Dynamic Sharing and Backward Compatibility on 64-Bit Machines

William E. Garrett, Ricardo Bianchini,
Leonidas Kontothanassis, R. Andrew McCallum,
Jeffery Thomas, Robert Wisniewski, and
Michael L. Scott

TR 418
April 1992

University of Rochester
Computer Science Department
Rochester, NY 14627-0226

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 most multi-user operating systems are difficult 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 see a tremendous opportunity to extend the advantages of sharing across application boundaries. We believe that these advantages can be realized without introducing major changes to the Unix programming model. In particular, we believe that it is both possible and desirable to incorporate shared memory segments into the hierarchical file system name space.

Our approach has two components: First, we use dynamic linking to allow programs to access shared data and code in the same way they access ordinary (private) variables and functions. Second, we unify memory and files into a single-level store that facilitates the sharing of pointers. This second component is made feasible by the 64-bit addresses of emerging microprocessors.

This work was supported in part by NSF grants CCR-9005633 and CDA-8822724, and by Brazilian CAPES and NUTES/UFRJ fellowships. Authors' email addresses:
{garrett, ricardo, kthanasi, mccallum, thomas, bob, scott}@cs.rochester.edu.

1. Introduction

Memory sharing between arbitrary processes is at least as old as Multics [44]. 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 4.2 and 4.3 BSD releases [34]; it appears in several commercial systems, including SunOS [23] and IRIX. 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 [66] and Chorus [49] 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 [11, 20, 35, 36, 43, 47] 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 [32] and RPC stub generators [3, 26]).

(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 [6], 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. Some (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, reentrant 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.

Unfortunately, anecdotal evidence suggests that user-level programmers employ shared memory mainly for special-purpose management of memory-mapped devices, and for interprocess 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 in systems with more than one address space, since pointers may be interpreted differently by different processes. Any data object visible to two different processes must appear at the same virtual address from each point of view. Moreover, any two data objects simultaneously visible to the same process must have different virtual addresses from that process's point of view. If processes map objects into their address spaces dynamically (e.g. as a result of following pointers), the only general way to preclude address conflicts is to assign every sharable object a unique, global virtual address. Such *uniform addressing* requires a consensus mechanism that is not a standard part of existing systems. It also makes virtual addresses an extremely scare resource on 32-bit machines.

We believe that pointers are crucial for realizing the full potential of shared memory. We therefore adopted uniform addressing for in-core code and data in our earlier Psyche system [52, 53], arguing that the advent of 64-bit architectures would soon eliminate the scarcity of virtual addresses. With the recent release of microprocessors such as the MIPS R4000 and the DEC Alpha [17], we believe that uniform addressing can be adopted without hesitation for large, multi-user systems. Moreover, the truly enormous amount of space addressable in 64 bits makes it possible to extend uniform addressing into the file system, and to unify the entire memory hierarchy into an unsegmented single-level store.

In summary, we believe it is time for the advantages of memory sharing, long understood in the open operating systems community [48, 57, 60], to be extended into environments with multiple users and hardware-enforced protection domains. Such a move is particularly attractive on 64-bit machines, though much can be done to facilitate sharing even with shorter addresses.

From a practical point of view, we believe that dynamic sharing and uniform addressing can, and in fact should, be implemented in a backward-compatible fashion in systems such as Unix. We concur with Weiser et al. [62] that the proliferation of languages and protection boundaries in Unix will make it difficult to realize the full flexibility of the open system model. Short of this goal, however, we still see tremendous opportunity to make Unix more convenient and efficient through the exploitation of shared memory, without introducing major changes to the kernel, existing programs, or the (loosely-defined) Unix programming model.

We provide an overview of our approach in section 2, and a more detailed rationale and comparison to related work in section 3. We discuss implementation details in section 4, provide examples of the use of our tools in section 5, and conclude in section 6.

2. Overview

Our emphasis on shared memory has its roots in the Psyche project [51, 52]. 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 [37-40, 53]. Fundamental to this work was the assumption that sharing would occur both within and among applications. Our current work [55] 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 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 principal goal is to make cross-program sharing easy, while maintaining compatibility with existing Unix facilities. In the process, we expect to improve the performance of most applications that adopt shared memory as an alternative to messages or files.

An early prototype of our system ran under SunOS, but we are now working on Silicon Graphics machines (with SGI's IRIX operating system), in anticipation of 64-bit hardware and of compilers that generate 64-bit addresses. Throughout this paper, we use present tense for things we have already built, future tense for things we definitely plan to build, and conditional tense for things we aren't yet sure are desirable or for which we aren't yet sure we will want to expend the effort of implementation.

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 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.

2.1. Dynamic Linking

Our dynamic linking system associates 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 syn-

chronization) 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 linker for sharing, *lds*, is 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 in the single-level store.

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-agreed-upon virtual address (see section 2.2 below), and is internally relocated on the assumption that it resides at that address.

