AN IMPLEMENTATION OF SHARED MEMORY FOR UNIX WITH REAL-TIME SYNCHRONIZATION

by

Paul K. Harter, Jr. and Gregory R. Bollendonk

CU-CS-310-85 September, 1985

University of Colorado, Department of Computer Science, Boulder, Colorado.

An Implementation of Shared Memory for Unix with Real-Time Synchronization

Paul K. Harter, Jr.
Gregory R. Bollendonk

1. Introduction

The Distributed Computing Support system (DCS) [Harter 85b] is designed to provide both high-level language and operating systems support for the programmer building parallel, distributed applications. The above reference specifies the DCS user interface and describes the design of the DCS system. In particular, it discusses the abstraction of a distributed shared variable, and the required kernel support for shared memory. This paper describes the kernel implementation of shared memory upon which distributed shared memory rests.

Although the actual design and coding of the shared memory extension was done in the context of the DCS system, it had its beginnings as a class project for an advanced course in operating systems in the Fall Semester 1984. The purpose of the course was to provide students the opportunity to make a detailed study of the implementation of a "real" operating system, Unix. The term project for the course was to specify a useful extension to Unix and to suggest a possible implementation. One of the groups suggested the addition of shared memory. Their project provided the initial framework for our final design and implementation.

During the Spring Semester 1985, the Distributed Computing Support system was conceived and designed. Much of the low-level design and implementation of DCS was carried out as the class project for a graduate seminar in networking and distributed computing being taught by the first author.

The DCS system was to support distributed computations via asynchronous remote procedure calls and distributed shared variables. Distributed shared variables were to allow multiple processes, cooperating in a given computation, to share "global" variables, with access to these variables to be independent of process location. Sharing by processes executing on separate nodes would have to be be based on a message protocol. While processes executing on the same node could use the same protocols, it was clear that they would be able to share these variables directly at much lower cost through shared memory. Thus the design of DCS provided the impetus for extending the original shared memory proposal and carrying through with its implementation.

The implementation was originally conceived for a DEC VAX 11/780 running Unix 4.2BSD and first brought up under SUN Unix version 1.3 running on SUN workstations. We have recently added our shared memory implementation to the newest version of the operating system (SUN Unix version 2.0) with only minor modification. In fact, the installation took less than one man-day.

The remainder of this report is organized as follows. Section 2 discusses the background of interprocess communication (IPC) in Unix and the requirements introduced by the DCS system. Section 3 describes the shared memory facility provided by our extension, while Section 4 discusses the kernel implementation. Section 5 discusses performance issues, and Section 6 summarizes our results and conclusions.

2. Background

The original design of Unix [Ritchie 74, Ritchie 78] contained very little support for communication between processes. There was no shared memory, no general message facility, and no general mechanism for interprocess synchronization. There were two IPC channels available, pipes and signals.

A *pipe* provides a one-way, FIFO, byte stream between a pair of related processes accessed via the standard Unix I/O calls. Processes are delayed upon trying to **read** from an empty pipe or **write** to a full one. The advantage to pipes is that they are well tailored to a type of processing very common to Unix systems. *Signals* provide a simple form of software interrupt facility. A signal is sent from one process to another and typically results in the asynchronous invocation of a *handler* routine in the receiving process. Signals are useful for fairly simple, pre-arranged synchronization between processes.

The greatest disadvantage to the two mechanisms described above is their lack of generality. In particular, both are limited to use between *related* processes. For pipes, this relation is kinship via **fork**'s from the same parent, while for signals it is determined by association with the same terminal. This precludes, for example, the dynamic creation of pipes between existing processes desiring to communicate. In the case of signals, the number and meanings of signals are fixed and determined by the system, and a signals can carry no information. Thus, a signal may not even carry the identity of the sender, which places a very strict limit on flexibility.

Arbitrary process pairs may communicate via the file system by opening the same file, but this has two drawbacks. First, speed is limited by the file system speed, even for small messages, and second, without a general synchronization mechanism, coordination of access to the shared file is difficult. Some Unix programs use the existence or non-existence of a known file to simulate a binary semaphore, but this is clearly sub-optimal.

2.1. IPC in Berkeley Unix 4.2

Later versions of Unix have contained extensions to allow more general forms of IPC. ATT extended Unix to allow for shared memory segments, first in a non-paged and then in a paged environment. In addition to shared memory segments, ATT Unix has semaphores and message queues. All three are implemented as Unix objects similar to files and subject to standard Unix protection checking on access. While our implementation is completely independent of the file system, it has some features in common with ATT's System V.

The berkeley unix group took a different approach and added a completely general IPC facility, which included support for networking and internetworking. This system provides a new abstraction called a socket, which is a port for sending and receiving messages. Sockets are handled within a program much as file descriptors have been. In fact, the standard system **read** and **write** calls work transparently on some classes of sockets. Differentiating sockets and file descriptors are network addresses and additional system calls to associate addresses with sockets (binding) and to associate sockets in different processes with each other (connection establishment). As an alternative to the (semi-permanent) association of two sockets implied by a connection, individual messages may be addressed to specific destinations. A last and important distinction between files an sockets is that sockets are not sharable objects and thus have no need for sets of permissions such as files have. Any process can use its socket to send a message to any address.

