Linking Shared Segments

W. E. Garrett, M. L. Scott, R. Bianchini, L. I. Kontothanassis, R. A. McCallum, J. A. Thomas, R. Wisniewski and S. Luk – University of Rochester

ABSTRACT

As an alternative to communication via messages or files, shared memory has the potential to be simpler, faster, and less wasteful of space. Unfortunately, the mechanisms available for sharing in Unix are not very easy to use. As a result, shared memory tends to appear primarily in self-contained parallel applications, where library or compiler support can take care of the messy details. We have developed a system, called Hemlock, for transparent sharing of variables and/or subroutines across application boundaries. Our system is backward compatible with existing versions of Unix. It employs dynamic linking in conjunction with the Unix *mmap* facility and a kernel-maintained correspondence between virtual addresses and files. It introduces the notion of *scoped linking* to avoid naming conflicts in the face of extensive sharing.

1. Introduction

Multi-user operating systems rely heavily on the ability of processes to interact with one another, both within multi-process applications and between applications and servers of various kinds. In the Unix world, processes typically interact either through the file system, or via some form of message passing. Both mechanisms have their limitations, however, and support for a third approach — shared memory — can also be extremely useful.

Memory sharing between arbitrary processes is at least as old as Multics[17]. It suffered something of a hiatus in the 1970s, but has now been incorporated into most variants of Unix. The Berkeley *mmap* facility was designed, though never actually included, as part of the 4.2BSD and 4.3BSD releases[12]; it appears in several commercial systems, including SunOS. ATT'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[1] and Chorus[18] can be used to establish data sharing between arbitrary processes.

Shared memory has several important advantages over interaction via files or messages.

1. Many programmers find shared memory more conceptually appealing than message passing. The growing popularity of distributed shared memory systems[16] suggests that programmers will adopt a sharing model even at the expense of performance.
2. Shared memory facilitates transparent, asynchronous interaction between processes, and shares with files the advantage of not requiring that the interacting processes be active concurrently.
3. When interacting processes agree on data formats and virtual addresses, shared memory provides a means of transferring information from one process to another without translating it to and from a (linear) intermediate form. The code required to save and restore 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).
4. When supported by hardware, shared memory is generally faster than either messages or files, since operating system overhead and copying costs can often be avoided. Work by Bershad and Anderson, for example[4], indicates that message passing should be built on top of shared memory when possible.
5. As an implementation technique, sharing of read-only objects can save significant amounts of disk space and memory. All modern versions of Unix arrange for processes executing the same load image to share the physical page frames behind their text segments. Many (e.g., SunOS and SVR4) extend this sharing to dynamically-linked position-independent libraries. More widespread use of position-independent code, or of logically-shared, re-entrant code, could yield additional savings.

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.

Despite its merits, however, shared memory in Unix remains largely confined to inheritance-based sharing within self-contained multiprocessor applications, and special-purpose sharing with devices. We speculate that much of the reason for this limited use lies in the lack of a transparent interface: access to private memory is much simpler and more easily expressed than access to shared memory; sharing is difficult to set up in the first place and variable and functions in shared memory cannot be named directly.

Both the System V *shm* and Berkeley *mmap* facilities require the user to know significant amounts of set-up information before sharing can take place. Processes must agree on ownership of a shared segment, and (if pointers are to be used) on its location in their respective address spaces. Processes using *shm* must also agree on some form of naming convention to identify shared segments (*mmap* uses file system naming). Most important, neither *mmap* nor *shm* allows language level access to shared segments. 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 many machines, but there is a loss in both transparency and type safety — static names are not available, explicit initialization is required, and any substructure for the shared memory is imposed by convention only.

In an attempt to address these problems we have developed a system, Hemlock,[1] that automates the creation and use of shared segments. Our goal in developing Hemlock was to simplify the interface to shared memory facilities while increasing the flexibility of the shared memory segments at the same time. Hemlock consists of new static and dynamic linkers, a run-time library, and a set of kernel extensions. These components cooperate to map and link shared segments into programs, providing type safety and language-level access to shared objects, and hiding the distinction between shared and private objects. Hemlock also facilitates the use of pointers to shared objects by maintaining a special file system, with a globally-consistent mapping between virtual addresses and sharable files. The mapping ensures that a given shared object lies at the same virtual address in every address space. Finally, through its *lazy dynamic linking*, Hemlock allows the programmer to design applications whose components, both private and shared, are determined at run time.

We focus in this paper on linker support for sharing, including *scoped linking* to avoid the naming conflicts that arise when linking across conventional application boundaries, *dynamic linking* to permit the private and shared components of applications to be determined at run time, and *lazy linking* to minimize unnecessary work. We provide an overview of Hemlock in section 2, and a more detailed description of its linkers in section 3. We describe example applications in section 4, discuss some semantic subtleties in section 5, and conclude in section 6.

2. An Overview of Hemlock

Our emphasis on shared memory has its roots in the Psyche project[19, 20]. 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[14, 21]. The fundamental assumption of this work was that sharing would occur both within and among applications. Our current work[7, 23] is an attempt to make that sharing commonplace in the context of traditional operating systems.

Hemlock uses dynamic linking to allow processes to access shared code and data with the same syntax employed for private code and data. It also places shared segments into a special file system that maintains a globally-consistent mapping between sharable objects and virtual addresses, thereby ensuring that pointers to shared objects will be interpreted consistently in different protection domains. Unlike the ‘‘shared’’ libraries of systems such as SunOS and SVR4, Hemlock supports genuine write sharing, not just the physical sharing of logically private pages. Unlike such integrated programming environments as Cedar[24] and Emerald[10], it supports sharing of modules written in conventional languages, in a manner that is backward compatible with Unix. An early prototype of Hemlock ran under SunOS, but we are now working on Silicon Graphics machines (with SGI’s IRIX operating system). Our long-term plans call for the exploitation of processors with 64-bit addressing, but this is beyond the scope of the current paper.

We use the term *segment* to refer to what Unix and Mach call a ‘‘memory object’’. Each segment can be accessed as a *file* (with the traditional Unix interface), or it can be mapped into a process’s address space and accessed with load and store instructions. A segment that is linked into an address space by our static or dynamic linkers is

[1] Named for an evergreen tree species common in upstate New York, and for one of the lakes from which Rochester obtains its water supply.

referred to as a *module*. Each module is created from a *template* in the form of a Unix .o file. Each template contains references to *symbols*, which are names for *objects*, the items of interest to programmers. (Objects have no meaning to the kernel.) The linkers cooperate with the kernel to assign a virtual address to each module. They *relocate* modules to reside at particular addresses (by finalizing absolute references to internal symbols; some systems call this *loading*), and they *link* modules together by resolving cross-module references.

Our linkers associate a shared 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 (see Figure 1). 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.

