WWOS (now HotOS), 1992

Position Paper:
Shared Memory Ought to be Commonplace

Michael L. Scott
William Garrett

University of Rochester
Computer Science Department
Rochester, NY 14627-0226
{scott,garrett}@cs.rochester.edu

Shared memory as a programming abstraction is widely used within parallel applications. It is *not* widely used *between* applications, but we argue that it should be. Specifically, we suggest that shared memory is both faster and more intuitive than the principal alternatives in many cases, and that the general disuse of shared memory facilities in systems such as Unix is due in large part to a lack of appropriate tools.

We are pursuing a series of measures to make shared memory more convenient. We use dynamic linking to allow programs to access shared or persistent data the same way they access ordinary variables and functions. We also unify memory and files into a single-level store that facilitates the sharing of pointers and capitalizes on emerging 64-bit architectures. We exploit existing interfaces and tools wherever possible, to remain backward-compatible with Unix.

1. Shared Memory is Good

After a decade of focusing on message passing and RPC in distributed systems, the OS community has in recent years returned to an interest in shared memory. The growing popularity of distributed shared memory systems [5], for example, represents a realization that many programmers and system designers find shared memory attractive as a conceptual model for inter-process interaction even when the underlying hardware provides no direct support.

Memory sharing between arbitrary processes is at least as old as Multics [6]. It suffered something of a hiatus in the 1970s, but has now been incorporated in most variants of Unix. The Berkeley *mmap* facility was designed, though never actually included, as part of the 4.2 and 4.3 BSD releases [3]; it appears in several commercial systems, including SunOS. AT&T's *shm* facility became available in Unix System V and its derivatives. More recently, memory sharing via inheritance has been incorporated in the versions of Unix for several commercial multiprocessors, and the external pager mechanisms of Mach [13] and Chorus can be used to establish data sharing between arbitrary processes.

Shared memory has several important advantages over interaction via files or message passing. It is generally more efficient than either of the other alternatives, since operating system overhead and copying costs can often be avoided. Many programmers find it more conceptually appealing than message passing, at least for certain applications. It facilitates transparent, asynchronous interaction between processes, and shares with files the advantage of not requiring that the interacting processes be active concurrently. Finally, it provides a means of transferring information from one process to another without translating it to and from a (linear) intermediate form. This last advantage is particularly compelling: the code required to save and restore

This work was supported in part by NSF grants CDA-8822724 and CCR-9005633.

information in files and message buffers is a major contributor to software complexity, and much research has been aimed at reducing this burden (e.g. through data description languages and RPC stub generators), with only partial success.

Both files and message passing have applications for which they are highly appropriate. Files are ideal for data that have little internal structure, or that are frequently modified with a text editor. Messages are ideal for RPC and certain other common patterns of process interaction. At the same time, we believe that many interactions currently achieved through files or message passing could better be expressed as operations on shared data. Many of the files described in section 5 of the Unix manual, for example, are really long-lived data structures. It seems highly inefficient, both computationally and in terms of programmer effort, to employ access routines for each of these objects whose sole purpose is to translate what are logically shared data structure operations into file system reads and writes. In a similar vein, we see numerous opportunities for servers to communicate with clients through shared data rather than messages, with savings again in both cycles and programmer effort. Put another way, we believe it is time for the advantages of memory sharing, long understood in the open operating systems community [7, 10, 11], to be extended into environments with multiple users and hardware-enforced protection domains.

2. Shared Memory Is a Pain

Anecdotal evidence suggests that user-level programmers employ shared memory mainly for special-purpose management of memory-mapped devices, and for inter-process interaction within self-contained parallel applications, generally on shared-memory multiprocessors. They do not use it much for interaction *among* applications, or between applications and servers. Why is this?

Much of the explanation, we believe, stems from a lack of convenience. Consider the System V *shm* facility, the most widely available set of shared memory library calls. Processes wishing to share a segment must agree on a 32-bit *key*. Using the key, each process calls *shmget* to create or locate the segment, and to obtain its segment *id*, a positive integer. Each process then calls *shmat* to map the segment into its address space. The name space for keys is small, and there is no system-provided way to allocate them without conflict. *Shmget* and *shmat* take arguments that determine how large the segment is, which process creates it, where it is mapped in each address space, and with what permissions. The user must be aware of these options in order to specify a valid set of arguments. Finally, since *shmat* returns a pointer, references to shared variables and functions must in most languages (including C) be made indirectly through a pointer. There is no performance cost for this indirection on most machines, but there is a loss in both transparency and type safety — static names are not available, explicit initialization is required, and any sub-structure for the shared memory is imposed by convention only.

Less immediate, but equally important, is the issue of long-term shared data management. Segments created by *shmget* exist until explicitly deleted. Though they can be listed (via the *ipcs* command), the simple flat name space is ill-suited to manual perusal of a significant number of segments, and precludes the sort of cleanup that users typically perform in file systems. The *shm* facility makes it too easy to generate garbage segments, and too difficult to name, protect, and account for useful segments.

