Linked List
    - Queues and Stacks
  - Structures containing pointers to Structures
  - Hash Table
  - Tree

A **Thread** is UNIX's abstraction of a processor. It is a virtual central processing unit, complete with registers (all the data contents of a real processor), a private stack, and optionally some private thread-local storage. A multi-threaded application can use any of the following kinds of threads:
- Multiple processes, each with one or more kernel threads accessing a shared memory area or a memory-mapped file. .... Subject of this paper.
- Multi-threaded process (Kernel threads).............................. Some applicability.
- Lightweight processes (User threads) .... Not covered herein
- M x N Threads (Combination of Kernel and User Threads). Not covered herein

Additionally, the ***access patterns*** for the shared data items determine the complexity of the programming task.

- No Writers -- Static read-only data
  - No problem, everybody just reads it.
  - Typical case: Multiple processes running the same program
- One Writer, One or more Readers
  - Typical case: Data for an online, web-server catalog
  - Writer may need to say, "New data available."
  - Reader may need to say, "I'm finished."
- Multiple Writers (Including "read and update")
  - Typical case: Online order processing updating inventory

As the complexity of the access pattern increases, so does the need for synchronization of the threads. The synchronization process includes notification, locking, and the use of semaphores.

Process Structure, Threads and Shared Memory Area,

Process Structure

Let's take a look at the structure of a typical process. It's a container for memory and one or more threads. In HP-UX, the memory is divided into ***quadrants,*** each of which is 1/4 of the process's total virtual memory range.

Q1: Read-only area for programs
Q2: Data area private to this process and its threads
Q3: Memory shared with other processes
(Shared memory segments and memory-mapped files)
Q4: HP-UX (Read-only) (Gateway page, I/O maps, etc.)

By default a process has this memory structure and one thread.

Q1:

Q1 contains the executable text of the program that this process is running. This is always read-only, so we are not really concerned with any memory-sharing in this quadrant. However, HP-UX does it for us and we get the benefit somewhat indirectly. When 100 processes are all running the same program (say "vi"), HP-UX loads only one copy into real memory and points all 100 processes at that one copy -- saving lots of real memory that can be used for other things.

Q2:

Q2 is the private data area of a process. This area is shared among all threads belonging to this process but is not shared with other processes. Note that when a process forks[1] -- creates a child -- the child process gets a *copy* of this area. The child does not share the Q2 space.

Most programs run as a single process with a single thread. In this case, the one thread simply considers the Q2 data area as private to itself and uses it. On the other hand, if the program creates additional kernel threads or uses the lightweight-thread package (LWP), then some or all of the data items in Q2 can be shared among the multiple threads of this process. The shared items need to be protected against simultaneous update -- as described later in the section on locks.

Q3:

When we talk about sharing memory between two processes, we are talking about the Q3 memory area -- either shared memory segments allocated with **shmem()** or memory-mapped files allocated with **mmap()**. In HP-UX, chunks of shared memory in Q3 are mapped into the same virtual memory address in all of the processes that share the

[1] Depending on how you do the "fork()", and up until you do the "exec()" it is possible to share this area. However, the child is not allowed to change anything in this area before the "exec()". If it does, the results are undetermined (read "usually catastrophic"); therefore, we are going to ignore this special condition and move on.

memory area. That is, the shared memory area starts on the same virtual address in each process virtual space.

Other UNIX systems may map the shared memory segment into a different starting virtual address in each process. In general, UNIX systems only guarantee that the *displacement* into the shared memory area is the same for each process -- not the starting address. This means that any pointers to data items within the shared memory must be self-relative or must be displacements from the start of the area. They cannot be real pointers. In general, in order to write a program which uses shared memory in a portable manner across multiple implementations of UNIX, you should expect that the shared memory area will be mapped into a different contiguous virtual address range in each process which uses it.

Q4:

Q4 is shared read-only among all processes. It is the portion of HP-UX that is visible to user programs (e.g.: The gateway page that provides entrances to all system services.)