Hemlock's linker for sharing, *lds*, is currently implemented as a wrapper that extends the functionality of the Unix *ld* linker. Lds defines four *sharing classes* for the object modules (.o files) from which an executing program is constructed. These classes are *static private*, *dynamic private*, *static public*, and *dynamic public*. Classes can be specified on a module-by-module basis in the arguments to lds. They differ with respect to the times at which they are created and linked, and the way in which they are named and addressed; see Table 1[2].

At static link time, lds creates a load image containing a new instance of every private static module. It also creates any public static modules that do not yet exist, but leaves them in separate files; it does not copy them into the load image. A public module resides in the same directory as its template (.o) file, and has a name obtained by dropping the final '.o'. It also has a unique, globally-

[2]For the purposes of this paper, we use the word 'process' in the traditional Unix sense. Like most researchers, we believe that operating systems should provide separate abstractions for threads of control and protection domains. Our work is compatible with this separation, but does not depend upon it.

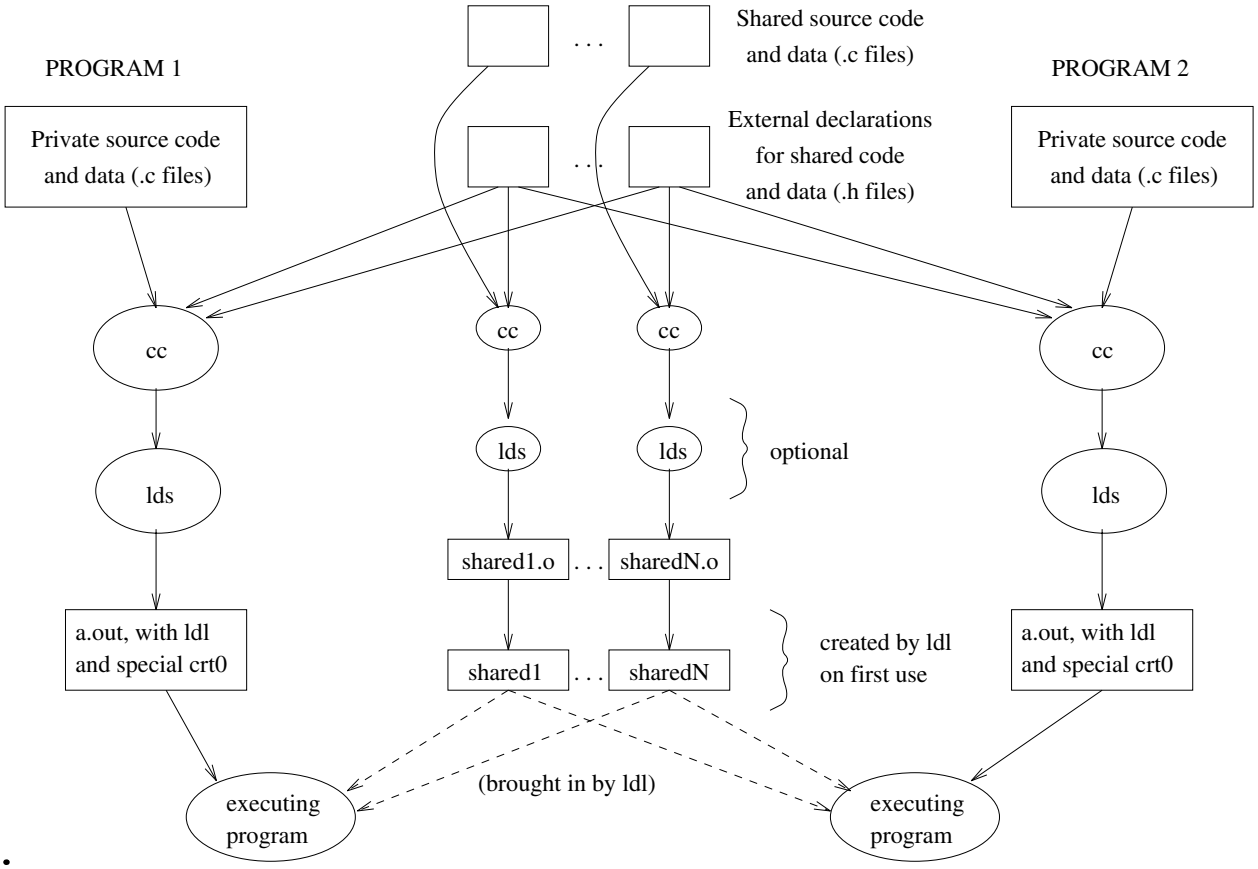

Figure 1: Building a Program with Linked-in Shared Objects

agreed-upon virtual address, and is internally relocated on the assumption that it resides at that address. Public modules are persistent; like traditional files they continue to exist until explicitly destroyed.

Lds resolves undefined references to symbols in static modules. It does not resolve references to symbols in dynamic modules. In fact, it does not even attempt to determine which symbols are in which dynamic module, or insist that the modules yet exist. Instead, lds saves the module names and search path information in the program load image, and links in an alternative version of crt0.o, the Unix program start-up module. At run time, crt0 calls our lazy dynamic linker, *ldl*.

Ldl uses the saved information to locate dynamic modules. It creates a new instance of each dynamic private module, and of each dynamic public module that does not yet exist. It then maps static public modules and all dynamic modules into the process address space, and resolves undefined references from the main load image to objects in the dynamic modules. If any module contains undefined references (this is likely for dynamic private modules, and possible for newly-created public modules), ldl maps the module without access permissions, so that the first reference will cause a segmentation fault. It installs a signal handler for this fault. When a fault occurs, the signal handler resolves any undefined external references in (all pages of) the module that has just been accessed, mapping in (possibly inaccessibly) any new modules that are needed.

This *lazy linking* supports a programming style in which users refer to modules, symbolically, throughout their programming environment. It allows us to run processes with a huge ''reachability graph'' of external references, while linking only the portions of that graph that are actually used during any particular run. We envision, for example, rewriting the *emacs* editor with a functional interface to which every process with a text window can be linked. With lazy linking, we would not bother to bring the editor's more esoteric features into a particular process's address space unless and until they were needed.

At static link time, modules are specified to lds the same way they are specified to ld: as absolute or relative path names. When attempting to find modules with relative names, lds uses a search path that can be altered by the user. It looks first in the current directory, next in an optional series of directories specified via command-line arguments, then in an optional series of directories specified via an environment variable, and finally in a series of default directories. Lds applies the search strategy at static link time for modules with a static sharing class. It passes a description of the search strategy to ldl for use in finding modules with a dynamic sharing class.

A template (.o) file is generally produced by a compiler. In addition, it can at the user's discretion be run through lds, with an argument that retains relocation information. In this case, lds can be asked to include search strategy information in the new .o file. When creating a new dynamic module from its template at run time, ldl attempts to resolve undefined references out of the new module using the search strategy (if any) specified to lds when creating that module. If this strategy fails, it reverts to the strategy of the module(s) that make references into the new module. This *scoped linking* preserves abstraction by allowing a process to link in a large subsystem (with its own search rules), without worrying that symbols in that subsystem will cause naming conflicts with symbols in other parts of the program. Scoped linking is discussed in further detail in the following section.