The Berkeley *mmap* facility is somewhat more convenient. By using the file system naming hierarchy, *mmap* avoids the problems with *shm* keys, and facilitates manual maintenance and cleanup of segments. *Mmap*'s arguments, however, are at least as numerous as those of the *shm* calls. Programmers must still determine who creates a segment. They must open and map segments explicitly, and must be aware of their size, location, protection, and level of sharing. Most important, they must access shared objects indirectly, without the assistance of the language-level naming and type systems.

Linked data structures pose additional problems for cross-application shared memory, since pointers may be interpreted differently by different processes. Any object accessed through shared pointers must appear at the same location in the address space of every process that uses it. If processes map objects into their address spaces dynamically (e.g. as a result of following pointers), the only way to preclude address conflicts is to assign every sharable object a unique, global virtual address. While such an assignment may be possible (if a bit cramped on 32-bit machines), it requires a consensus mechanism that is not a standard part of existing systems.

3. What Can We Do About It?

Our emphasis on shared memory has its roots in the Psyche project [4, 8, 9]. Our focus in Psyche was on mechanisms and conventions that allow processes from dissimilar programming models (e.g. Lynx threads and Multilisp futures) to share data abstractions, and to synchronize correctly. Fundamental to this work was the assumption that sharing would occur both within and among applications. Our current work can be considered an attempt to make that sharing commonplace in the context of traditional operating systems. We use dynamic linking to allow processes to access shared or persistent code and data with the same syntax employed for private code and data. In addition, we unify memory and files into a single-level store that facilitates the sharing of pointers, and capitalizes on emerging 64-bit architectures.

Our dynamic linking system associates each shared memory segment with a Unix .o file, making it appear to the programmer as if that file had been incorporated into the program via separate compilation. Objects (variables and functions) to be shared are generally declared in a separate .h file, and defined in a separate .c file (or in corresponding files of the programmer's language of choice). They appear to the rest of the program as ordinary external objects. The only thing the programmer needs to worry about (aside from algorithmic concerns such as synchronization) is a few additional arguments to the linker; no library or system calls for set-up or shared-memory access appear in the program source.

Our implementation runs on a Silicon Graphics multiprocessor. An extended version of the Unix *ld* linker allows programmers to specify which program modules are to be private and which shared. For private modules, it supports both static and dynamic linking. For shared modules, it supports both persistent and temporary data (the latter is reinitialized when no longer accessed by any running process). A dynamic linker (inspired by Ho's *dld*[1]) takes control at process start-up time, to find and link in shared (and private dynamic) modules. Shared segments appear in the file system, and have names that are derived from the name of the .o template. Each dynamically-linked module may in turn require additional modules to be found and linked. These additional modules are brought in on demand, when first referenced.

To facilitate the use of pointers from and into shared segments, we employ a single-level store in which every sharable object (whether dynamically linked or not) has a unique, globally-agreed-upon virtual address. We retain the traditional Unix file system and shared memory interfaces, both for the sake of backward compatibility and because we believe that these interfaces are appropriate for many applications. We treat the interfaces as alternative *views* of a single underlying abstraction. This is in some sense the philosophy behind the Berkeley *mmap* call; we take it a step further by providing names for ''unnamed'' memory objects (in a manner inspired by Killian's */proc* directory [2]), and by providing every byte of secondary storage with a unique virtual address. To some extent, we return to the philosophy of Multics, but with true global pointers, a flat address space, and Unix-style naming, protection, and sharing.

In our 32-bit prototype, we have reserved a 1 Gbyte region between the Unix bss and stack segments, and have associated this region with a dedicated ''shared file system.'' With 64-bit addresses, we plan to extend the shared file system to include all of secondary store. A user-level handler for the SIGSEGV signal catches references to symbols in segments that have not yet been linked, or to pointers into segments that have not yet been mapped. The handler uses new kernel

calls to translate the faulting address into a path name, and to open and map the file. It invokes the dynamic linker if necessary, and then restarts the faulting instruction.

Example

To illustrate the utility of cross-application shared memory, consider the Unix *rwhod* daemon. Running on each machine, *rwhod* periodically broadcasts local status information (load average, current users, etc.) to other machines, and receives analogous information from its peers. As originally conceived, it maintains a collection of local files, one per remote machine, that contain the most recent information received from those machines. Every time it receives a message from a peer it rewrites the corresponding file. Utility programs read these files and generate terminal output. Standard utilities include *rwho* and *ruptime*, and many institutions have developed local variants. Using an earlier prototype of our tools under SunOS, we re-implemented *rwhod* to keep its database in shared memory, rather than in files, and modified the various lookup utilities to access this database directly. The result is both simpler and faster.