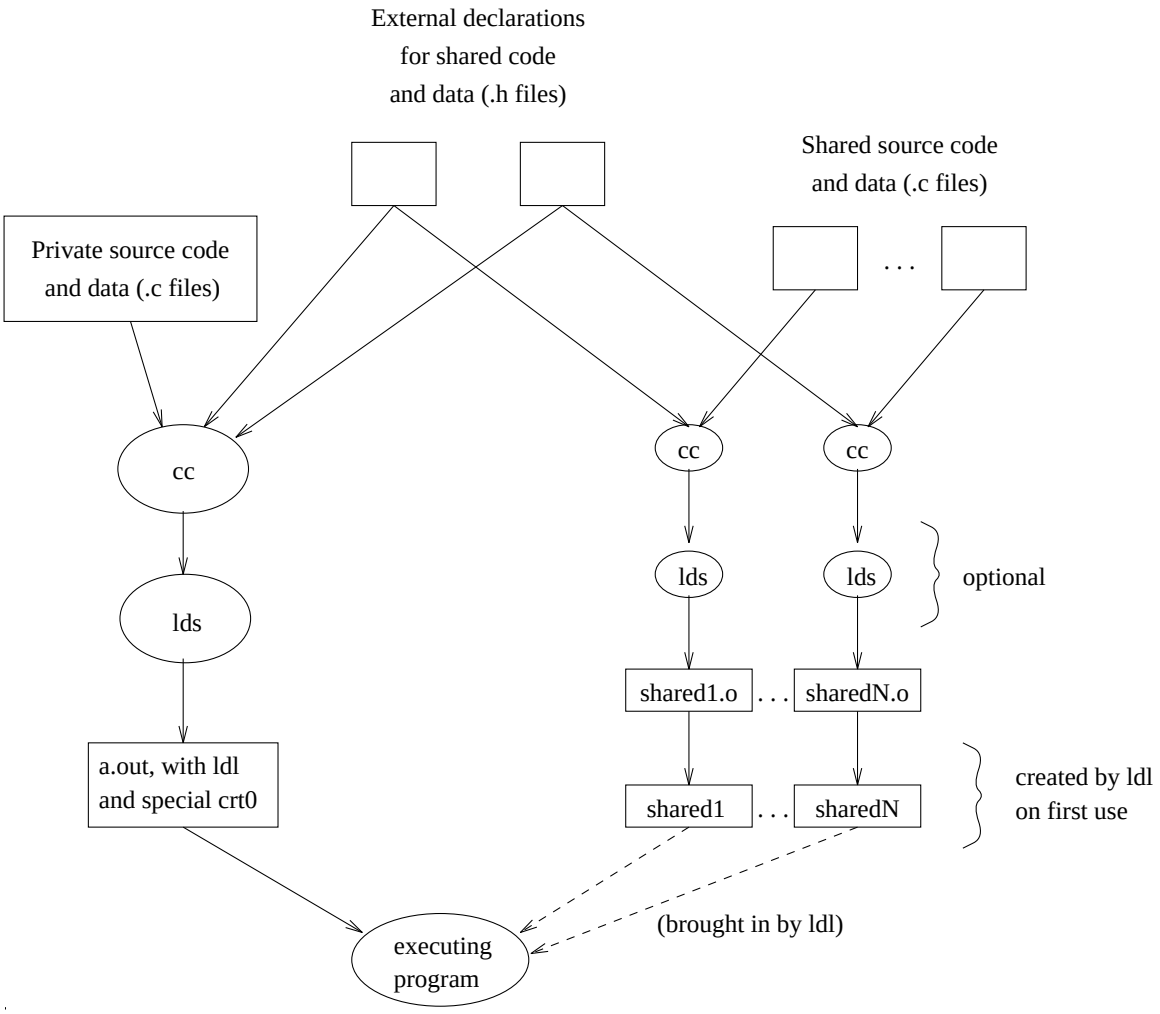

Figure 1: Building a Program with Linked-in Shared Objects

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 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, re-writing 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, then 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. (This mechanism subsumes the search path that ld uses for library archives; see the manual pages in the appendix for details). 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 (–r on most systems). 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. In a complicated program, resolution of external symbols can result in a directed acyclic graph (DAG) of module inclusions, with different search strategies at each level (see figure 2).

It is possible — expected, in fact — that large programs will have more than one symbol with the same name. Per-module search rules eliminate ambiguity in the resolution of references to these symbols. They preserve 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. We discuss this issue further in section 3.4.2.

It should be emphasized that modules of all classes appear in our single-level store (or will on a 64-bit machine), and are thus potentially sharable. The differences between public and