To facilitate the use of pointers, we must insist that all public modules be linked at the same virtual address in every protection domain. To ensure such *uniform addressing* on a 64-bit machine, we would associate a unique range of virtual addresses with every Unix file. On 32-bit machine, we maintain addresses only for files on a special disk partition, and then insist that public modules (and the templates from which they are created) reside on this partition. We retain the traditional Unix interfaces to the shared file system, both for the sake of backward compatibility and because we believe that these interfaces are appropriate for many applications.

The user-level handler for the SIGSEGV signal catches references to modules that are not currently

Sharing Class	When linked	New instance created/destroyed for each process	Default portion of address space
Static private Dynamic private	Static link time Run time	yes	Private
Static public Dynamic public	Static link time Run time	no	Public

Table 1: Class creation and link times

part of the address space of the executing process. The handler actually serves two purposes: it cooperates with ldl to implement lazy linking, and it allows the process to follow pointers into segments that may or may not yet be mapped. When triggered, the handler checks to see if the faulting address lies in the shared portion of the process's address space. If so, it uses a (new) kernel call to translate the address into a path name and, access rights pemitting, maps the named segment into the process's address space. If the address lies in a module that has been set up for lazy linking, the handler invokes ldl to resolve any undefined or relocatable references. (These may in turn cause other modules to be set up for lazy linking.) Otherwise, the handler opens and maps the file. It then restarts the faulting instruction. For compatibility with programs that already catch the SIGSEGV signal, the library containing our signal handler provides a new version of the standard *signal* library call. When the dynamic linking system's fault handler is unable to resolve a fault, a program-provided handler for SIGSEGV is invoked, if one exists.

3. Linking in Hemlock

Linker support for sharing capitalizes on the lowest common denominator for language implementations: the object file. By making modules correspond to object files, Hemlock gives the programmer first-class access to the objects they contain — with language-level naming, type checking, and scope rules — without modifying the compilers. By comparison, sharing based on pointer-returning system calls is comparatively distant from the programming language. The subsections below provide additional detail on the linkers, the shared file system, and the rationale for lazy and scoped linking.

The Linkers

Our current static linker is implemented as a wrapper, lds, around the standard IRIX ld linker. The wrapper processes new command line options directly related to its functionality and passes the others to ld. Lds-specific options allow for the association of sharing classes with modules and the specification of search paths to be used when locating modules. In addition, lds provides ldl with relocation information about static modules and warns the user if the dynamic modules do not yet exist. We are in the process of building a completely new stand-alone static linker that will also support scoped linking, currently available only in the dynamic linker, ldl.

Both lds and ldl use an extended search strategy for modules, inspired by the analogous strategy in the SunOS dynamic linker. At static link time, lds searches for modules in (1) the current directory, (2) the path specified in a special command-line argument, (3) the path specified by the LD_LIBRARY_PATH environment variable, and (4) the default library directories. If there is more than one static module with the same name, lds uses the first one it finds. At execution time, ldl searches for dynamic modules in (1) the path specified by the LD_LIBRARY_PATH environment variable, and (2) the directories in which lds searched for static modules: the directory in which static linking occurred, the directories specified on the lds command line, the directories specified by the LD_LIBRARY_PATH variable at static link time, and the default directories. Users can arrange to use new versions of dynamic modules by changing the LD_LIBRARY_PATH environment variable prior to execution. This feature is useful for debugging and, more important, for customizing the use of shared data to the current user or program instance. (We return to this issue in section 4 below.) Lds aborts linking if it cannot find a given static module. It issues a warning message and continues linking if it cannot find a given dynamic module.

To support the dynamic linker, lds creates a data structure listing the dynamic modules, and describing the search path it used for static modules. To give ldl a chance to run prior to normal execution, lds links C programs with a special start-up file. It would use similar files for other programming languages. Ldl also creates any static public modules that do not yet exist, and initializes those objects from their templates. Finally, in the current wrapper-based implementation, lds must compensate for certain shortcomings of the IRIX ld. Ld refuses to retain relocation information for an executable program, so lds must save this in an explicit data structure. Ld also refuses to resolve references to symbols at absolute addresses (as required for static public modules), so lds must do so.

Ldl differs from most dynamic linkers in several ways. Its facilities for lazy and scoped linking are discussed in more detail below. In addition, it will use symbols found in dynamically-linked modules to resolve undefined references in the statically-linked portion of the program, even when the location of those symbols was not known at static link time. To cope with an unfortunate limitation of the R3000 architecture, ldl insists that modules be compiled with a flag that disables use of the processor's performance-enhancing global pointer register. Addressing modes that use the pointer are limited to 24 bit offsets, and are incompatible with a large sparse address space. To cope with a similar 28-bit addressing limit on the processor's jump instructions, lds and ldl arrange for over-long branches to be replaced with jumps to new, nearby code fragments that load the appropriate target address into a register and jump indirectly.

Address Space and File System Organization

Given appropriate rights, programs should be able to access a shared object or segment simply by using its name. But different kinds of names are useful for different purposes. For human beings, ease of use generally implies symbolic names, both for objects and for segments: the linkers therefore accept file system names for segments, and support symbolic names for objects. For running programs, on the other hand, ease of use generally implies addresses: programs need to be able to follow pointers, even if they cross segment boundaries. It is easy to envision applications in which both types of names are useful. Any program that shares data structures and also manipulates segments as a whole may need both sets of names.

In our 32-bit prototype, we have reserved a 1G-byte region between the Unix heap and stack segments, and have associated this region with the kernel-maintained shared file system. The file system is configured to have exactly 1024 inodes, and each file is limited to a maximum of 1M bytes in size. Hard links (other than '.' and '..') are prohibited, so there is a one-one mapping between inodes and path names. We have modified the IRIX kernel to keep track of the mapping internally, and have provided system calls that translate back and forth.

All of the normal Unix file operations work in the shared file system. The only thing that sets it apart is the association between file names and addresses. Mapping from file names to addresses is easy: the *stat* system call already returns an inode number. We provide a new system call that returns the filename for a given inode, and we overload the arguments to *open* so that the programmer can open a file by address instead of by name, with a single system call. For the sake of simplicity, the mapping in the kernel from addresses to files employs a linear lookup table. We initialize the table at boot time by scanning the entire shared file system, and update it as appropriate when files are created and destroyed. For an experimental prototype, these measures have the desirable property of allowing the filename/address mapping to survive system crashes without requiring modifications to on-disk data structures or to utilities like *fsck* that understand those structures.