Threads

Let's look at a typical thread. A thread is a virtual processor with a private stack and private thread-local storage. The register contents of the thread, the thread's stack, and any thread-local storage are never shared with another thread. When UNIX executes a thread, it loads all of the register values and other processor contents that belong to this thread into a real processor and runs the thread by jumping to the next instruction of the thread. Whenever UNIX suspends a thread, it copies all of the processor contents for that thread into (UNIX private) storage before UNIX leaves the thread. Essentially, the registers and other processor contents are private to the thread. They are never shared. Likewise, the thread's stack is always private to the thread. Data items on the stack are never shared with another thread. A thread may also own data items in thread-local storage. These items are also private to the thread and will never be shared with another thread.

Storage which is not shared

- The stack of each thread is always private to that thread.
- Thread-local storage is private to each thread
- The processor registers are private to each thread

Shared Memory Areas

A shared memory area is created as a global, named entity within the UNIX machine. Subsequently, each process that wants to use the shared memory area maps the area into the process' virtual memory. When each process is finished using the shared memory area, the process removes the mapping from the process' virtual address. When the application is completely finished with the shared memory area, it is removed from the system (destroyed). Removing the shared memory area from the UNIX system is a separate operation from unmapping it. In the case of a shared memory segment this is a call to shmctl(). In the case of a memory-mapped file, the file is deleted using the standard file system mechanisms.

It is quite normal for programs within an application to expect that their shared memory areas just exist under all normal circumstances. Programs within the application simply assume that the shared memory exists, map it into their data area and begin using it. The one-time creation of the shared memory area and the (possibly never) deletion of the shared memory area are handled as exceptions to the norm by special programs.

The next two sections describe the system calls used for each type of shared memory area.

Creating and Using a Shared Memory Segment

This section uses the description of each system service call as the title for the discussion of how that call is used. The system services are presented in the order in which they would normally be used in the application.

#include sys/types.h
#include sys/ipc.h
#include sys/shm.h
The application using a shared memory segment should include these system headers in all programs.

key_t ftok(char* path, int id)
In order to create a shared memory segment or to access one that has already been created, we first need a key that we can use as the name of the segment. The key must be unique within the system and UNIX has its own idea of what a key should look like. The system call ftok() creates and returns a key based on the path to a file and an integer. This file will *not* be read nor written, and its contents are *not* what is mapped into the virtual memory -- ftok only needs the character string of the path. However, the path must be to a valid file that the program is authorized to read. A common practice is to use the full pathname of the program executable for the primary program in the application. Note that the same path and id will always return the same key_t value; so we just need to decide which path and id all the programs will use. If an application uses several shared memory segments, it normally uses the same path for all of them and uses a different integer (id) for each segment.

int shmget(key_t key, int size, int shmflg);
The key is used as input to shmget() which returns an integer shared-memory-id (always positive). If the IPC_CREAT flag is set in shmflg (**shmflg | IPC_CREAT)** then a new shared memory segment is created and its shared-memory-id is returned. If IPC_CREAT is clear, the segment associated with this key must already exist in the system and the shared-memory-id of the existing segment is returned. On failure shmget() returns (-1)

and **errno** provides the details of the error. On success, we have a positive integer shared-memory-id.

void* shmat(int shmid, void* shmaddr, int shmflg)
This system call maps the shared memory segment identified by the shared-memory-id (**shmid**) into the data area of the calling process and returns the starting address of the segment. **shmaddr** is normally specified as zero in order to allow the system complete freedom to select the appropriate address. The flags in **shmflg** allow mapping the segment as a read-only area, or as read/write. The address returned by shmat() is the base pointer used for all access to the shared memory segment.

At this point, the program is ready to use the shared memory segment.