2.2. DCS Requirements

With the ever increasing availability of powerful workstations, there has been a rapid increase in the number of computing environments made up of personal workstations connected via a local area network. Since each workstation is dedicated to an individual who is likely to spend most of his time typing (eg. word processing, program editing), the workstations will be grossly under-utilized most of the time. The DCS system is an attempt to harness this surplus power to do useful computing.

The concurrent availability of a number of under-utilized workstations opens up many new possibilities if the physical parallelism inherent in separate CPU's can be exploited. Due to the relative high cost of communication between machines [Popek 81, Peterson 79] when compared to memory access and instruction execution, use of a local area network for distributed parallel computation must be restricted to algorithms whose demand for cycles greatly exceed their need for interprocess communication. Schnabel [Schnabel 85] is investigating a class of numerical algorithms that appear ideally suited for this environment.

As mentioned in the Introduction, the Distributed Computing Support (DCS) system supplies language support for distributed computing via a collection of systems routines. The user abstractions provided by DCS are the

asynchronous remote procedure call and distributed shared variable. While these are described in detail elsewhere [Harter 85b], a brief description will be given here to motivate the implementation of shared memory.

An asynchronous remote procedure call is like a normal procedure call in that input and output arguments are passed and returned. It may or may not actually execute remotely, so the programmer may make no assumptions as to the execution site, although we plan to investigate the feasibility of allowing some programmer input on location in the future. The difference is that rather than the normal procedure call synchronization, wherein the caller is suspended during execution of the procedure, in the asynchronous call the caller continues to execute. The caller may then either merely continue to compute or make more asynchronous calls. Resynchronization of the caller and the callee occurs at the callers discretion, when he may request that he be suspended pending completion of outstanding calls.

In the case of the synchronous procedure call, the called procedure may access global data declared in some surrounding scope. The utility of this type of access prompted us to attempt to provide a similar type of access in the asynchronous case. The programmers view of a distributed shared variable is that of a variable that may be read, written, or updated atomically with respect to other processes, where an update has the form:

var := user_function (var, arg),

with the constraint that *arg* may not contain a direct reference to another shared variable. Thus, although the programmer must be aware of the fact that the variable may change between subsequent reads by a single process, the variable is always in a consistent state, independent of size or the locations of sharing processes.

The user of the DCS abstractions need not be concerned with networks, addresses, ports (or sockets), or the extra system interfaces for sending and receiving messages. However their implementation placed a number of requirements on its host systems. First, it required a general message passing facility to allow for the distribution of processes, arguments and results, and for the sharing of data in a distributed environment. All of our machines currently run Berkeley Unix 4.2 and the IPC implementation there could meet our needs for message passing in the network.

The implementation of distributed shared variables posed additional problems. The distributed shared variables associated with a computation must be equally accessible to all processes of that computation independent of their physical site of execution. This desire for equal, efficient access excluded excluded an implementation where processes obtained and assigned values by sending messages to a special storage or caretaker node. Thus, a copy of each variable is kept at each node. Further, the various copies of a shared variable must be kept consistent across nodes. This is ensured in DCS by a collection of manager

processes (Distributed Shared Memory Processes, DSMP's), one on each node. The DSMP's on the various nodes communicate update information to each other and guarantee adherence to the update discipline specified for each variable. Thus, the DSMP must also be able to access the shared data.

Again, our desire for efficiency caused us to reject a solution, where the DSMP "owned" the copy of the shared data on each node and responded to requests by the various processes on that node. Particularly in the case of multiple concurrent computations or multiple processes of a single computation on one node, the DSMP could present a performance bottleneck to in the system. Thus, to avoid the cost of sending messages within a node and the bottleneck effect of having one process do so much work, it is necessary for processes to be able to access the same set of memory locations. This feature was not available in Berkeley Unix, so we decided to add it.

Finally, there are synchronization requirements. The design of the DCS interface requires that any access to variables must be atomic. That is, it must be possible to read or write a variable atomically. There is no problem with this in communication between nodes to keep variables up to date, since the update protocols have been designed to send and install entire variables at once. A more difficult problem arises within a node, where the DSMP and one or more user processes all share a single copy of a shared variable. Since these variables may be larger than the width of the memory data path, atomic access is not guaranteed by instruction atomicity for reads and writes, let alone updates. This is an instance of the standard mutual exclusion problem. In our case, however, there are real-time constraints imposed by the role of the DSMP, who must maintain many variables simultaneously.

In adding shared memory to Unix, we could also add semaphores [Dijkstra 68a] or some other mechanism for mutual exclusion, but the standard semantics of semaphores is inadequate. Since the DSMP must acquire exclusive access to install changes from other nodes in the system, it would have to do a **P** on a semaphore that could be held by a user process. Since the user process may fail or terminate while holding the semaphore, the DSMP would be stuck forever. Even barring the error case, a user process could hold the semaphore over a page fault or disk read. Since the DSMP may be serving many processes and many shared variables, this wasted time could lead to significant performance degradation. This led us to the design of a semaphore that could be used in the face of performance constraints. The semaphores we included (see Section 3 and the Appendix) provide a timeout period so that a **P** operation will return after the semaphore has been decremented or the timeout period has passed. The timeout can be specified as having zero (for polling), infinite or some finite length.