With 64-bit addresses, we will extend the shared file system to include all of secondary store, and will relax the limits on the number and sizes of shared files. We plan to provide every segment, whether shared or not, with a unique, system-wide virtual address. At the same time, we plan to retain the ability to overload addresses within a reserved, private portion of the 64-bit space. Within the kernel, we will abandon the linear lookup table and the direct association between inode numbers and addresses. Instead, we will add an address field to the on-disk version of each inode, and will link these inodes into a lookup structure — most likely a B-tree — whose presence on the disk allows it to survive across re-boots.

Lazy Dynamic Linking

Public modules in Hemlock can be linked both statically and dynamically. The advantage of dynamic linking is that it allows the makeup of a program to be determined very late. With dynamic linking, an application can be composed of different modules from run to run, depending on who is running it, what directories and modules currently exist, what changes have recently been made to environment variables, etc.

We expect to rely on run-time identification of modules for a variety of purposes. By using search paths containing directories that are named relative to the current or home directory, we can arrange for applications to link in data structures that are shared with other applications belonging to the same user, project etc. Similarly, by modifying environment variables prior to execution, we can arrange for new processes to find shared data in a temporary directory. We describe the use of this technique in parallel applications in section 4 below.

Dynamic linking is already used in several Unix systems (including SunOS and SVR4) to save space in the file system and in physical memory, and to permit updating of libraries without recompiling all the programs that employ them. In many of these systems, position-independent code (PIC) permits the text pages of libraries to be physically shared, but this is only an optimization; each process has a private copy of any static variables. The PIC produced by the Sun compilers uses jump tables that allow functions to be linked lazily, but references to data objects are all resolved at load time. Sun's ld also insists that all dynamically-linked libraries exist at static link time, in order to verify the names of their entry points.

Hemlock uses dynamic linking for both private and shared data, and does not insist on knowing at static link time which symbols will be found in which dynamically-linked modules. This latter point may delay the reporting of errors, and can increase the cost of run-time linking, but increases flexibility. Lds requires only that the user specify the names of all modules containing symbols accessed directly from the main load image. It then accepts arguments that allow the user to specify a search path on which to look for those modules at run time. Any module found may in turn specify a search path on which to look for modules containing symbols that *it* references.

Our fault-driven lazy linking mechanism is slower than the jump table mechanism of SunOS, but works for both functions and data objects, and does not require compiler support. We do not currently share the text of private modules, but will

do so when PIC-generating compilers become available under IRIX. Given the opportunity, we will adopt the SunOS jump-table-based lazy linking mechanism as an optimization: modules first accessed by calling a (named) function will be linked without fault-handling overhead.

Several dynamic linkers, including the Free Software Foundation's dld[9] and those of SunOS and SVR4, provide library routines that allow the user to link object modules into a running program. Dld will resolve undefined references in the modules it brings in, allowing them to point into the main program or into other dynamically-loaded modules. The Sun and SVR4 routines (*dlopen* and *dlsym*) do not provide this capability; they require the newly-loaded module be self-contained. Neither dld nor the explicitly-invoked Sun/SV routines resolves undefined references in the main program; they simply return pointers to the newly-available symbols.

Scoped Linking

Traditional linking systems, both static and dynamic, deal only with private symbols. They bind all external references to a given name to the same object in all linked modules. If more than one module exports an object with a given name, the linker either picks one (e.g., the first) and resolves all references to it, or reports an error. Our system of dynamic linking, with shared symbols and recursive, lazy inclusion of modules, presents cases where either behavior is undesirable.

Specifying that a module is to be included in a program starts a link in a potentially long chain. Hemlock allows modules to have their own search path and list of modules, which in turn may have their own lists, recursively. Linking a single module may therefore cause a chain reaction that ends up incorporating modules that the original programmer knew nothing about. These modules may have external symbols that the original program knew nothing about. Some of these external symbols may have the same name as external symbols exported by the main program, even though they are actually unrelated. This possibility introduces a potentially serious naming conflict.

The problem is that linkers map from a rich hierarchy of abstractions to a flat address space.

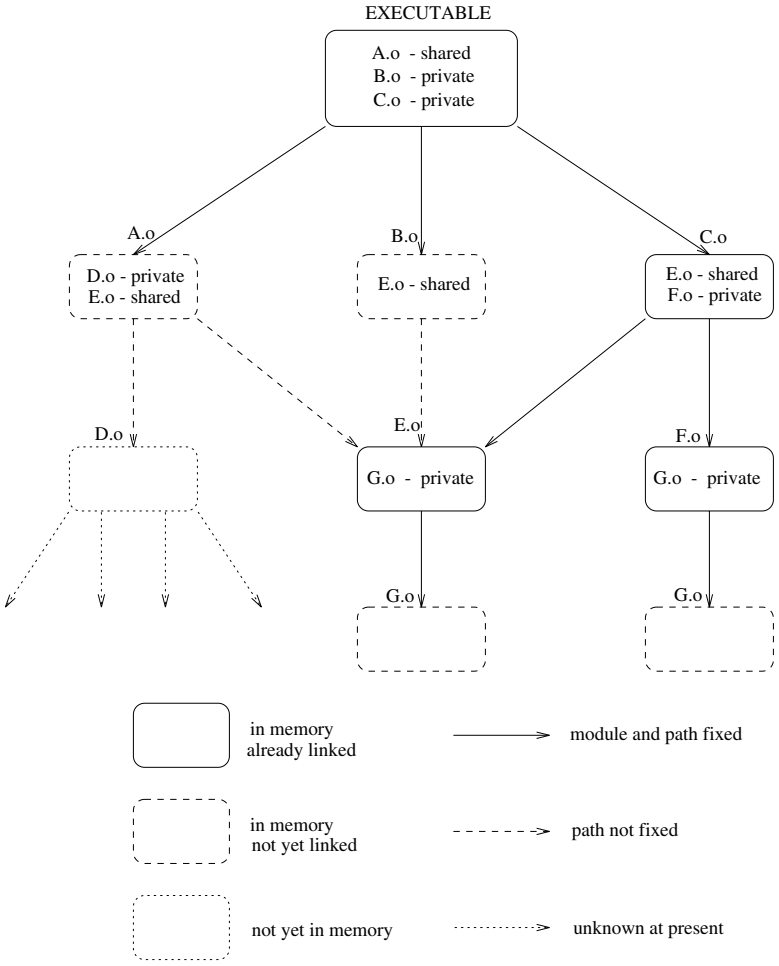

Figure 2: Hierarchical Inclusion of Dynamically-Linked Modules

Various programming languages (e.g., Modula-2 and Common Lisp) that use the idea of a module for abstraction already deal with this problem. Their implementations typically preface variables and function names with module names, thereby greatly reducing the chance of naming conflicts. Scoped linking provides similar freedom from ambiguity, in a language-independent way.

When a module M is brought in, its undefined references are first resolved against the external symbols of modules found on M's own module list and search path. If this step is not completely successful, consideration moves up to the module(s) that caused M to be loaded in — M's ''parent'', so to speak: remaining undefined references are resolved against the external symbols of modules found on the parent's module list and search path. If unresolved references still remain, they are then resolved using the module list and search path of M's grandparent, and so on.