In an HP-UX system, the virtual address returned to any process will always be the same for any given segment. That is, a segment will have the same virtual address in all processes that map it. This means that it is possible to use absolute pointers among the data items within the shared memory segment. Note that this is not the case in other UNIX systems.

int shmdt(char* shmaddr)
This call is normally used just before each program exits to delete the mapping of the segment from the process' virtual address space. Even if all processes delete the mapping, the shared memory segment is retained by UNIX until it is removed via shmctl(IPC_RMID) or the **ipcrm** command. This makes it possible for a subsequent execution of the application to map the shared memory segment and begin using the shared data items within it -- just as they were left when the segment was unmapped.

shmctl(int shmid, int cmd, int, struct shmid_ds* buf)
This function provides a wide variety of control options for the shared memory segment. The specific operation is determined by the value of **cmd**:

IPC_STAT:	Copy the information structure associated with this shared memory id into the structure pointed-to by buf.
IPC_SET:	Set the userid, groupid or mode of the shared memory segment. Normally you must be superuser or the creator of the segment to change this.
IPC_RMID:	Remove the shared memory identifier from the system after all processes that are currently mapping this segment have released it. The resources will be reclaimed and the data in the memory segment will be lost.

Normally the last program of an application session will use **IPC_RMID** to remove the shared memory from the system.

Commands for Manipulating Shared Memory Segments

UNIX provides two commands for manipulating shared memory segments. The **ipcs** command reports on shared memory segments. The **ipcrm** command removes a message queue, a semaphore set, or a shared memory segment from the system. These commands can be used interactively or can be placed into a shell script as part of the application.

Creating and Using a Memory Mapped File

This section uses the description of each system service call as the title for the discussion of how that call is used. The system services are presented in the order in which they would normally be used in the application.

There is nothing special about the creation of a memory mapped file -- only in how an application uses it. Almost any UNIX disk file can be memory mapped. A simple program to create and initialize a memory mapped file would open the file, ask how big it should be, write that many zero bytes into the file using write(), and close the file.

#include sys/types.h
#include sys/ipc.h
#include sys/mman.h
Each program should include these headers which define the function prototypes and data typedefs.

open(char* path, int oflag)
The program opens the file to be memory mapped using the standard open() system call. This returns an integer file descriptor for the file.

caddr_t mmap(caddr_t addr, size_t len, int prot, int flags, int fd, off_t off)
The integer file descriptor is used as the **fd** input to mmap(). This system call returns the address where the shared memory area begins. The program assigns this to a void pointer which is used as the base pointer for all accesses to the shared memory area. Input parameters to mmap() are as follows:

addr: The requested virtual address within the process to map the file. This should normally be zero to allow UNIX complete flexibility in selecting an appropriate virtual address within your process.

len: The requested length (in bytes) to map. This is normally a well-known value within the application. The size of the file can be obtained with standard directory calls or the (**ls -l**) command.

prot: PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE

flags:

MAP_SHARED	Writes will change the common copy
MAP_PRIVATE	Writes will create a private page (common copy not chgd)

fd: File descriptor of the file to map. It must be currently open for read (if PROT_READ is set) and open for write (if PROT_WRITE is set).

off: Offset into the file (byte displacement) which will be mapped to the starting address of the shared memory area. The range (off, off+len) will be mapped into (start, start+len) where *start* is the returned starting address of the shared memory area.

The program is now ready to use the memory-mapped file.

munmap(caddr_t addr, int len)
This system call unmaps a range of addresses from the process. Normally a program doesn't bother with unmapping the memory-mapped file. It just stops using the shared memory area. When the program exits, the mapping is released.

Intermediate Summary

OK, to summarize, we have a couple of different types of shared memory areas -- shared memory segments and memory-mapped files. After either type is created and mapped into one or more processes, the contents of the shared area are accessed just like any other area of memory. Because we allowed UNIX to select the starting address of the shared memory area, all of the accesses to it will be handled through a base pointer which was returned to the program by the UNIX system call.

The shared memory area has a name (either a key or a filename) to identify it uniquely. When any given shared memory area is mapped into multiple processes, all processes access the same physical copy of the shared memory area. Updates made by one process are seen immediately by all of the other processes.