Our modified version of *rwho* saves about 30 lines of code (not a huge amount, but about 17% of the total length). The modified version of *rwhod* is about the same size as the original, mainly because it has to duplicate the associative naming performed ‘‘automatically’’ by the file system directory mechanism. Set-up time for *rwho* expands from about 90 ms to about 250 ms on a Sparcstation 1 (this comparison is a bit unfair to the shared memory approach, because SunOS performs dynamic linking of ‘‘shared’’ libraries prior to giving us a timing hook). At the same time, approximately 20 ms per machine on the network is saved by reading information directly from shared memory, rather than a file. Similarly, in *rwhod*, about 10 ms is saved in the processing of every broadcast packet. On our local network of 65 *rwho*-equipped machines, the new version of *rwhod* saves about 2.1 seconds of compute time per hour on each Sparcstation 1.

4. Status and Plans

As of March 1992, we have a 32-bit version of our tools running on an SGI 4D/480 multiprocessor. These tools consist of (1) extensions to the Unix static linker, to support shared segments; (2) a dynamic linker that finds and maps such segments (and any segments that they in turn require, recursively) on demand; (3) modifications to the file system, including kernel calls that map back and forth between addresses and path name/offset pairs, and (4) a fault handler that adds segments to a process’s address space on demand, triggering the dynamic linker when appropriate. Our SGI machine is scheduled to be upgraded to 64-bit R4000 processors in the fall, at which point we will extend our system to include all of secondary store. Questions for our ongoing work include:

(1) In what form should currently unnamed memory objects (e.g. private text, data, and stack segments) appear in the file system?

(2) Do processes need to be able to overload virtual addresses, or will a single virtual-to-physical translation suffice for the whole machine?

(3) How best can our experience with Psyche (specifically, multi-model parallel programming and first-class user-level threads) be transferred to the Unix environment?

(4) To what extent can in-memory data structures supplant the use of files in traditional Unix utilities?

(5) In general, how much of the power and flexibility of open operating systems can be extended to an environment with multiple users and languages?

Many of the issues involved in this last question are under investigation at Xerox PARC (see [12] in particular). The multiple languages of Unix, and the reliance on kernel protection, pose serious obstacles to the construction of integrated programming environments. It is not clear whether all

of these obstacles can be overcome, but there is certainly much room for improvement. We believe that shared memory is the key.

Acknowledgment

Our ongoing work is the subject of a spring 1992 graduate seminar at Rochester, and is a collaborative effort with Ricardo Bianchini, Leonidas Kontothanassis, Andrew McCallum, Jeff Thomas, and Bob Wisniewski. Details on our prototype tools and future plans can be found in a technical report, now in preparation.

References

[1] W. W. Ho and R. A. Olsson, ‘‘An Approach to Genuine Dynamic Linking,’’ *Software — Practice and Experience 21*:4 (April 1991), pp. 375-390.

[2] T. J. Killian, ‘‘Processes as Files,’’ *Proceedings of the Usenix Software Tools Users Group Summer Conference*, 12-15 June 1984, pp. 203-207.

[3] S. J. Leffler, M. K. McKusick, M. J. Karels and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, The Addison-Wesley Publishing Company, Reading, MA, 1989.

[4] B. D. Marsh, M. L. Scott, T. J. LeBlanc and E. P. Markatos, ‘‘First-Class User-Level Threads,’’ *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, 14-16 October 1991, pp. 110-121. In *ACM SIGOPS Operating Systems Review 25*:5.

[5] B. Nitzberg and V. Lo, ‘‘Distributed Shared Memory: A Survey of Issues and Algorithms,’’ *Computer 24*:8 (August 1991), pp. 52-60.

[6] E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, Cambridge, MA, 1972.

[7] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray and S. C. Purcell, ‘‘Pilot: An Operating System for a Personal Computer,’’ *Communications of the ACM 23*:2 (February 1980), pp. 81-92.

[8] M. L. Scott, T. J. LeBlanc and B. D. Marsh, ‘‘A Multi-User, Multi-Language Open Operating System,’’ *Proceedings of the Second Workshop on Workstation Operating Systems*, 27-29 September 1989, pp. 125-129.

[9] M. L. Scott, T. J. LeBlanc and B. D. Marsh, ‘‘Multi-Model Parallel Programming in Psyche,’’ *Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming*, 14-16 March, 1990, pp. 70-78. In *ACM SIGPLAN Notices 25*:3.

[10] D. Swinehart, P. Zellweger, R. Beach and R. Hagmann, ‘‘A Structural View of the Cedar Programming Environment,’’ *ACM Transactions on Programming Languages and Systems 8*:4 (October 1986), pp. 419-490.

[11] J. H. Walker, D. A. Moon, D. L. Weinreb and M. McMahon, ‘‘The Symbolics Genera Programming Environment,’’ *IEEE Software 4*:6 (November 1987), pp. 36-45.

[12] M. Weiser, L. P. Deutsch and P. B. Kessler, ‘‘UNIX Needs a True Integrated Environment: CASE Closed,’’ Technical Report CSL-89-4, Xerox PARC, 1989. Earlier version published as ‘‘Toward a Single Milieu,’’ *UNIX Review 6*:11.

[13] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black and R. Baron, ‘‘The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System,’’ *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, 8-11 November 1987, pp. 63-76. In *ACM SIGOPS Operating Systems Review 21*:5.