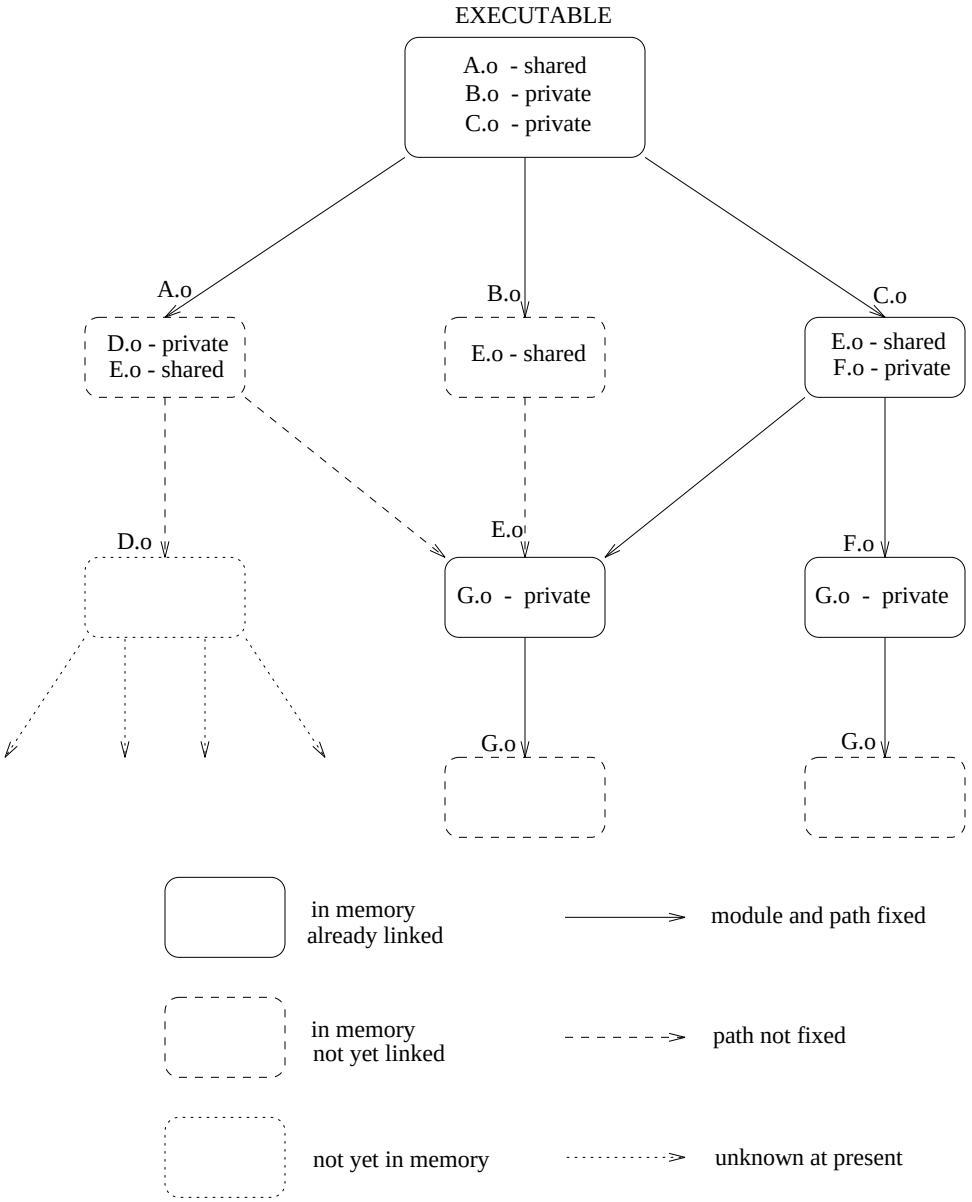

Figure 2: Hierarchical Inclusion of Dynamically-Linked Modules

private modules lie in (1) whether or not a new instance is created for each process,[1] (2) whether or not the module survives process termination, and (3) where it appears in the address space and file system (see table 1, page 21). Public modules are persistent; like traditional files they continue to exist until explicitly destroyed. (We consider the issue of garbage collection in section 3.2.) Private modules can be shared only by mapping in the file that represents the private space of another process; private modules are never *linked* into more than one protection domain.

[1] 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.

2.2. Single-Level Store

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 [30]), 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 1G byte region between the Unix heap and stack segments, and have associated this region with a dedicated ‘‘shared file system’’ (see figure 4, page 24). 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. With 64-bit addresses, we will extend the shared file system to include all of secondary store. 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. This ability is in contrast to the strict single-translation approach of systems such as Opal [12, 13]. We discuss this issue further in section 3.2; for the moment, suffice it to note that private modules (including the main module of every process) are linked into the private, overloaded portion of the address space, but public modules are linked at their globally-understood address.

The simplest way to allocate space in the single-level store would be to statically partition addresses into a file specifier and offset — 4G segments of up to 4G bytes each, for example. The fragmentation entailed by this partitioning is enormous, but it poses no implementation problems, and still leaves plenty of room for everything in a 64-bit space. If fewer than 64 address bits are significant, or if address bits are desired for other purposes (e.g. to extend addressing over a large network), then most segments should be created with a smaller maximum size, determined perhaps by Unix’s per-process file size *limit*.

As mentioned in the previous section, a user-level handler for the SIGSEGV signal catches references to modules that are not currently part of the address space of the executing process. This 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 uses a (new) kernel call to translate the faulting address into a path name and, if possible, to map 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. A program-provided handler for SIGSEGV, if any, is invoked when the dynamic linking handler is unable to resolve a fault.

Memory segments that traditionally have no file system name, such as private text, data, and stack segments, remain nameless in our prototype system. On a 64-bit machine, we plan to place them under a special ‘‘/proc’’ hierarchy in the file system, as suggested by Killian [30].

We have considered several possible organizations for /proc. It is difficult to evaluate these organizations, however; we doubt our ability to anticipate all the uses there could be for the

directory. When our 64-bit machines arrive, we are likely to start with a minimalist approach. Each process would appear under /proc as a directory named after the process's id. Within each directory, a single file, named after the executable being run, would contain the process's private modules (both static and dynamic). For each public module linked to the process, there would be two symbolic links, one to the module's template and another to the module itself, which resides in the template's directory. Fancier organizations for /proc might separate the text, data, and stack portions of the executable into separate files (allowing them to be protected individually), separate the various private modules into separate files, organize processes under per-user subdirectories (to facilitate browsing), or reflect the hierarchical order in which modules were linked.

For protection modes on files in /proc, we expect to rely on the user's Unix *umask*. A process can of course alter protections explicitly, using the existing file system interface. In general, we expect umask-based protections to be liberal — permitting write permission on text, execute permission on data, etc. For safety's sake, ldl maps pages with more appropriate, restrictive permissions. We have considered having ldl maintain /proc without kernel assistance, but such an approach has several drawbacks. Executables would have to be copied into /proc explicitly, at considerable expense. Programs that do not use our tools would not appear, and programs that terminate abnormally would not disappear. We would also have to implement a wrapper routine around *fork*, to make sure that child processes appear. As a result, we believe that kernel support for /proc will be required. The kernel can use copy-on-write techniques to avoid the overhead of physically copying large executables into the /proc directory. It can also ensure that all processes appear in /proc, and that space is cleaned up upon unexpected process termination.

3. Discussion and Rationale

Having provided an overview of our tool set, we now turn to a more detailed discussion of issues and comparison to related work.

3.1. Backward Compatibility

One of the major goals of our work is to maintain as much compatibility as possible with the Unix operating system. This goal stands in sharp contrast to the approach taken by much experimental systems research, our own Psyche project included.

It is always tempting to start with a clean slate. Decisions can be made purely on the basis of merit, without the temptation to conform to inferior but established conventions. Unfortunately, the clean slate approach makes it difficult to build up enough infrastructure to attract real users and accumulate meaningful experience. It also makes it easy to stray into side issues that contribute little to the main thrust of the project. We devoted four years to the Psyche project without approaching a production quality environment. We accumulated some genuine and valuable experience, but less than we had hoped. We also discovered when we were done that many of the issues that consumed time and energy early on (management of access rights, for example) were in hindsight unimportant.

Our current work is similar in spirit to Psyche. As a result, we believe we have a better understanding of where we are going at this point than we had when we started work on Psyche, and we are willing to forgo the freedom of the clean slate approach. By emphasizing backward compatibility,

(1) We expect to leverage a lot of existing software, and to cater to serious users. The heart of our tool kit is of course new and experimental, but the supporting environment is solid and pre-existing. Our modified IRIX kernel supports a large user community in daily production use on the department's main cycle server.

(2) We inherit a set of answers to peripheral questions. If we are uncertain which approach to adopt on a particular issue, or if the issue is not central to what we're doing, we can simply

leave it alone. Invocation of precedent has proven to be a useful aid to self-discipline, keeping discussions focused on issues that really matter.

(3) We retain a set of abstractions which, while certainly not perfect, have a proven track record of usefulness [46]. For issues on which we defer to Unix, the results will most likely be acceptable.

We should emphasize that we are not committed to *complete* compatibility with Unix. We consider ourselves free to make changes to anything that really needs changing. Our experience, however, has been that very little actually falls into this category; existing tools and conventions are remarkably compatible with both dynamic linking and uniform addressing.