Shared Data Items

Now that we understand how the shared memory area is mapped into the various processes which are sharing it; let's look at how to organize the data items within the shared memory area. Static allocation of the data items is the easier to program and provides a lot of functionality in a straightforward manner. Dynamic allocation of data within the shared memory segment provides substantially more flexibility at a substantially higher complexity of programming.

Static Data Allocation

Static data allocation is simply using a fixed layout of the shared memory area. The easiest way to do this is in "C" is to declare a structure that describes all of the data within the shared segment. The pointer to the start of the shared segment is declared to be a pointer to this type of structure. All data items in the shared segment are data items in the structure, and all are accessed through the structure and the base pointer to it.

Case 1 -- Example: Web Server CGI Programs

Static allocation is particularly useful in situations where one process is the writer and many other processes are the readers.   When only one thread updates the shared area, and when there are no complex organizations (e.g.: linked lists, hash tables) it is rarely necessary to lock the data items before using them. The writer simply stores the values into the data items and the readers read the values whenever they want them. Mapping a shared memory area is fast, so the CGI programs simply map the shared memory, send the data to the client, and exit.

Case 2 -- Example: Image Processing Application

A second typical use is where two or more processes operate on a single large data array but only one process uses it at a time. An example would be an image-processing application where the programs are small in relation to the size of the digital image being manipulated. A control program creates the shared segment, loads the picture, and then runs a series of image manipulation programs -- each of which maps the shared segment into itself, operates on the image, and exits.

Case 3 -- Communications or Database Server Application

A third example is the situation of a pair of processes, each of which uses a portion of the shared area to send messages to the other process. This is very effective when the messages are so large that copying them through a pipe would affect performance. In this example a single process acts as the server for other programs in other processes -- synchronizing updates to a central database or managing telecommunications for all of the programs.   The server copies the data directly from the database or communications line directly to the shared memory segment for the receiving application program.

Synchronization between processes

In some applications, it may not be necessary to provide any synchronization among the processes using a shared memory segment. In the case 1 above, each "reader" may be an HTTP process running a CGI script within a web server. When the HTTP process runs, it maps the shared memory segment and retrieves the latest information to send to the client. Whenever the information changes, some process running a program from a local login or telnet session simply changes the data item in the shared memory segment. The next time an HTTP process runs, it will retrieve the updated data. In this case, there is no synchronization whatsoever between reader and writer. Note that the HTTP processes didn't even exist when the shared memory area was created.

In other applications using shared memory segments it is necessary for one process to be able to "tap the other process on the shoulder". In case 2 above, the control process simply runs one child at a time and waits for the child to exit before running the next one. The program in the child process maps the shared memory segment, updates it, and exits. The UNIX-provided "child is finished" interrupt is sufficient communication between the processes.

In case 3 above, one process places the (large) message into its portion of the shared memory segment and needs to notify the other process to retrieve the message. Although message queues and semaphores can be used for this purpose, I have found that the most simple and straightforward mechanism is a pair of UNIX-pipes between the processes. Initialize the pipes in non-buffering mode and use the **write()** system call to send a short message (a few bytes) as a command to the other process. If the processes are parent and child, these can be unnamed pipes. If the processes are not so closely related, then use sockets or named pipes instead. An additional advantage of using pipes is that a process can use the **select()** system call to manage it's input queue. This is very useful when one of the processes is actually a server talking to 20 or 30 other processes, with 20 or 30 shared memory segments and 20 or 30 pairs of pipes -- one pair to each of the other processes.

Dynamically-allocated Data

Applications which use dynamically allocated data within shared memory areas tend to be the more complicated ones. Typically, there are multiple processes or multiple threads of execution simultaneously updating both the contents of the data items and their organization (the pointers among them).

Case 4 -- Example: Multi-Processor Number Grinder

We can have a multi-threaded application where several threads place service requests into a linked-list that is used by another thread as an input queue. We now have multiple threads (possibly in multiple processes) allocating and filling each linked-list item (the contents) and then adding it to the linked-list (the organization) -- with another process removing the items from the linked-list, processing the requests, and releasing the linked-list item. This is a very low-overhead method for synchronizing multiple processors.