3. Supplied Features

In this section we describe the interface and features included in our implementation as motivated in the previous section. The Unix manual pages for the new system routines are contained in the Appendix. First we describe the interface for attaching to and using shared segments, and then the declaration and use of semaphores.

3.1. Shared Segments

Our implementation provides shared memory segments that are mapped indistinguishably into the memory space of processes. Ordinary reading and writing of variables in shared memory occur in the same way as access to variables in private data space, i.e. there are no special access functions or system overhead involved. Shared segments may exist anywhere within the address space of a process and the number of shared segments that may be accessed by a process at one time is limited only by system table space. Our implementation currently enforces a system-wide limit of 20 segments of up to 20 pages existing concurrently, however these limits may be changed simply by modifying manifest constants and recompiling the affected modules. The only restriction on the location of shared segments is that they begin on page boundaries and occupy an integral number of pages. Our implementation of shared data segments requires two new system calls, **vshare**, and **vrlse**. These new calls allow the user process to create, attach, and detach itself from shared segments within the system.

3.1.1. Creation and Rendezvous

In order to share memory among a set of processes, one of the processes must "create" a shared segment by declaring a piece of its address space sharable. This is done by informing the system that it wishes to share a segment of its address space via a call to the routine **vshare** giving the start address and size of the space to share, and a null segment identifier *seg_id*. The kernel returns a unique (within the machine) identifier which is used to refer to the shared segment in the future.

Other processes in the set may then attach to the segment just created via the same system call. A process wishing to attach to a shared segment must do so by obtaining the *seg_id* for the segment and "trading in" a piece of its address space for a previously declared shared segment. Thus, prior to sharing memory there must be some initial communication between the sharing processes to exchange the *seg_id*, for example via a pre-arranged socket port name or file name. Having obtained the *seg_id*, the segment is mapped via a call to **vshare** giving the *seg_id* of the shared segment and start address and size of the address space to be traded. When this call returns, both processes have the shared segment mapped into their address spaces, though possibly at different logical addresses.

While attached to a shared segment, a process may neither **fork** a new copy of itself, nor may it **exec** a new text image. Although it would be possible to design and implement a reasonable semantics for the result of a **fork**, it was not required for our purposes. Since the kernel code implementing the **fork** operation is rather complex with interfaces throughout the system, we chose not to support it. The **exec** call is also complicated, but in this case it is hard to imagine integrating the semantics of **exec** with those of shared memory. An **exec** system call overwrites the address space of the calling process with the text and data segments of the program being **exec**'ed, expanding the address space if necessary. This causes several problems. First, since the address space of the process is likely to change shape across the **exec**, it may well be that the shared segment would be overwritten by the code of the new program. This is not likely to be the desired effect, and avoiding it would require the programmer to worry about object code sizes and other details. Second, since all data space of the process calling **exec** is reinitialized, the process would not be able to "remember" where the shared segment began. Thus, it seems that there isn't a clearly correct way to implement this feature at all. Finally, if the effect of a **fork** or an **exec** in the presence of shared memory is desired, it can be obtained by releasing the segment, making the call, and reattaching to the segment afterwards.

A process attached to a shared segment may detach itself from that segment explicitly by calling the routine **vrlse** giving the *seg_id* of the segment it wishes to detach. On the other hand, a process may be detached implicitly. When a process terminates, it is detached from all shared segments to which it is attached before the system goes through the standard termination processing.

In either case, when a process is detached from a shared segment, there are two possibilities. If the process being detached is the only process attached to the shared segment, then the segment becomes unsharable and the calling process continues with the data in the segment unchanged. On the other hand, if there are several processes attached to the shared segment at the time of the call, then the system replaces the shared segment in the calling process with new zero-filled pages.

3.1.2. Ownership

Shared segments are shared equally, i.e. there is no "owner" of a shared segment. Although the shared segment is initially part of the private address space of the process that first declares it sharable, that process has no special rights to the segment afterward. A process may attach to the shared segment by knowing its size and *seg_id*. Though this is no protection from malicious processes, the likelyhood of guessing both correctly by accident is small. Thus, there are no privilege classes or special access rights for the segment that must be checked prior to granting access to the requested segment.

The process that "creates" the segment may very well not be the last one to access it, and the lifetime of a shared segment is not limited by the lifetime of the creator. Processes may come and go, but the segment remains sharable until every process that attached the segment has either released it explicitly or terminated.

3.2. Semaphores

Our implementation of semaphores is a natural extension to our shared data segments, and semaphores are intended to coordinate access to the shared objects contained in these segments. A shared data segment may have a number of semaphores, each associated with a particular offset within the segment. Note: The semaphores are *associated* with offsets in segments, not located in the segments, so a user process may only access a semaphore via the kernel operations supplied. Thus, while use of semaphores to coordinate access is not required, they can not be overwritten by accident.

Semaphores provide an efficient resource control mechanism for a user process to synchronize access with any other process, with minimal kernel overhead. Our implementation of semaphores requires three new system calls: **getsem**, **Psem**, and **Vsem**, for the creation and use of semaphores. Again, the number of semaphores is limited only by system table space, which is easily modified.