Unix programmers are accustomed to building complex sets of tools. We hope to make those tools faster, smaller, easier to write, and easier to tie together. In keeping with this goal, we have attempted to *integrate* our tools with Unix, as opposed to building a new environment on top of Unix. This means that we impose as few restrictions as possible on what users can do while still taking advantage of our work. For example, we do not require that users program in a given language (only that their compilers generate standard format object files), nor do we forbid them from mapping different segments at the same virtual address in different protection domains (see section 3.2 below). These decisions have opportunity costs: we cannot count on compilers to implement protection, collect garbage, or generate self-descriptive data structures, nor can we pursue hardware optimizations based on the use of a single virtual-to-physical translation [31]. From our perspective, these costs are overshadowed by the benefits of backward compatibility.

3.2. Address Space Organization

Figure 3 contains a picture of our single-level store as it will appear on a 64-bit machine. As suggested in earlier sections, we will reserve a 32-bit portion of the 64-bit virtual address space for private code and data. Addresses in the private portion of the address space will be overloaded: different segments will appear at the same (private) address in different address spaces, and every segment in the private portion of an address space will also appear (with a different address) in the public portion of the address space.

The kernel maintains the mapping between file names and addresses in the single-level store. A process is free to *mmap* a file at some other address in the public portion of the address space, but we assume that reasonable processes will not do so. Ldl links private modules at addresses in the private portion of the address space and public modules at the appropriate address in the public portion of the address space. A process can determine the public address of a private object by calling a library routine. The routine uses a pair of kernel calls to (1) translate the private address into a file name and offset, and (2) translate the file name into a public address. In section 5.3 we explain how this mechanism can be used to facilitate the creation of parallel programs.

Every program begins execution in the private portion of the address space. Traditional, unmodified Unix programs never use public addresses. Our expectation is that programmers will gradually adopt a style of programming in which public addresses are used most of the time. Backward compatibility is thus the main (though not the only — see below) 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. [13] 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 our environment, however, and we retain it by weight of precedent.

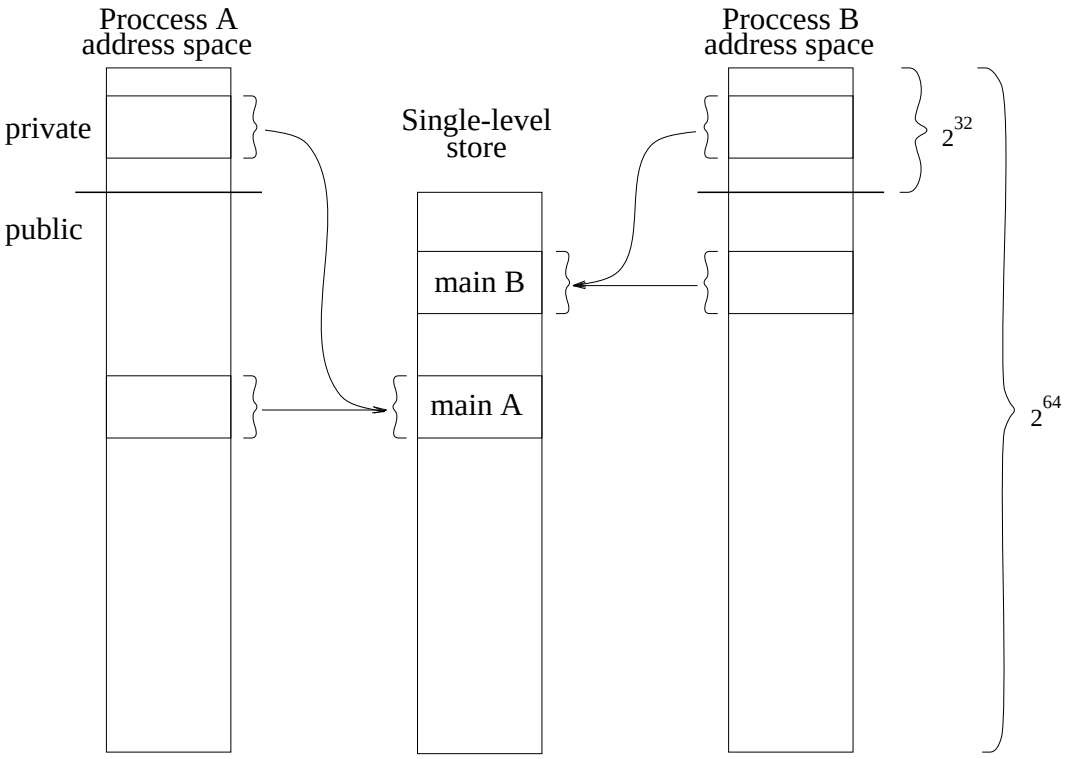

Figure 3: The Public/Private Address Space Organization

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.

It is not yet clear to us whether *all* uses of address overloading should be considered anachronisms. We adopted a single translation for in-core code and data in Psyche, but were not entirely happy with the result. We were forced, for example, to change the semantics of BBN's Uniform System library [58] in order to port it to Psyche.[2] We fear that other programs and programming environments (e.g. Lisp interpreters that use address bits for tags) may also insist on the ability to overload addresses. It is tempting to argue that all such programs are poorly designed, but we are hesitant to do so.

[2] The Uniform System provides a set of worker processes, one per processor, that share a central work queue. It specifies that static variables for each process lie at the same virtual address, so that when a pointer to a local data structure is passed to another processor, it will point to the corresponding data structure there. We were forced to place the data structures at different addresses in Psyche, and had to modify programs that expected the addresses to be the same.