Suppose that the application is a three-dimensional simulation where the program divides the total job into lots of subcubes and assigns a thread to each subcube. Because of data differences among the subcubes, some threads will finish before others, place their results onto the input queues of other threads, read their own input queue and press on -- allowing each thread to work continuously without waiting for the others.

Case 5 -- Example: Object Oriented Database

Another common scenario is a complex organization of structures that contain pointers to other structures -- for example, an in-memory, object-oriented database. This can be implemented in "C" by defining each structure type in a header file, dynamically allocating a place in the shared data area for each structure instance, and then using standard "C" expressions to manipulate the in-memory data items, to create pointers among the structure instances, and to follow the pointers from structure to structure.

These examples demonstrate three areas that are somewhat unique to dynamically allocated data items -- pointers among data items, functions to allocate and free portions of the shared segment, and locks to synchronize access among multiple threads.

Pointers:

In the general case, pointers to data items in a shared memory area or a memory-mapped file require special consideration because the starting address of the shared memory or memory-mapped file will not always be the same. When the starting address of the shared area changes, the addresses of all the data items within that shared area change also. This can happen several ways -- even in HPUX where each shared-memory segment is always mapped to the same virtual address in each process that simultaneously uses that segment.

1. A memory-mapped file is stored on disk between uses. Each time the file is mapped into virtual memory the starting address may be different.

For example: the virtual address range that maps the file today may be unavailable when the file is first mapped tomorrow.

2. Some versions of UNIX will map a shared memory segment at different virtual addresses in different processes even when the processes are using the segment simultaneously.

In order to handle different starting addresses for a shared data area, we don't store the actual address of the data item in the pointer. We store a relative value and add a correction each time the pointer is used.   There are two common ways to do this: self-relative "pointers" and displacements from the start of the shared area.

Self-relative "Pointers"

Self-relative "pointers" are long integers that store the difference between the address of the data item and the address of the pointer itself. Note that this difference may be positive or negative so the integer needs to be signed. Regardless of where the shared memory area starts, the distance between the "pointer" and the data item within the area doesn't change. Whenever the pointer is used, we simply add the address of the "pointer" to the value stored in the "pointer". Cast the address to (char*) to do the add and then cast the sum to be a pointer to the correct data type.

Displacements

The second common method is for the "pointer" to hold the displacement of the data item from the start of the shared memory area. Every time the "pointer" is stored, we subtract the starting address of the shared memory area -- and then add it back each time the "pointer" is used (with appropriate casts to (char*) and to the data type).

A variation on the displacement method relies on the data items being aligned on their natural boundaries. Data items in the shared area are referenced by their index from the starting address. Multiple base pointers -- one for each data type that may be stored in the shared area -- point to the start of the shared area. Each of these base pointers points to the same address -- the start of the data area, which normally is aligned on a page boundary.   Since we know the data type of each item in the shared area, each reference to it is merely its index from the base pointer of the correct data type.

Functions to Allocate and Free Portions of the Shared Area

In any shared memory scenario that dynamically allocates and frees portions of the area, you must provide your own functions for this purpose. The standard MALLOC() and FREE() don't handle shared memory segments. Additionally, it normally is impossible to expand the size of the shared memory segment (or the memory-mapped file) during the execution of your program. The scenario that works on all implementations is to start with a shared memory area larger than will ever be used, map it all into a single contiguous range of virtual memory, and then provide your own **allocate()** and **deallocate()** functions to subdivide it.

Locks

When multiple threads are simultaneously updating the data items in a shared memory segment, we need a mechanism to keep one thread from running over another one. An example from everyday life is a traffic intersection (let's say that both streets are one-way). Only one car at a time can use the intersection. If two cars try to cross at the same time, we have a collision. This is undesirable, so we install a traffic light, a mechanism to ensure that only one car goes through at any given time -- even if two cars arrive at the same time.