The linking structure of a program can be viewed as a DAG (see Figure 2), in which children can search up from their current position to the root, but never down. Modules wishing to have control over their symbols must specify appropriate modules and directories on their module list and search path. Modules wishing to rely on a symbol being resolved by the parent can simply neglect to provide this information. References that remain undefined at the root of the DAG are left unresolved in the running program. If encountered during execution they result in segmentation faults that are caught by the signal handler, and could be used (at the programmer's discretion) to trigger application-specific recovery.

4. Example Applications

In this section we consider several examples of the usefulness of cross-application shared memory.

Administrative Files

Unix maintains a wealth of small administrative files. Examples include much of the contents of /etc, the score files under /usr/games, the many ''dot'' files in users' home directories, bitmaps, fonts, and so on. Most of these files have a rigid format that constitutes either a binary linearization or a parsable ASCII description of a special-purpose data structure. Most are accessed via utility routines that read and write these on-disk formats, converting them to and from the linked data structures that programs really use.

For the designer of a new structure, the avoidance of translation may not be overwhelming, but it is certainly attractive. As an example of the possible savings in complexity and cost, consider the *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 the early 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 was both simpler and faster. On our local network of 65 *rwhod*-equipped machines, the new version of *rwho* saves a little over a second each time it is called. Though not earthshaking, this savings may be significant: many members of our department run a windowing variant of *rwho* every 60 seconds. We are currently porting the new server and utilities to our SGI-based system.

Utility Programs and Servers

Traditionally, UNIX has been a fertile environment for the creation and use of small tools that can be connected together, e.g., via pipes. Other systems, including Multics and the various open operating systems[24, 25] encourage the construction of similar building blocks at the level of functions, rather than program executables. In future work, we plan to use Hemlock facilities to experiment with functional building blocks in Unix. We also plan to experiment with the use of shared data to improve the performance of interfaces between servers and their clients.

When synchronous interaction is not required, modification of data that will be examined by another process at another time can be expected to consume significantly less time than kernel-supported message passing or remote procedure calls. Even when synchronous communication across protection domains is required, sharing between the client and server can speed the call. In their work on lightweight and user-level remote procedure calls, Bershad et al. argue that high-speed interfaces permit a much more modular style of system construction than has been the norm to date[4]. The growing interest in *microkernels*[28] suggests that this philosophy is catching on. In effect, the microkernel argument is that the proliferation of boundaries becomes acceptable when crossing these boundaries is cheap. We believe that it is even more likely to become acceptable when the boundaries are blurred by sharing, and processes can interact without necessarily crossing anything.

Parallel Applications

A parallel program can be thought of as a collection of sequential processes cooperating to accomplish the same task. Threads in a parallel application need to communicate with their peers for

synchronization and data exchange. On a shared memory multiprocessor this communication occurs via shared variables. In most parallel environments global variables are considered to be shared between the the threads of an application while local variables are private to a thread. In systems like Presto[3], however, both shared *and* private global variables are permitted. Presto was originally designed to run on a Sequent multiprocessor under the Dynix operating system. The Dynix compilers provide language extensions that allow the programmer to distinguish explicitly between shared and private variables. The SGI compilers, on the other hand, provide no such support.

When we set out to port Presto to IRIX in the fall of 1991, the lack of compiler-supported language extensions became a major problem. The solution we eventually adopted was to explicitly place shared variables in memory segments shared between the processes running the application. Placement had to be done by editing the assembly code, and was extremely tedious when attempted by hand. We created a post-processor to automate this procedure; it is 432 lines long (including 105 lines of *lex* source), and consumes roughly one quarter to one third of total compilation time. It also embeds some compiler dependencies; we were forced to re-write it when a new version of the C compiler was released.

We are currently modifying our Presto implementation to use our dynamic linking tools. Selective sharing can be specified with ease. Shared variables must still be grouped together in a separate file, but editing of the assembly code is no longer required. The parent process of the application, which exists solely for set-up purposes, and does none of the application's work, does not link the shared data file. Rather, it creates a temporary directory, puts a symbolic link to the shared data template into this directory, and then adds the name of the directory to the LD_LIBRARY_PATH environment variable. At static link time, the child processes of the parallel application specify that the shared data structures should be linked as a dynamic public module. When the parent starts the children, they all find the newly-created symlink in the temporary directory. The first one to call ldl creates and initializes the shared data from the template, and all of them link it in.[3] When the computation terminates the parent process performs the necessary cleanup, deleting the shared segment, template symlink, and temporary directory.

Programs with Non-Linear Data Structures

Even when data structures are not accessed concurrently by more than one process, they may be shared sequentially over time. Compiler symbol tables are a canonical example. In a multi-pass compiler, pointer-rich symbol table information is often linearized and saved to secondary store, only to be reconstructed in its original form by a subsequent pass. The complexity of this saving and restoring is a perennial complaint of compiler writers, and much research has been devoted to automating the process[15].[4] Similar work has occurred in the message-passing community[8].

With pointers permitted in files, and with a global consensus on the location of every segment, pointer-rich data structures can be left in their original form when saved across program executions. Segments thus saved are position-dependent, but for the compiler writer this is not a problem; the idea is simply to transfer the data between passes.

In a related case study, we have examined our compiler for the Lynx distributed programming language[22], designed around scanner and parser generators developed at the University of Wisconsin. The Wisconsin tools produce numeric tables which a pair of utility programs translate into initialized data structures for separately-developed scanner and parser drivers, written in Pascal. Since Pascal lacks initialized static variables, the initialization trick depends on a non-portable correspondence in data structure layouts between C and Pascal.

With Hemlock, the utility programs that read the numeric output of the scanner and parser generators would share a persistent module (the tables) with the Lynx compiler. The utility programs would initialize the tables; the compiler would link them in and use them. These changes would eliminate between 20 and 25% of code in the utility programs. They would also save a significant amount of time: the C version of the tables is over 5400 lines, and takes 18 seconds to compile on a Sparcstation 1.

An additional example can be found in the *xfig* graphical editor, which we have re-written under Hemlock. While editing, xfig maintains a set of linked lists that represent the objects comprising a figure. It originally translated these lists to and from a pointer-free ASCII representation when reading and writing files. As the same time, xfig must be able to copy the pointer-rich representation, to duplicate objects in a figure. The Hemlock version of xfig uses the pre-existing copy routines for files, at a savings of over 800 lines of code.

5. Discussion

Public vs. Private Code and Data

A representation of addressing in Hemlock appears in Figure 3. The public portion of the address space appears the same in every process,

[3] Ldl uses file locking to synchronize the creation of shared segments.

[4] Some of this research is devoted to issues of machine and language independence, but much of it is simply a matter of coping with pointers.

though which of its segments are actually accessible will vary from one protection domain to another. Addresses in the private portion of the address space are overloaded; they mean different things to different processes. Private modules (including the main module of every process) are linked into the private, overloaded portion of the address space, while public modules are linked at their globally-understood address.