3.2.1. Creation and Use

Since semaphores are associated with offsets within shared segments, a user process must be attached to a sharable segment in order to create or use a semaphore. A semaphore is created via a call to the kernel routine **getsem**, specifying a shared segment, an offset, and an initial value. The system returns a unique (within the machine) identifier *sem_id* (semaphore identifier), which is used to refer to the semaphore in the future. Subsequent calls to **getsem** specifying the same segment and offset location will return the same sem_id and have no effect on the semaphore value. Thus, two processes sharing a segment need not agree in advance on which is to create semaphores. If two processes attempt to create and then decrement the same semaphore simultaneously, then one process will create it successfully and exactly one process will successfully complete the **P** operation (assuming an initial value of 1). There are no guarantees as to the identity of either.

Once a process has the sem_id for an associated semaphore, it can operate on the semaphore with the new system calls **Psem** and **Vsem**, **P** and **V** respectively, although the semantics of our calls do not exactly match those of in their pure forms. First, in order to facilitate the use of semaphores for resource counting (number of shared buffer slots etc.), we have implemented the so-called PV-chunk operations [Vantilborgh 72] to reduce the likelihood of deadlock. Thus, a semaphore may be incremented or decremented by an integer value

specified as a parameter to the system call.

As mentioned in Section 2.2 on the requirements imposed by the DCS system, it must be possible to guarantee sychronization and mutual exclusion in a real-time environment. This implies that the standard semantics for the **P** operation is inadequate to our purposes, since it implies possible arbitrary delay of the caller. Therefore, our **Psem** system call allows the user process to specify the amount of time it wishes to wait while trying to decrement the value of the semaphore. A return code indicates that the semaphore was successfully decremented (0) or that the call timed out (-1).

Thus, a process calling the kernel routine **Psem** specifies not only the semaphore (sem_id) but a decrement value (decr) and timeout value (time) as well, and the semantics is:

Psem(sem_id,decr,time) $\equiv$ suspend caller until (sem_id $\geq$ decr $\|$ time elapsed)
Then atomically execute:
If (sem_id $\geq$ decr) {
sem_id $-=$ decr; return (0) }
else return(-1)

There are two special case values for the timeout interval. A value of 0 specifies that **Psem** is to return immediately, even if the decrement cannot be performed. A value of -1 specifies that the call is not to return unless the semaphore is successfully decremented. Thus, the call **Psem**(sem_id, 1, -1) has the same semantics as Dijkstra's **P**(sem_id) operation.

The **Vsem** system call requires a sem_id and a semaphore increment value. If the semaphore value is incremented enough to allow one or more waiting processes to proceed, then waiting processes are awakened in FIFO order. It is possible for a process waiting in a **Psem** operation with a high decrement value to be overtaken by a process with a low decrement value in the case where the **Vsem** increment was not great enough to satisfy the affected process.

Once created, a semaphore remains in existence as long as the segment with which it is associated exists. When the shared segment has been **vrlse**'d by all processes attached to it, the segment becomes unsharable and all semaphores associated with the segment are deleted from the system.

The five new system calls described above (**vshare**, **vrlse**, **getsem**, **Psem**, and **Vsem**) provide a general, efficient mechanism for data sharing and synchronization among any number of unrelated processes. Data access is indistinguishable from normal (private) access and no copying of information is required. The implementation of **Psem** and **Vsem** operations results in a synchronization mechanism involving very little kernel overhead, especially when compared to the use of "lock files." The efficiency of these operations will be further discussed in Section 5.

4. Implementation

The facilities introduced above have been implemented on the SUN workstation under the Sun Micro Systems version of Unix (for our purposes, a port of Berkeley Unix 4.2). The basic kernel environment for SUN Unix virtual memory includes a set of page tables for each process and a global, circularly linked list of page frame descriptors to implement a variation of the "clock" algorithm for memory management [Babaoglu 81]. Each page frame descriptor references the user page table entry for the page it contains. In addition, the SUN workstation has a separate memory mapping module, which translates the addresses generated by the CPU. The page tables of the currently executing process must be loaded from main memory into the memory map unit for translation to take place.

Our approach to shared memory was to implement the simplest scheme consistent with reasonable performance. Two processes sharing a segment have identical page table entries for the ranges of addresses corresponding to the shared segment. Semaphores exist only in the kernel, and may be referenced only in the **Psem** and **Vsem** calls via the sem_id's returned from the **getsem** system call.

Although the user interface for shared segments does not include the notion of a distinguished "owner process" for a shared segment, it is necessary to make such a distinction at the implementation level. Thus, in the following, we will assume that each shared segment has a current *owner* and zero or more *subordinates*. The owner is the process that first calls **vshare** to make make a portion of its address space sharable a subordinate is any process that subsequently attaches to the segment. The (physical) page frames containing the (virtual) pages that are made sharable by the call will contain the shared segment throughout its existence. These frames are initially allocated to the owner process before it calls **vshare**, and continue to be allocated to that process afterwards. The reason is that the page frame descriptors for any allocated pages in the system must contain a reference to a user page table, and it seemed simplest to leave them allocated to the owner. Thus, the owner is the only process that has "physical memory" allocated for the shared segment. If the owner releases the segment, then ownership transfers to one of the subordinates (if any), and the frame descriptors are modified to point to its page tables.