Locks in a computer are similar to the traffic light. Even if two threads arrive at the lock at the same time on different processors, only one thread is allowed through and the other thread is forced to wait. Locks synchronize the use of those portions of the code where we must have only one thread executing at a time. For example: updating the linkage pointers of a linked-list. There are several data items (the backward and forward pointers) which must all be updated together or we end-up with the problem of the "next" item in the list not pointing to the "previous" one. We cannot allow one thread to start updating the pointers and simultaneously have another thread trying to update them differently. In order to synchronize the threads through this "critical section" of code we create a lock to control access to the code section. Each thread "acquires" the lock before beginning its updates and "releases" the lock after it finishes. While the lock is held (acquired) by any thread, any other thread which tries to acquire that lock is forced to wait until the first thread releases the lock.

At the lowest level, locks are implemented by the hardware of the computer system. In UNIX, they are abstracted as semaphores, which are manipulated with the following system calls:

- **semget()** to create the semaphore.

- **semctl()** to initialize the semaphore to "unlocked"
- **semop()** to lock and unlock the semaphore

See the man pages for the exact format and parameters of these system calls. Essentially the semaphore is created using **semget()** and then initialized to a value of 1 using **semctl()**. To acquire the lock, each thread calls **semop()** to decrement the value of the semaphore. If the semaphore value is 1, the value is decremented to zero and the thread proceeds. If the semaphore is already 0, the thread is put to sleep until the semaphore goes positive. To release its lock, the thread calls **semop()** to increment the value. When the value goes positive, any threads sleeping (waiting for this semaphore) are awakened, allowed to compete for the semaphore. One thread wins and the others go back to sleep.

Locks are often thought-of as being assigned to particular data items -- such as to an instance of a structure. In order to update the contents of an instance of a structure, the code acquires the lock for that instance, updates the contents (as many of the data items that it wants to), and releases the lock. In this manner, all of the updates to that structure instance are consistent with one another.

A less obvious example is a counter. Acquire the counter's lock, increment the counter, and release the lock. The reason is that the action of incrementing the counter requires multiple instructions in the processor (load value into a register, add 1, store value). With a counter unprotected by a lock, if an interrupt occurs between the load and the store, another thread could update the counter. When the interrupted thread resumes, it then stores its (now stale) value into the counter -- resulting in the counter value erroneously reflecting only a single increment.

When locks are used with a complex organization of structures such as a linked list, or trees of structures containing pointers to other structures; it is generally best to assign separate locks to protect the navigation (usually one lock for all the pointers) and then an individual lock per structure instance to protect the contents of that structure (that instance). In order to update the contents of a structure instance, the code acquires the navigation lock; follows the pointers until it arrives at the structure of interest; locks the contents lock for that instance; unlocks the navigation lock; updates the structure contents; then unlocks the contents lock.

Summary

Shared memory segments and memory-mapped files are standard services provided by UNIX systems. Each of these two types of shared memory allows multiple processes to access the same physical memory pages.

Shared memory areas provide a very high-speed method for communicating among multiple programs and multiple processes -- communication at the speed of the processor itself. When one process stores data in the shared memory area, a reference from any of the processes retrieves the updated value. Shared memory allows applications to be organized as multiple programs -- with the obvious advantages to implementation, enhancements, and ongoing maintenance.

Within a shared memory area, the data items can be allocated either as a static structure where only the contents (the values) change; or the data items can be dynamically allocated as needed -- connected in real time into complex organizations. Static allocation provides simplicity of programming, while dynamic allocation provides ultimate flexibility to handle real-time changes. The programmer must supply the allocation and deallocation functions to subdivide the shared memory area.

Shared memory areas will not always begin at the same virtual address, so pointers in shared data should be self-relative or be displacements from the start of the area. Where multiple threads are updating the same shared data item, the data item should be protected with a lock to ensure that each set of updates is consistent.