Every program begins execution in the private portion of the address space. In our current 32-bit system, only one quarter of the address space is public, and traditional, unmodified Unix programs never use public addresses. In a 64-bit system, the vast majority of the address space would be public, and we would expect programmers to gradually adopt a style of programming in which public addresses are used most of the time. Backward compatibility is thus the principal motivation for providing private addresses. Some existing programs (generally not good ones) assume that they are linked at a particular address. Most existing programs are created by compilers that use absolute addressing modes to access static data, and assume that the data are private. Many create new processes via *fork*.

Chase, et al.,[5] observe that the Unix fork mechanism is based in a fundamental way on the use of static, private data at fixed addresses. Their Opal system, which adopts a strict, single global translation, dispenses with fork in favor of an RPC-based mechanism for animating a newly-created protection domain. We adopted a similar approach in Psyche; we agree that fork is an anachronism. It works fine in Hemlock, however, and we retain it by weight of precedent. The child process that results from a fork receives a copy of each segment in the private portion of the parent's address space, and shares the single copy of each segment in the public portion of the parent's address space. In all cases, the parent and child come out of the fork with identical program counters. If the parent's PC was at a private address, the parent and child come out in logically private but identical copies of the code. If the parent's PC was at a public address, the parent and child come out in logically shared code, which must be designed for concurrent execution in order to work correctly.

Like Psyche, Hemlock adopts the philosophy that code should be considered shared precisely when its static data is shared. Under this philosophy, the various implementations of ''shared''

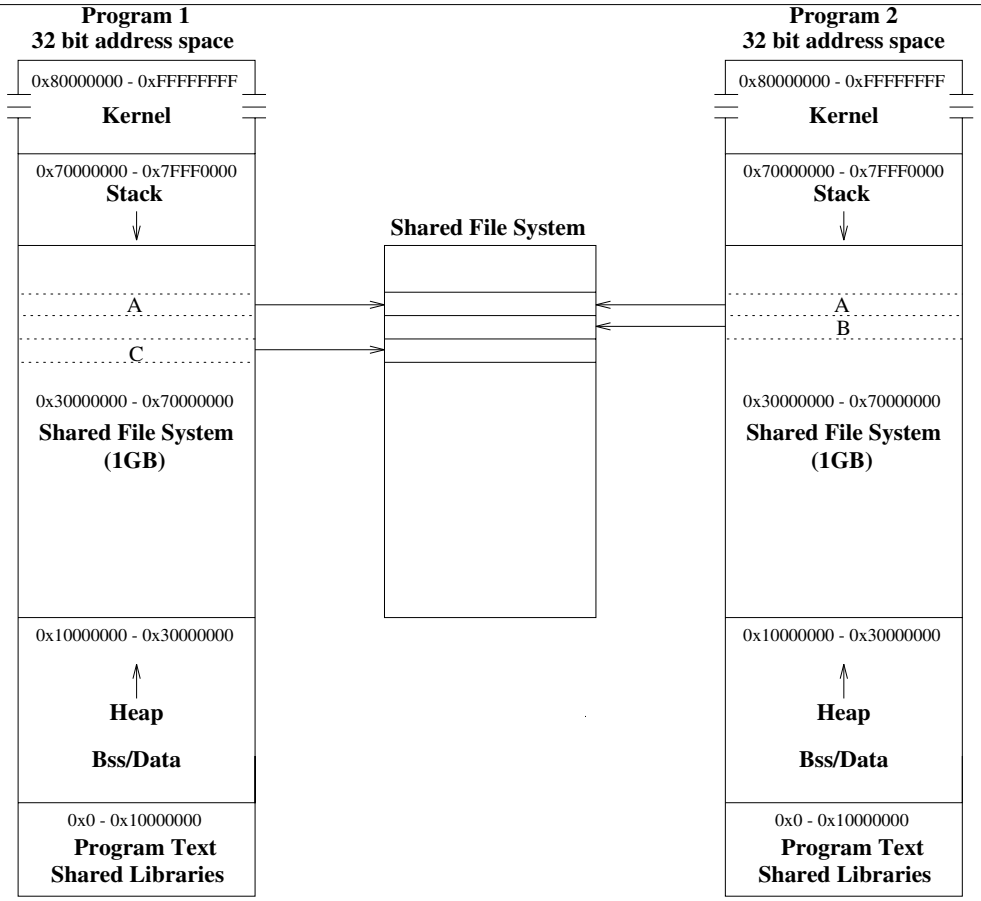

Figure 3: Hemlock Address Spaces

libraries in Unix are in fact space-saving implementations of logically *private* libraries. There is no philosophical difference between these implementations and the much older notion of ''shared text''; one is implemented in the kernel and the other in the linkers, but both serve to conserve physical page frames while allowing the programmer to ignore the existence of other processes.

A different philosophical position is taken in systems such as Multics[17], Hydra[27], and Opal, which clearly separate code from data and speak explicitly of processes executing in shared code but using private (static) data. Multics employs an elaborate hardware/software mechanism in which references to static data are made indirectly through a base register and process-private link segment. Hydra employs a capability-based mechanism implemented by going through the kernel on cross-segment subroutine calls. Opal postulates compilers that generate code to support the equivalent of Multics base registers in an unsegmented 64-bit address space.

With most existing Unix compilers, processes executing the same code at the same address will access the same static data, unless the data addresses are overloaded. This behavior is consistent with the Hemlock philosophy. Code in the private portion of the address space is private; if it happens to lie at the same physical address as similar-looking code in another address space (as in the case of Unix shared text), the overloading of private addresses still allows it to access its own copy of the static data. Code in the public portion of the address space is shared if and only if more than one process chooses to execute it, in which case all processes access the same static data.

In practice, we can still share physical pages of code between instances of the same module by using position-independent code (PIC), which embeds no assumptions (even after linking) about the address at which it executes or about the addresses of its static data or external code or data. Compilers that generate linkage-table-based PIC are already used for shared libraries in SunOS and SVR4, and will soon be available under IRIX.[5]

The decision as to whether sharable code at a given virtual address always accesses the same static data is essentially a matter of taste; we have adopted a philosophy more in keeping with Unix than with Multics. In code that is logically shared (with static data that is shared), Hemlock programmers can differentiate between processes on the basis of

- parameters passed into the code in registers, or in an argument record accessed through a register (frame pointer),
- return values from system calls that behave differently for different processes (possible only if processes are managed by the kernel),
- explicit, programmer-specified overloading of (a limited number of) addresses, or
- programming environment facilities (e.g., environment variables) implemented in terms of one of the above.

[5]We should emphasize that our system does not *require* PIC. In fact, the SGI compilers don't produce it yet. When it becomes available we will obtain no new functionality, but we will use less space.