We added two data structures to the kernel, a segment descriptor table (sd_map) and a semaphore descriptor list (sem_list). The user values seg_id and sem_id are indices into sd_map and sem_list respectively. Each sd_map entry contains the size and start address of the shared segment in the owner's address space, a pointer to the owner's process table entry, a list of processes sharing that segment, and finally the start index to a list of the semaphores associated with that segment (sem_list). Each sem_list entry contains the index of the next in the list, the associated offset within its segment, the semaphore's

current value and the first index in a list of processes waiting on that semaphore (wait_list), and a pointer back to the sd_map entry describing the segment with which the semaphore is associated. The wait_list is a simple linked list of delayed **Psem** operations, each indicating the delayed process and the value by which it desires to decrement the semaphore.

When **vshare** is called, it first screens the input parameter values for legality. If the seg_id parameter is "0", a new sd_map entry is created corresponding to the segment described by the call parameters, the calling process is marked unswappable, and the involved pages and page frames are locked in memory. If the seg_id parameter is non-0, then the size in the sd_map entry corresponding to the passed seg_id is compared to the passed size. If the sizes match, then the callers pages are returned to the system and the page table entries from the segment's owner are copied into the caller's page table.

Our decision to lock shared segments into memory can be traced to a number of factors. First, me desired that our extension have minimal impact on existing kernel routines and data structures to make it easy to add to other versions of the system in the future. Further, considering the complexity involved when compared to the actual performance gains, the positive return on our effort would have been minimal. If shared segments were not locked in memory, it would be necessary to modify the paging mechanism to keep page tables in several processes consistent when a page is paged out. Reference or modified bits would have to be copied from the process making the reference to the owner process (this information is referenced from the frame descriptor). These modifications would require another data structure in the kernel and one or more new fields in the kernel's process table. A mechanism similar to that currently used for shared text segments would have to be used, with the added complication that shared text segments may reside at different addresses in different processes.

The decision to lock shared pages and make their processes unswappable does not incur great performance penalties. First, processes are swapped in their entirety relatively infrequently in normal system execution, so making a process sharing memory unswappable does not involve a large cost. Second, shared memory segments involve pages being referenced by several processes, which are referenced comparatively frequently. By the nature of the clock paging algorithm used in the kernel, these frequently referenced pages would probably remain in core anyway. Thus, locking them in memory is unlikely to change their core residence patterns. To insure that this is the case, limits are set on the total number of pages that may be involved in shared segments and hence locked into core.

The **vrlse** call takes as its parameter the seg_id of a shared segment to which the caller is attached. After verifying that this is the case, one of three actions is taken. If the caller is a subordinate for the segment, then the callers page tables are modified so as not to refer to the shared segment and are

marked "fill on demand." This means that pages will be allocated to the process as those addresses are accessed. If the caller is the owner and there are existing subordinates, then the page frame descriptors are modified to point to the page tables of one of the subordinates, who thus becomes the new owner. The caller then gets "fill on demand" page table entries to replace the ones that referred to the shared segment. Finally, if the caller is the last process attached to the segment, the sd_map entry is deleted and the pages of the segment are unlocked from core. In all cases, if the process has just released its last shared segment, it is made swapable again.

When a process calls **getsem** with a seg_id, offset and initial value, it is first verified that the process is attached to the segment in question, that the offset is legal, and that the initial value is non-negative. If so, if there is not already a semaphore declared for that location, a new sem_list entry is allocated and initialized, and appended to the sem_list for the segment whose seg_id was passed. If the semaphore already exists, then its sem_list entry is located but not re-initialized. In both cases, the index (sem_id) of the entry is returned to the caller.

Psem first checks to see whether the caller is actually attached to the segment associated with the sem_id passed and whether the decrement value is positive. If so, it checks the value of the semaphore to determine whether a decrement is possible, i.e. whether the result would be non-negative. If so, the value is decremented and 0 is returned to indicate success. If not, the handling depends on the timeout value passed. If zero, the call returns immediately with -1 to indicate failure to decrement. If the value is -1, then a wait_list entry is allocated and appended to the current wait_list. Then the process calls the kernel **sleep** routine from within **Psem** using the address of the wait_list entry, and specifying a priority (PZERO) to prevent having to handle signals. Finally, for any positive time value, the caller allocates the wait_list entry as above, but before going to sleep, calls the kernel **timeout** routine saying that it is to be awakened in any case if the timeout period is exceeded. Then, when the process is awakened, it must determine whether it was awakened due to a timeout or as the result of a **Vsem** call. In the former case, **Psem** returns with an error code indicating a timeout has occurred. In the latter case, the routine **untimeout** will be called to cancel the previous request for a **wakeup**, and the call will return successfully after decrementing the semaphore.

Last, the **Vsem** call also checks for legality exactly as does the **Psem** routine. It then increments the value of the semaphore and scans the wait_list from front to back looking for processes who can now safely decrement the semaphore. For each such process, the address of the wait_list entry is passed to the kernel **wakeup** routine which makes the process executable again. When it has a chance to run, it will decrement the semaphore and return from the **Psem** call. A call to **Vsem** with legal parameters always returns successfully and never causes the caller to be delayed.