3.3. Shared Code

The above discussion of *fork* raises an interesting issue: under what circumstances should code be considered to be shared? In traditional systems, the distinction between copying and sharing is irrelevant for read-only objects. The real issue for most programmers is whether static data and global variables referenced in a high-level language are private or shared. As in Psyche, we adopt the philosophy that code should be considered shared precisely when its static data is shared. Under this philosophy, the various implementations of ‘‘shared’’ 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 [44], Hydra [65], and Opal [13], 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 our 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 iff 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). In the basic sense of the term, PIC is code that embeds no assumptions (even after linking) about the address at which it executes. For our purposes, we also insist that PIC embed no assumptions about the addresses of static data or external code or data. Instead, references and calls to these objects are made *indirectly* through linkage tables (generally placed in adjacent pages and accessed via pc-relative addressing), so that different processes can place their data at different addresses, and can resolve external symbols differently. Compilers that generate this sort of PIC are already used for shared libraries in SunOS and SVR4, and will soon be available under IRIX. PIC can be used in either the private or public portions of our address space to share the physical code pages of module instances at different addresses.[3]

The alternative approach to compiler construction — using base registers to access static data — has the advantage that it leads trivially to the sharing of physical pages of code among logically-private instances of a module. It has the disadvantage (from our point of view) that it places logically distinguishable objects (code fragments that use different static data) at a single public address. We prefer to place these objects at different addresses. This preference is essentially a matter of taste; we have adopted the Unix philosophy, as opposed to the Multics philosophy.

---

[3] 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.

Base registers allow processes to share physical pages of code while accessing different static data, even if they are executing the code at the same virtual address. If we place logically distinguishable instances of a module at different addresses, however, then the linkage tables of PIC will also allow the processes to access different static data, without overloading addresses. (Base registers are likely to be faster, but the difference is not large, and would be offset in part by the overhead of base register maintenance in the procedure calling sequence.) Moreover if references to external code and data are to be resolved differently in different processes, then linkage tables will be necessary for this purpose anyway. The tables can be reached via base registers or, again, if module instances lie at different virtual addresses, via pc-relative addressing. In summary, assuming that public addresses are never overloaded, base registers are mandatory if processes executing the same code at the same address are to access different static data or resolve their external references differently; base registers are optional if processes executing the same code at the same address always use the same static data and resolve their external references the same way.

In code that is logically shared (with static data that is shared), programmers can still 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.

3.4. Naming

Naming is a central issue in the design of almost any system. In our work there are three main categories of names: file names, addresses, and symbols. File names are symbolic names for segments. Addresses are low-level names both for segments and for the objects they contain. Symbols are high-level names for objects. The relationship between file names and addresses is one of the principal issues in the design of our single-level store. The binding of symbols to addresses is the central task of the linkers.

3.4.1. Naming in the Single-Level Store

A primary concern when proposing methods of sharing between programs is that they should be easy. 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.

Our system unifies file names and addresses, allowing both to be ‘‘first class.’’ The single-level store associates an address with a file at file creation time. Programs using our linkers are guaranteed that these addresses are the ones at which modules will be mapped and linked. Furthermore, programs are guaranteed that they can follow pointers into segments even when they contain no symbols, and have not been linked. Our signal handler catches segmentation faults in executing programs; if the faulting address exists in the single level store, and the process has

appropriate permissions, the segment is brought in to the user's address space and the faulting instruction is restarted.

Introducing a strong connection between file names and addresses has its complications. Traditional Unix file systems support aliases in the form of extra ''hard'' links and ''soft'' (symbolic) links. In order to map from addresses to file names, we must identify one alias as canonical. Several approaches are possible; in our 32-bit prototype, we allow only one hard link per file, and map the file's address to that link's full path name.[4] When fielding an access fault, our signal handler calculates the closest common ancestor of a target file and the current directory. It then checks access permissions from the current directory and from the root, and maps the file if either path works.

Traditional file systems support a *huge* name space for files — much larger than can be represented in 64 bits. As a result, the mapping between file names and addresses cannot be static; we have to assign addresses to file names on the fly. We need to be careful, however. If a file is deleted by accident, and then restored from tape, we would like to put it back at its previous address. We would also like to put it back at its previous address if it is deleted temporarily in the process of being updated (this happens, for example, when checking files in and out of *rcs* [59]). Our solution is to remember the addresses of deleted files, and to be careful about when we re-use addresses.

When a file is removed, the kernel remembers its name and address. At file creation time it checks to see whether the name of the new file matches the name of a file that was recently deleted. If so, it places the new file at the same place as the old one. It attempts to hold off assigning new names to old addresses by allocating addresses in a circular fashion. Addresses associated with deleted files are put at the end of the loop.

In our prototype system, 1M byte address ranges correspond one-to-one with Unix inodes. Creation of a new inode is the same as allocation of space out of our single level store (see section 4.4). Since we have a limit of 1024 shared files, we can afford to remember associations between addresses and deleted files forever. In a full single-level store, we would need to place a time limit on this memory. We plan to exclude /tmp and /proc; files created in these directories are meant to be transient. As a result, associations can probably be guaranteed to last for days or weeks: Ousterhout et al.'s (admittedly dated) study of file access patterns [45] reports a rate of only about 500 file creations and 500 deletions per hour, even including /tmp.

A related problem occurs when renaming files. Given that the address of a file is closely associated with its name, should a ''move'' operation place the file at a new address, or should it just change the name and leave the address the same? Our prototype currently leaves the address the same, but this is simply an artifact of our association between addresses and inodes. We plan to overload the arguments to the *rename* system call to make both options available, and are currently debating which should be the backward-compatible default (it may not matter much, so long as we re-write the *mv* command to support both options, since existing programs do not rely on the single-level store). Leaving the address the same is certainly cheaper, particularly for directories; changing the address of everything under a given directory would require time proportional to the size of the file system subtree. Of course, leaving the address the same interferes with the ability to re-create files at the same address, but this can be handled in the same ways that Unix currently deals with attempts to create a file with the same name as an existing file.

---

[4] We also considered giving every hard link its own address. A segment with more than one hard link could be accessed at more than one address. This would be easy to implement, but we're not sure we like its semantics.

3.4.2. Naming in the Linkers

Naming is also important with regard to how the linkers work. 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 else 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. Our linkers allow 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 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. Various programming languages (e.g. Modula-2 and Common Lisp) that use the idea of a module for abstraction already deal with this problem. They typically preface variable and function names with module names, thereby greatly reducing the chance of naming conflicts. One of our goals, however, has been to avoid the need to depend on any particular programming language or paradigm. Our system should allow recursive, dynamic linking no matter what the language. We achieve this goal by adopting a convention that removes the ambiguity present in the static linkers. 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 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 (figure 2, page 8), 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.[5]

3.5. Caveats

Sharing and addressing are not without cost. Although we firmly believe that increased use of addressable 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.

---

[5] Some Lisp environments respond to this sort of fault by prompting the user interactively for the location of a missing function; we could easily do the same.