Caveats

Easy sharing is unfortunately not without cost. Although we firmly believe that increased use of cross-application shared memory can make Unix more convenient, efficient, and productive, we must also acknowledge that sharing places certain responsibilities on the programmer, and introduces problems.

Synchronization

Files are seldom write-shared, and message passing subsumes synchronization. When accessing shared memory, however, processes must synchronize explicitly. Unix already includes kernel-supported semaphores. For lighter-weight synchronization, blocking mechanisms can be implemented in user space by providing standard interfaces to thread schedulers[13], and several researchers have demonstrated that spin locks can be used successfully in user space as well, by preventing, avoiding, or recovering from preemption during critical sections[2, 6, 13], or by relinquishing the processor when a lock is unavailable[11].

Garbage Collection

When a Unix process finishes execution or terminates abnormally, its private segments can be reclaimed. The same cannot be said of segments shared between processes. Sharing introduces (or at least exacerbates) the problem of garbage collection. Good solutions require compiler support, and are inconsistent with the anarchistic philosophy of Unix. We see no alternative in the general case but to rely on manual cleanup. Fortunately, our shared file system provides a facility crucial for manual cleanup: the ability to peruse all of the segments in existence. Our hope is that the manual cleanup of general shared-memory segments will prove little harder than the manual cleanup of files, to which programmers are already accustomed.

Position-Dependent Files

As soon as we allow a segment to contain absolute internal pointers, we cannot change its address without changing its data as well. Files with internal pointers cannot be copied with *cp*, mailed over the Internet, or archived with *tar* and then restored in different places. Though many files need never move, in other cases the choice between being able to use pointers and being able to move and copy files may not be an easy one to make. Figures

from our modified version of xfig, for example, can safely be copied only by xfig itself.

Dynamic Storage Management

In the earlier overview section, we suggested that dynamic linking might encourage widespread re-use of functional interfaces to pre-existing utilities. It is likely that the interfaces to many useful functions will require variable-sized data structures. If the text editor is a function, for example, it will be much more useful if it is able to change the size of the text it is asked to edit. This suggests an interface based on, say, a linked list of dynamically-allocated lines, rather than a fixed array of bytes. We have developed a package designed to allocate space from the heaps associated with individual segments, instead of a heap associated with the calling program. This package is used by the Hemlock version of xfig. We expect that as we develop more applications we will be able to determine the extent and type of new storage management facilities that will be necessary.

Safety

It is possible that a programming error will cause a program to make an invalid reference to an address that happens to lie in a segment to which the user has access rights. Our signal handler will then erroneously map this segment into the running program and allow the invalid reference to proceed. We see no way to eliminate this possibility without severely curtailing the usefulness of our tools. The probability of trouble is small; the address space is sparse.

It is also possible that a program will circumvent our wrapper, execute a kernel call directly, and replace our signal handler. Since use of our tools is optional, we do not regard this as a problem; we assume that a program that uses our tools will use only the normal interface.

Finally, programming under Hemlock using shared memory requires a more defensive style of programming than is normally necessary when communicating via messages or RPC. It is easier to implement sanity checks for RPC parameters than it is to implement them for arbitrary shared data segments. Servers must be careful that their proper operation is not dependent on the proper operation of their clients.

Loss of Commonality

The ubiquity of byte streams and text files is a major strength of Unix. As shared-memory utilities proliferate, there is a danger that programmers will develop large numbers of incompatible data formats, and that the ''standard Unix tools'' will be able to operate on a smaller and smaller fraction of the typical user's data.

Many of the most useful tools in Unix are designed to work on text files. To the extent that persistent data structures are kept in a non-linear, non-text format, these tools become unusable. Administrative files, for example, are often edited by hand. There are good arguments for storing them as something other than ascii text, but doing so means abandoning the ability to make modifications with an ordinary text editor.

It is not entirely clear, of course, that most data structures *should* be modified with a text editor that knows nothing about their semantics. Unix provides a special locking editor (*vipw*) for use on /etc/passwd, together with a syntax checker (*ckpw*) to verify the validity of changes. System V employs a non-linear alternative to /etc/termcap (the *terminfo* database), and provides utility routines that translate to and from (with checking) equivalent ascii text.

Similar pros and cons apply to the design of programs as filters. The ability to pipe the output of one process into the input of another is a powerful structuring tool. Byte streams work in pipes precisely because they can be produced and consumed incrementally, and are naturally suited to flow control. Complex, non-linear data structures are unlikely to work as nicely. At the same time, a quick perusal of Unix directories confirms that many of the file formats currently in use have a rich, non-byte stream structure: *a.out* files, *ar* archives, *core* files, *tar* files, TeX *dvi* files, compressed files, inverted indices, the SunView defaults database, bitmap and image formats, and so forth.

6. Conclusion

Hemlock is a set of extensions to the Unix programming environment that facilitates sharing of memory segments across application boundaries. Hemlock uses dynamic linking to allow programs to access shared objects with the same syntax that they use for private objects. It includes a shared file system that allows processes to share pointer-based linked data structures without worrying that addresses will be interpreted differently in different protection domains. It increases the convenience and speed of shared data management, client/server interaction, parallel program construction, and long-term storage of pointer-rich data structures.

As of November 1992, we have a 32-bit version of Hemlock running on an SGI 4D/480 multiprocessor. This version consists 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 in a dedicated shared file system, and (4) a fault handler that adds segments to a process's address space on demand, triggering the dynamic linker when appropriate.

Hemlock maintains backward compatibility with Unix, not only because we wish to retain the huge array of Unix tools, but also because we believe that the Unix interface is for the most part a good one, with a proven track record. We believe that backward compatibility has cost us very little, and has gained us a great deal. In particular, retention of the Unix file system interface, and use of the hierarchical file system name space for segments, provides valuable functionality. It allows us to use the traditional file read/write interface for segments when appropriate. It allows us to apply existing tools to segments. It provides a means of perusing the space of existing segments for manual garbage collection.

Problems that we are currently investigating include:

- Language Heterogeneity
  Hemlock uses the object file as a lowest common denominator among programming languages. It provides no magic, however, to ensure that object files produced by different compilers will embed compatible assumptions about the naming, types, and layout of shared data. These problems are not new of course; programs whose components are written in different languages, or compiled by different compilers, must already deal with the issue of compatibility. Problems are likely to arise more often, however, when sharing among multiple programs. We are interested in the possibility of automatically translating definitions of shared abstractions written in one language into definitions and optimized access routines written in another language.
- Synchronous Communication
  We plan to add a protection-domain switching system call to our modified IRIX kernel to support synchronous communication across protection boundaries in Hemlock. We speculate that the ability to migrate unprotected functionality into shared code will allow us in many cases to increase the degree of parallelism, and hence the performance, of fast RPC systems.
- Scoped Static Linking
  Because lds is implemented as a wrapper for ld, scoped linking is currently available in Hemlock only for dynamic modules. We plan to correct this deficiency in a new, fully-functional static linker.