Four other kernel modules were modified to allow for shared memory. The modules for **fork** (kern_fork.c) and **exec** (kern_exec.c) were modified to test whether the caller is attached to shared memory and disallow the call if it is. The module handling process termination (kern_exit.c) was modified to test the terminating process for the existence of shared memory and to **vrlse** any shared segments. Finally, the system initialization routine (init_main.c) was modified to cause initialization of the shared memory data structures.

5. Performance

Shared memory provides a large speed improvement over the use of sockets for sharing of information among processes on the same machine. In this section, we give the results of some timing measurements made with our implementation.

First, the system calls to implement synchronized communication in shared segment are relatively efficient. The following table shows the times and number of instructions involved in a null system call, i.e. a system call with no kernel code executed beyond that for context switches, and our synchronization calls. These tests were run on an otherwise unloaded SUN 120 workstation. The parenthesized values in the instructions column give instruction counts normalized to the null system call.

Call	Time for 10,000	Time for 1	# of Instructions
null	3.5 sec	350 usec	292 (0 -> base)
Vsem	4.1 sec	410 usec	342 (50)
Psem	4.3 sec	430 usec	358 (66)
getsem	5.5 sec	550 usec	458 (166)

As can be seen from the above results, the **Psem** and **Vsem** calls are very fast. Most of the cost is in the checking of parameters (eg. verifying the existence of the semaphore and that the caller is attached to the segment with which it is associated), and the *timeout* and *untimeout* routines in the kernel, which are used for our implementation. The getsem call, which is executed only once for each semaphore, is somewhat more expensive, but still on the order of half the cost of the two context switches necessary to execute any system call.

Far more interesting than a simple listing of times for system calls is the data on a comparison of shared memory with sockets as a means of transferring data between processes. We set up two pairs of processes, one pair communicating via shared memory, the other via UNIX sockets. Each pair involved a reader process and a writer process. The writer was to transfer a series of blocks of various sizes to the reader process. Each size was transferred 100 times and the resulting times were then normalized to one transfer.

The shared memory implementation was essentially a one slot version of the well-known bounded buffer problem. The two processes had the following

outlines

```
Writer:     loop 100                Reader:     loop 100
                fill buffer                         P(full)
                V(full)                     read buffer
                P(empty)                            V(empty)
                end                                 end
```

Buffer sizes ranged from 128 bytes to 8192 bytes

Similar code was set up for the two processes communicating via sockets.

```
Writer:     loop 100                Reader:     loop 100
                fill buffer                         recv(buffer)
                send(buffer)                        read buffer
                recv(ACK)                           send(ACK)
                end                                 end
```

Buffer sizes ranged from 128 bytes to a maximum of 2048 bytes.
We have no data for stream sockets with buffers longer than
2048, as the implementation would not permit us to send longer buffers.

Below is a comparison of asynchronous block data communication transfer times between two processes using stream sockets versus shared memory with semaphores. Each read or write data transfer must wait for an acknowledge from the previous read or write operation.

Buffer Size (bytes)	Stream Socket (milliseconds)	Shared Memory (milliseconds)
128	19.5	3.3
256	24.6	3.6
512	38.0	3.8
1024	62.8	4.6
1536	93.2	5.5
2048	107.0	6.4
4096	no data	9.4
8192	no data	17.2

For this test, we used a simple data transfer, because it seemed the most straightforward and involved the fewest assumptions about program usage patterns. As a result, the timings are as favorable as possible to the socket implementation. One could easily imagine providing a pseudo-shared memory facility based on sockets, with one process acting as a memory server. This server process would handle all access to shared data. Thus, for a simple update, it would be necessary for a process to request and receive data, process it and create a new value, and finally send it back, all via sockets. For this case, the times above for sockets would essentially double. The shared memory

implementation would be somewhat faster than above, since the update would involve a single **Psem**, a single **Vsem**, and the transfer of only enough data to make the update.

6. Conclusion

We have implemented shared memory in a version of Unix running on a SUN work station within the context of a general facility for distributed programming. The facility includes shared memory segments for data sharing and semaphores for synchronization. The semaphore implementation includes a time-out facility to make it useful for coordinating access to shared objects in an environment with real-time performance constraints. The implementation involved minimal change to the existing kernel and resulted in data sharing far more efficient and general than previously available under Unix.

7. Acknowledgements

We owe a debt of gratitude to many people for this work, Dennis Heimbigner taught the advanced systems course in which it began, and added his insight to the design of the semaphore facility. Bob Gray, Keith Cowley, and Grant Rose were part of the initial project group, and Mike Schweitzer contributed to the final implementation done primarily by the second author. Finally, the presentation was much improved by Jon Shultis and Evi Nemeth who waded through earlier versions.

References

[Babaoglu 81] O. Babaoglu, W. Joy.
Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Referenced Bits.
Proceedings of the Eighth Symposium on Operating Systems Principles, (Asilomar, 1981), published as *Operating Systems Review 15* (5):78-86, (December 1981).

[Dijkstra 68a] E. W. Dijkstra.
Cooperating Sequential Processes.
In *Programming Languages,* (F. Genuys, Editor), Academic Press, New York, 1968.