3.5.1. 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 [37], 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, 18, 37], or by relinquishing the processor when a lock is unavailable [8, 29].

3.5.2. 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 single-level store 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. We do not believe that manual cleanup would be viable without a perusal mechanism.

3.5.3. Position-Dependent Files

Addresses for files are both a blessing and a curse. They introduce the problems with file restoration and renaming discussed in section 3.4.1. More important, they inherently lead to position dependence. 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.

In many cases, we expect that position dependence will not be a problem. Many files never need to move. In other cases, however, the choice between being able to use pointers and being able to move and copy files may not be an easy one to make. In section 5.4, for example, we consider the use of pointers in files that represent graphical figures. By storing figures in their linked internal format, we can eliminate substantial amounts of code devoted to translation when reading and writing. At the same time, we lose the ability to treat a figure file as a self-contained, position-independent object.

Those who wish to retain position independence can of course choose not to exploit the addressability of files. As in the past, they can translate their data structures to and from a linear external form, or use relative pointers (offsets) rather than absolute addresses. In a language with good abstraction mechanisms (C++ for example), relative pointers can be as convenient as regular pointers, though certainly not as fast.

Alternatively, programmers can write application-specific routines to move and copy segments containing pointers. If one were to create descriptors that identify the locations of pointers, a general-purpose routine might also be able to perform appropriate translations when moving or copying a segment. Many Unix compilers produce .o files (templates) containing information of this sort when invoked with a debugging flag (usually ‘–g’); we plan to investigate the extent to which this information suffices to move and copy segments created from the template.

3.5.4. Dynamic Storage Management

In section 2.1 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-size 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 will almost certainly need conventions for allocating space from heaps associated with individual segments, instead of a heap associated with the calling program.

3.5.5. 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. Examples include *awk*, *sed*, *tr*, *diff*, *grep*, *sort*, *more*, *tail*, *lex*, *yacc*, and the various editors. To the extent that persistent data structures are kept in a non-linear, non-text format, these tools become unusable. In section 5.1, we consider such files as /etc/passwd, /etc/hosts, and /etc/termcap. Each of these files is edited by hand. There are good arguments for storing them in 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.

In the long run, one might consider new conventions for self-descriptive data structures, polymorphic utilities, or mechanisms to support incremental creation and consumption of complex data structures. In the short run, it seems unlikely that the ability to more easily manipulate complicated data structures will compromise the elegance of Unix. Our intent is not to argue that non-linear data structures are bad; clearly we believe that their benefits often outweigh their costs. We simply point out that the functionality we provide has limitations, and should be adopted with care.

3.6. Related Work

3.6.1. Open Operating Systems

Much of the motivation for our work stems from the literature on open operating systems. Open operating systems can be characterized by compiler provided protection, and a highly modular organization allowing easy customization. They provide a very simple and comprehensive mechanism for sharing. A variable or function may be used regardless of the module in which it was defined simply by prefixing its name with that of the associated module. For a single-user system, openness offers two compelling advantages: flexibility and efficiency. Flexibility stems from the opportunity to modify, invoke, or build upon existing pieces of code. Efficiency stems from the lack of heavyweight context switches, data movement across narrow

interfaces, or unnecessary layers of abstraction. The incredible productivity of Lisp environments such as Genera [60], and of the Pilot [48] and Cedar [57] projects at Xerox PARC, testify to the usefulness of open systems. Clark's experience with Swift [15] testifies to their efficiency. Open systems have attracted the attention of language designers as well [9, 64], and are an important part of the commercial market for personal computers.

Unfortunately, the flexibility and efficiency of open operating systems is obtained at a serious price. Protection is available only to the extent that it is provided by high-level language compilers. With multiple compilers (as on a commercial PC) there is no protection whatsoever. With a single available compiler there is little opportunity to use much pre-existing software. There is also no opportunity to employ programming styles or paradigms unsupported by the programming language. For example, one is generally unable to program simultaneously in Cedar, Smalltalk, and Lisp within a single open system. Our work is an attempt to provide some of the benefits gained in an open operating system while still providing a standard means of protection.

3.6.2. Shared Memory Operating Systems

Several other projects have attempted to encourage sharing in multi-user systems. Multics [44] and Hydra [65] are probably the best-known examples. Both provide a single-level store, but were implemented on narrow-address machines. Multics employs a segmented address space and relies on elaborate hardware addressing modes and procedure calling sequences to maintain segment registers and tables. Hydra employs capabilities that are interpreted by the kernel. Because of hardware limitations, neither is able to provide machine-readable pointers that are globally meaningful.

More recently, researchers at the University of Washington have designed a system called Opal that provides a single-level store on 64-bit machines [13] and supports a user-level object system [12]. By adopting a single, global virtual-to-physical mapping, Opal is able to realize all the advantages of cross-address-space sharing. As discussed in section 3.3, there are at least two approaches to the sharing of code that appear to be compatible with both our kernel and Opal's; at the moment, the two projects are pursuing different approaches. The projects also differ in that we are attempting to maintain compatibility with Unix, while Opal is free to start fresh. Among other things, our interest in backward compatibility has prompted us to permit the overloading of virtual addresses in a limited portion of the 64-bit space, and to provide our memory segments with symbolic names in the Unix file system hierarchy. This latter decision provides us with a rich set of ready-made tools for perusal and management of long-lived segments.

3.6.3. Software Implementation of Large Address Spaces

Many distributed systems provide large name spaces with software interpretation. Some provide names for heavyweight objects [1, 16]; others for communication ports [14, 42]; still others for mappable segments [33]. In any case, objects in the distributed name space cannot be accessed via hardware addressing modes (except perhaps with a temporary mapping), and must generally be treated differently from addressable local objects.

In a few systems, compiler support has been integrated with a sophisticated run-time environment to provide the appearance of uniform naming in a very large name space. The LOOM system [28] implements a 32-bit Smalltalk environment on a 16-bit machine. The Emerald system [27] provides a uniform object model on a distributed network of machines.