Along with the above goals there are a number of other questions that we expect will be answered as we continue to build larger applications with Hemlock. These include:

- How important is the ability to overload virtual addresses? Is it purely a matter of backward compatibility?
- How best can our experience with Psyche (specifically, multi-model parallel programming and first-class user-level threads) be transferred to the Unix environment?
- To what extent can in-memory data structures supplant the use of files in traditional Unix utilities?
- 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 [26] 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.

References

1. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, ''Mach: A New Kernel Foundation for UNIX Development,'' *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, pp. 93-112, June 1986.

2. T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, ''Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism,'' *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 53-79, February 1992. Originally presented at the *Thirteenth ACM Symposium on Operating Systems Principles*, 13-16 October 1991.

3. B. N. Bershad, E. D. Lazowska, H. M. Levy, and D. B. Wagner, ''An Open Environment for Building Parallel Programming Systems,'' *Proceedings of the First ACM Conference on Parallel Programming: Experience with Applications, Languages and Systems*, pp. 1-9, New Haven, CT, 19-21 July 1988. In *ACM SIGPLAN Notices 23*:9.

4. B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy, ''Lightweight Remote Procedure Call,'' *ACM Transactions on Computer Systems*, vol. 8, no. 1, pp. 37-55, February 1990. Originally presented at the *Twelfth ACM Symposium on Operating Systems Principles*, 3-6 December 1989.

5. J. S. Chase, H. M. Levy, M. Baker-Harvey, and E. D. Lazowska, ''How to Use a 64-Bit Virtual Address Space,'' Technical Report 92-03-02, Department of Computer Science and Engineering, University of Washington, March 1992.

6. J. Edler, J. Lipkis, and E. Schonberg, ''Process Management for Highly Parallel UNIX Systems,'' *Proceedings of the USENIX Workshop on Unix and Supercomputers*, Pittsburgh, PA, 26-27 September 1988. Also available as Ultracomputer Note #136, Courant Institute, N. Y. U., April 1988.

7. W. E. Garrett, R. Bianchini, L. Kontothanassis, R. A. McCallum, J. Thomas, R. Wisniewski, and M. L. Scott, ''Dynamic Sharing and Backward Compatibility on 64-Bit Machines,'' TR 418, Computer Science Department, University of Rochester, April 1992.

8. M. Herlihy and B. Liskov, ''A Value Transmission Method for Abstract Data Types,'' *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 4, pp. 527-551, October 1982.

9. W. W. Ho and R. A. Olsson, ''An Approach to Genuine Dynamic Linking,'' *Software — Practice and Experience*, vol. 21, no. 4, pp. 375-390, April 1991.

10. E. Jul, H. Levy, N. Hutchinson, and A. Black, ''Fine-Grained Mobility in the Emerald System,'' *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 109-133, February 1988. Originally presented at the *Eleventh ACM Symposium on Operating Systems Principles*, Austin, TX, 8-11 November 1987.

11. A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki, ''Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor,'' *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 41-55, Pacific Grove, CA, 13-16 October 1991. In *ACM SIGOPS Operating Systems Review 25*:5.

12. 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.

13. 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*, pp. 110-121, Pacific Grove, CA, 14-16 October 1991. In *ACM SIGOPS Operating Systems Review 25*:5.

14. B. D. Marsh, C. M. Brown, T. J. LeBlanc, M. L. Scott, T. G. Becker, P. Das, J. Karlsson, and C. A. Quiroz, ''Operating System Support for Animate Vision,'' *Journal of Parallel and Distributed Computing*, vol. 15, no. 2, pp. 103-117, June 1992.

15. C. R. Morgan, ''Special Issue on the Interface Description Language IDL,'' *ACM SIGPLAN Notices*, vol. 22, no. 11, November 1987.

16. B. Nitzberg and V. Lo, ''Distributed Shared Memory: A Survey of Issues and Algorithms,'' *Computer*, vol. 24, no. 8, pp. 52-60, August 1991.

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

18. M. Rozier and others, ''Chorus Distributed Operating Systems,'' *Computing Systems*, vol. 1, no. 4, pp. 305-370, Fall 1988.

19. M. L. Scott, T. J. LeBlanc, and B. D. Marsh, ''Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System,'' *Proceedings of the 1988 International Conference on Parallel Processing*, vol. II – Software, pp. 255-262, St. Charles, IL, 15-19 August 1988.

20. M. L. Scott, T. J. LeBlanc, and B. D. Marsh, ''Evolution of an Operating System for Large-Scale Shared-Memory Multiprocessors,'' TR 309, Computer Science Department, University of Rochester, March 1989.

21. 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*, pp. 70-78, Seattle, WA, 14-16 March, 1990. In *ACM SIGPLAN Notices 25*:3.

22. M. L. Scott, ''The Lynx Distributed Programming Language: Motivation, Design, and Experience,'' *Computer Languages*, vol. 16, no. 3/4, pp. 209-233, 1991. Earlier version published as TR 308, ''An Overview of Lynx,'' Computer Science Department, University of Rochester, August 1989.

23. M. L. Scott and W. Garrett, ''Shared Memory Ought to be Commonplace,'' *Proceedings of the Third Workshop on Workstation Operating Systems*, Key Biscayne, FL, 23-24 April 1992.

24. D. Swinehart, P. Zellweger, R. Beach, and R. Hagmann, ''A Structural View of the Cedar Programming Environment,'' *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 4, pp. 419-490, October 1986.

25. J. H. Walker, D. A. Moon, D. L. Weinreb, and M. McMahon, ''The Symbolics Genera Programming Environment,'' *IEEE Software*, vol. 4, no. 6, pp. 36-45, November 1987.

26. 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.

27. W. A. Wulf, R. Levin, and S. P. Harbison, *Hydra/C.mmp: An Experimental Computer System,* McGraw-Hill, New York, 1981.

28. *Usenix Workshop on MicroKernels and other Kernel Architectures*, Seattle, WA, 27-28 April 1992.

Author Information

Bill Garrett is a graduate student in the Computer Science Department at the University of Rochester. He received his B.S. from Alfred Unviersity in 1990 and his M.S. from Rochester in 1992. He can be reached c/o the Computer Science Department, University of Rochester, Rochester, NY 14627-0226. His e-mail address is garrett@cs.rochester.edu.

Michael Scott is an Associate Professor of Computer Science at the University of Rochester. He received his Ph.D. from the University of Wisconsin – Madison in 1985. His U.S. mail address is the same as Bill Garrett's. His e-mail address is scott@cs.rochester.edu.

Ricardo Bianchini, Leonidas Kontothanassis, Andrew McCallum, and Bob Wisniewski are graduate students in the Computer Science Department at the University of Rochester. Jeff Thomas is a graduate student in the Computer Science Department at the University of Texas at Austin. Steve Luk is an undergraduate at the University of Rochester, majoring in Computer Science.