[Harter 85b] P. K. Harter, Jr., P. Maybee.
DCS: A System for Distributed Computing Support.
University of Colorado, Computer Science TR #CU-CS-309-85

[Peterson 79] J. L. Peterson.
Notes on a workshop on distributed computing.
Operating Systems Review 13 (3):18-27, (July 1979).

[Popek 81] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, G. Thiel.
LOCUS: A Network Transparent, High Reliability, Distributed System.
Proceedings of the Eighth Symposium on Operating Systems Principles, (Asilomar, 1981), published as *Operating Systems Review 15* (5):169-177, (December 1981).

[Ritchie 74] D. M. Ritchie, K. Thompson.
The UNIX Time Sharing System.
Communications of the ACM 17 (7):365-375, (July 1974).

[Ritchie 78] D. M. Ritchie
The UNIX Time Sharing System: A Retrospective.
The Bell System Technical Journal 57 (6,Part 2):1947-1970, (July-August 1978).

[Schnabel 85] R. B. Schnabel.
Parallel Computing in Optimization.
Proceedings of the NATO Advanced Study Institute on Computational Mathematical Programming, (Klaus Schittkowski, Editor), Springer-Verlag, 1985.
Also Available as University of Colorado Technical Report CU-CS-282-84.

[Vantilborgh 72] H. Vantilborgh, A. Van Lamsweerde.
On an Extension of Dijkstra's Semaphroe Primitives.
Information Processing Letters, Volume 1, (1972), pp 54-79.

Appendix

The following pages contain the Unix manual pages for our shared memory system calls.

NAME

vshare – create sharable data segment, or attach process to previously created sharable data segment.

SYNOPSIS

```
status = vshare(&seg_id, size, addr)
int seg_id, size, status
char *addr
```

DESCRIPTION

Vshare will make a data segment within the calling process globally sharable to any other process within the system, or replace a segment from the calling process with a segment made sharable by a previous call (by another process). Prior to a *vshare* call, the process must contain a segment that is equal in size to the desired global shared data segment. *Size* must have granularity of NBPG (number of bytes per page), and *addr* must be on a page boundary. *Vshare* has two types of operations:

To create a sharable segment, the process provides *seg_id=0,* a valid *size,* and a pointer to the data segment that is to be shared, *addr.* This segment must be within the calling process' data space prior to the vshare call. Upon successful return, *seg_id* will contain the unique segment identifier of the created segment. This segment identifier may be used by other processes to attach to the segment. If the call was unsuccessful, then *seg_id* will be meaningless and *errno* will contain an error return value.

To attach to a previously created sharable data segment, the process must provide the unique segment identifier to the shared data space, *seg_id,* and the size of shared data segment, *size.* The identifier must be obtained from the original creator of the sequence (see above). The attaching process must also provide the virtual address *addr* of an equivalent data segment that is within its address space. This data segment will be released to the free memory pool, and the corresponding page table entries will be changed to point at the desired shared data segment. If the return value is 0, then the segment was successfully attached, otherwise it failed. Failure can be due to an invalid combination of *seg_id, size,* and *addr.*

ERRORS

vshare has a zero return value unless there has been an error, in which case the global value errno will be set as follows.

[ESRCH]	If *seg_id* not found (trying to attach to existing segment)
[ENOMEM]	If the shared data table is full
[EINVAL]	Bad *size* granularity, *addr* not in data space, or exceeds maximum size
[EALREADY]	If process is already attached to specified segment

EXAMPLE

```
seg_id = 0;             /* create new sharable segment */
size = 4096;            /* must have page granularity */
addr = x[0];            /* must have page boundary */
status = vshare(&seg_id, size, addr);
```

SEE ALSO

vrlse(2), getsem(2), Psem(2), Vsem(2)

AUTHOR

Greg Bollendonk, Grant Rose, Michael Schweitzer, Paul Harter

DIAGNOSTICS

When *vshare* returns a non-zero value, the global variable *errno* contains one of the above error codes.

BUGS

maximum segment size depends on size of maximum memory.

NAME

vrlse – release a virtual shared data segment

SYNOPSIS

status = vrlse(seg_id)
int seg_id, status

DESCRIPTION

Vrlse causes the shared data segment previously allocated to a process to be released. The segment is identified by *seg_id.*

If the calling process is the last process referencing the segment, then it is removed from the system, any semaphores declared within the segment are removed, and the calling process keeps the segment. If other processes are still attached to the segment, then the released segment is replaced by zero-filled pages in the address space of the calling process. The page table entries of the current process (u.u_procp) are unmapped from the shared segment, and the process is removed from the shared data map structure in the kernel.

ERRORS

Vrlse has a zero return value unless there has been an error, in which case the global value errno will be set as follows.

[EINVAL] If *seg_id* is out of range

[ESRCH] If *seg_id* is invalid for this process

EXAMPLE

```
seg_id = 22;            /* segment identifier returned from vshare */
status = vrlse(seg_id);
```

SEE ALSO

vshare(2), getsem(2), Psem(2), Vsem(2)

AUTHOR

Greg Bollendonk, Grant Rose, Michael Schweitzer, Paul Harter

DIAGNOSTICS

The global variable, errno, will be set if *vrlse* has a non-zero return code.