The term *pointer swizzling* is used to describe systems that transparently manage two distinct sets of addresses for objects: long addresses on secondary storage, and short addresses in main memory. Wilson [63] has observed that by implementing swizzling in the kernel's virtual memory system, it is possible to maintain consistent short addresses for all processes on the machine with a high degree of efficiency. When bringing a page into memory, Wilson's system

assigns virtual addresses to all pages referenced by pointers in the new page, but does not make these additional pages accessible until the pointers are actually followed. This technique was the inspiration for our lazy linking mechanism. Unfortunately, Wilson's full system requires compiler support to identify all pointers, and requires that all pages in memory be marked invalid before any of them can be ''unswizzled'' back to secondary store.

3.6.4. Use of Shared Memory to Improve Performance

Several previous projects have explored the use of shared memory to improve the performance of cross-address-space process interactions. Bershad [6] uses buffers mapped into sending and receiving address spaces to speed the implementation of intra-machine remote procedure calls. The implementation of the Lynx distributed programming language for the BBN Butterfly employs a similar optimization [50]. Mach [66] uses copy-on-write page sharing to optimize intra-computer data transfers, and its external pagers can be used to facilitate data sharing between processes.

3.6.5. Generalization of the Unix File System Interface

Our assignment of names to shared segments is one in a long series of new uses for the file system naming hierarchy. Pseudo-terminals and Unix domain sockets are now commonplace, but were missing in early versions of Unix. Bershad and Pinkerton [4] describe a general-purpose mechanism for installing a user-provided process that intercepts operations on a specified file. Gifford et al. [22] describe a similar mechanism that allows the user to simulate files that are created on demand.

Our plans (section 2.2) for a directory containing images of running programs were directly inspired by Killian [30]. In his system, the address space of each process appears in the file system as /proc/*nnnnn*, where *nnnnn* is the process id number. These files can be inspected and modified with the usual file access functions, *lseek*, *read* and *write*. In addition, some *ioctl* calls allow commands such as stop/go and signal interception. Currently we have no plans for process control via ioctl calls, but nothing we have specified so far prohibits the possibility of implementing them. Like Killian's /proc, our version provides a place in the file system for previously unnamed memory objects (such as the process heap and stack). Unlike Killian's flat name space, our structure is likely to be hierarchical, with directories for processes and separate file system names for a process's loaded public modules.

3.6.6. Dynamic Linking

Dynamic linking is already a part of several Unix systems. It is used 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. Under SunOS, for example, *ld* will arrange by default for load-time linking of library routines. Position-independent code (PIC) permits the text 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. Ld also insists that all dynamically-linked libraries exist at static link time, in order to verify the names of their entry points.

Our system 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. 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.

Both SunOS and dld [25] 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 routines (*dlopen* and *dlsym*) do not provide this capability; they require that the newly-loaded module be self-contained. Neither dld nor the explicitly-invoked Sun routines will resolve undefined references in the main program; they simply return pointers to the newly-available symbols.

Our use of dynamic linking is reminiscent of several other systems. Dynamic linking is an integral part of single-user open operating systems such as Cedar [57], and has been implemented under Unix as part of such self-contained environments as Emerald [27] and the Portable Common Runtime [61]. The CLAM user interface system [10] and SOS distributed object system [21] load C++ classes dynamically; the latter is based on the Andrew project's *Camphor* dynamic linker. Our work differs from these projects in its use of dynamic linking to share potentially writable objects transparently, between ordinary Unix programs.

4. Implementation Details

4.1. Static Linker

Our dynamic linker requires more information than the standard IRIX static linker, ld, provides. Lds is a wrapper for ld which provides this information. Therefore, in addition to the options specific to lds, the programmer should pass to cc the desired arguments for ld. Lds processes the options directly related to its functionality and passes the others to ld. Lds-specific options allow for the association of sharing classes with modules (–F*sharing class*) and the specification of search paths to be used when locating modules (–Q*search path*). In addition, lds provides ldl with relocation information about static modules and warns the user if the dynamic modules do not yet exist. The following paragraphs explain in greater detail the functionality provided by lds.

The four sharing classes implemented by lds are static private, dynamic private, static public, and dynamic public (see table 1). Modules specified as static are linked together during the static link phase. Modules specified as dynamic are lazily linked at run time. Public modules are persistent; that is, them continue to exist after the program has terminated.

sharing class	when linked	new instance created/destroyed for each process	default portion of address space	location in single-level store
static private dynamic private	static link time run time	yes	private	/proc
static public dynamic public	static link time run time	no	public	same directory as template

Table 1: Sharing Classes

Lds verifies the existence of modules. Different search rules are used at static link time and at run time. At static link time, lds searches for modules in (1) the current directory, (2) the path specified by the −Q option, (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. 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. 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 by the −Q option at static link time, the directories specified by the LD_LIBRARY_PATH variable at static link time, and the default directories. Users can therefore arrange to use new versions of dynamic modules (e.g. for debugging) by changing the LD_LIBRARY_PATH environment variable immediately prior to execution. (This search algorithm was inspired by the analogous algorithm in the SunOS dynamic linker.)

The IRIX ld linker does not retain relocation information when statically linking an executable program. Lds must therefore take additional steps to provide ldl with relocation information for static modules. Lds extracts this information from the individual object files and creates a data structure that is passed to the lazy linker by depositing it in a new C file, compiling this file, and including it in the list of arguments to ld. Lds also uses this C file to inform ldl of the names of the dynamic modules and the search path.

When a module is linked using the static public sharing class, lds needs to provide some of the functionality of ldl. Public modules have a fixed address for each symbol. Since the linker provided by IRIX was not designed to handle fixed addresses, lds must handle them itself. Before passing a private module on to ld, lds resolves each reference to a static public symbol with an absolute address. In addition, lds must create any static public modules that do not yet exist, and must initialize those objects from their templates.

4.2. Dynamic Linker

Our early work used Ho's [25] dynamic linker *dld* on the Sun Sparcstation. After moving to the SGI it became apparent that many of the features we were planning to implement would require a major restructuring of dld. This restructuring, together with differences in object file format and relocation classes, prompted us to design our own lazy dynamic linker (ldl) for the SGI.

A small change to crt0 ensures that ldl runs prior to the start of any program written in C. Automatic invocation of ldl in languages other then C would require similar modifications to the corresponding start-up file. Ldl maintains a dynamic data structure for each dynamically-linked private or public module. This data structure contains relocation information and information on symbols exported by the module. As described in section 2.1, these modules are brought into the executing program on demand.