BUGS

None.

NAME

getsem – create a semaphore for a shared memory location

SYNOPSIS

sem_id = getsem(seg_id, offset, vinit)
int seg_id, offset, vinit, sem_id

DESCRIPTION

Getsem creates a semaphore associated with a given shared memory location referenced by the *seg_id, offset* pair and returns the unique identifier associated with it by *getsem.* The identifier *sem_id* must be used for all subsequent *Psem* and *Vsem* calls. The initial value *vinit,* is the initialized value of the semaphore. A call to *getsem* can have two results:

Getsem creates a new semaphore for the given *seg_id, offset* pair, assigns an initial value *vinit,* to it, and returns the identifier *sem_id.*

Getsem is called with the same *seg_id, offset* pair used in a previous call to create a semaphore. In this case *getsem* returns the *sem_id* associated with that semaphore and ignores the initial value passed.

The semaphore will be deleted when the segment whose *seg_id* was passed to create the semaphore is deleted. This will occur when the last process attached to the segment performs a *vrlse* on that segment.

ERRORS

Getsem will return a positive integer *sem_id* unless there has been an error, in which case the global value errno will be set as follows.

[ESRCH]	If *seg_id* is not currently in the system
[EBADF]	If *seg_id* has invalid range
[ENOMEM]	If kernel semaphore free-list is empty
[EFAULT]	If *offset* is not in *seg_id*
[EINVAL]	If *vinit* < 0

EXAMPLE

```
seg_id = 5;              /* segment identifier returned from vshare */
offset = 320;
vinit = 1;
sem_id = getsem(seg_id, offset, vinit);
```

SEE ALSO

Psem(2), Vsem(2), vshare(2), vrlse(2)

AUTHOR

Greg Bollendonk, Grant Rose, Michael Schweitzer, Paul Harter

DIAGNOSTICS

The global variable, *errno,* will be set if *getsem* has a zero or negative return code.

BUGS

None.

NAME

Psem – atomically decrement the value of a semaphore

SYNOPSIS

status = Psem(sem_id, value, tov)
int sem_id, value, tov, status

DESCRIPTION

Psem tries to decrement the value of semaphore *sem_id* by *value*. *Sem_id* is the unique semaphore identifier returned from *getsem* and *value* is a positive integer. *Psem* has the following results:

If the semaphore has a value greater than or equal to *value*, then then semaphore is decremented by *value* and *Psem* returns immediately with a return value of zero.

If the semaphore has a value less than *value*, then the calling process is suspended for a maximum time-out period of *tov*. The time-out value is *tov/hz* seconds. Suspended processes are put on a FIFO queue for that semaphore. The value of *tov* has three possible ranges:

(1) equal to -1, causing 'wait-forever',
(2) equal to 0, causing immediate return,
(3) or a positive integer, wait until time-out has expired.

If the value of the semaphore is raised high enough to decrement it by *value*, prior to the end of the time-out period, then *Psem* returns with a return value of zero.

Otherwise, *Psem* will return a non-zero value if it fails to decrement the value of the semaphore within the specified time-out period.

ERRORS

Psem has a zero return value unless there has been an error, in which case the global value errno will be set as follows.

[ETIMEDOUT] If it fails to decrement semaphore *sem_id*, within time-out period, *tov*.

[EBUSY] If the semaphore could not be decremented when *tov* = 0.

[EINVAL] If *value* <= 0.

[ENOMEM] If the kernel free-list for waiting processes is empty.

[EFAULT] If *sem_id* is invalid

EXAMPLE

```
sem_id = 622;          /* semaphore identifier returned from getsem */
value = 1;
tov = 100;
status = Psem(sem_id, value, tov);
```

SEE ALSO

getsem(2), Vsem(2), vshare(2), vrlse(2)

AUTHOR

Greg Bollendonk, Grant Rose, Michael Schweitzer, Paul Harter

DIAGNOSTICS

The global variable, *errno*, will be set if *Psem* has a non-zero return code.

BUGS

Deadlock detection is not implemented.

NAME

Vsem – atomically increment the value of a semaphore

SYNOPSIS

status = Vsem(sem_id, value)
int sem_id, value, status

DESCRIPTION

Vsem increments the value of a semaphore specified by *sem_id,* where *sem_id* is the unique semaphore identifier returned from *getsem.* If the call is successful, the semaphore will be incremented by *value.* If processes are currently suspended waiting to decrement this semaphore, zero or more may be allowed to proceed based on their decrement values.

ERRORS

Vsem has a zero return value unless there has been an error, in which case the global value errno will be set as follows.

[EFAULT] If *sem_id* is invalid

[EINVAL] If *value* <= 0

EXAMPLE

```
sem_id = 6;                    /* semaphore identifier returned from getsem */
value = 1;
status = Vsem(sem_id, value);
```

SEE ALSO

getsem(2), Psem(2), vshare(2), vrlse(2)

AUTHOR

Greg Bollendonk, Grant Rose, Michael Schweitzer, Paul Harter

DIAGNOSTICS

The global variable, *errno,* will be set if *Vsem* has a non-zero return code.

BUGS

Deadlock detection is not implemented.