The most readily apparent difference between ldl and dld is that ldl resolves undefined external references in its host program while dld does not. Ldl is also more transparent then dld, although a dld-like interface for the advanced programmer is also provided (see the ldl man page in the appendix for more details).

Several difficulties in the implementation of ldl arose from inadequate documentation on the internal workings of the SGI compiler/linker system. SGI uses an object file format based on the System V standard Common Object File Format (COFF), but with modifications for SGI-specific relocation structures. We made guesses in several cases about the use of these structures, based on an examination of C compiler output. As a result, ldl turned out to be dependent on the behavior of the C compiler, and stopped working after installation of a new release of IRIX. We

have now re-written ldl to reduce the number of assumptions it makes about the compiler; we expect that our current version will be much more resistant to future changes.

Two incompatibilities between ldl and the SGI relocation system still exist at present. SGI uses indirection through a global pointer to reach objects in a special section of memory (.sdata and .sbss in figure 4, page 24) in a single machine instruction. By default SGI compilers try to put static data of 8 bytes or less into this special section. Ldl makes accessing information through this pointer impossible; dynamic linking fragments the address space, so initialized data is not contiguous. Luckily SGI compilers provide a flag to disable the global pointer. Ldl cannot dynamically link old objects compiled without the flag.

A second incompatibility is potentially more serious. The primary jump instruction used on the SGI uses a 28 bit offset from the current program counter. Address space fragmentation again makes it likely that some jumps will be resolved to locations more than 28 bits away. Even in our 32-bit prototype, such jumps occasionally occur. When ldl detects one, it currently prints an error message and halts execution. We have devised but not yet implemented a solution based on patching in branches to 32-bit jumps.

The only unusual kernel support required by ldl is a mechanism to control lazy linking by protecting and unprotecting pages. The *mprotect* system call provides this capability on the Sun, but is not yet available on the SGI. As a temporary measure, we have modified IRIX to leave the text segment of the main program writable. We unmap and remap entire dynamic segments for lazy linking. Given adequate documentation, it should be possible to port ldl to other versions of Unix with relative ease.

4.3. Signal Handler

When ldl maps a segment that contains undefined references, it maps the segment without access permissions. When (and if) such a segment is first accessed, a SIGSEGV signal (segmentation fault) occurs. We have implemented a handler that catches such signals, calls ldl to resolve any undefined references in the segment, and then restarts the faulting instruction. This handler is the runtime interface to ldl that makes lazy linking work. It is installed by our alternate version of crt0, using the BSD reliable signal interface, and exists for the lifetime of the process.

One complication with this implementation is that programs that employ our tools may wish to define their own SIGSEGV handlers. We would like to ensure that our handler catches all and only the SIGSEGV signals that occur in the process of lazy linking.

For programs using our tools, lds links in a wrapper for the BSD *signal* routine. Calls to install handlers for all signals other than SIGSEGV pass directly through to the old routine, which calls into the kernel as always. Calls to install handlers for SIGSEGV are redirected, and never make it to the kernel; the wrapper simply records them and returns. When a SIGSEGV signal occurs, the handler installed by crt0 checks to see if the signal resulted from an attempt to access a segment that ldl mapped in without permissions. If so, it calls ldl to resolve any undefined references in the segment, remaps it with permissions, and then restarts the faulting instruction. Otherwise, it invokes a user-defined handler, if any, or performs the default action (terminate the process and generate a core file) if the program has not defined its own handler.

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 64-bit address space is sparse.

It is also possible that a program will circumvent our *signal* 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.

4.4. Shared File System

When moving to a wide-address microprocessor it becomes feasible to give unique addresses to disk locations, allowing memory to be treated as a fast cache for the file system. This scheme allows files to contain pointers to other files, extending the programming paradigm in a number of useful ways (see section 5).

As mentioned in section 2.2, our 32-bit prototype system uses a dedicated disk partition as a ‘‘shared file system,’’ which the kernel associates with a 1G byte region lying between the bss and stack segments of a typical IRIX process (see figure 4). For the sake of simplicity, 30-bit addresses in the shared file system (obtained by subtracting 0x30000000 from the virtual address) are interpreted as a 10-bit inode number and 20-bit offset. The file system is configured to have exactly 1024 inodes, and the tail of any file longer than 1M bytes is not addressable.

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. Again for the sake of simplicity, this call employs a linear lookup table. We initialize the table at boot time by scanning the entire shared file

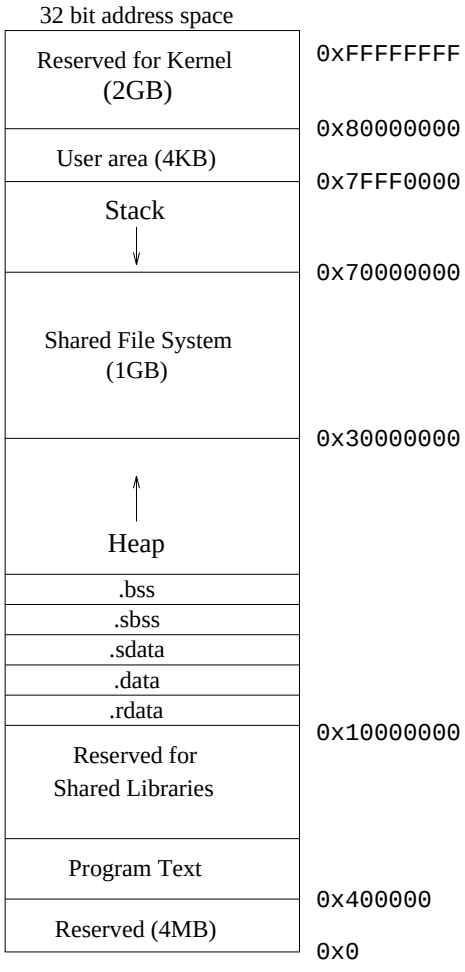

Figure 4: Layout of the 32-bit Address Space (not to scale)

system, and update it as appropriate when files are created and destroyed. To simplify the recovery of addresses when a deleted file is restored, we plan to pre-allocate all inodes to dummy files, circumventing the normal disk space management routines. 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.

In the 64-bit version of our system, we expect to 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.

4.5. Summary of Kernel Modifications

Our 32-bit prototype system currently includes the following kernel changes:

- The boot-up procedure scans the shared file system to construct the address-to-file-name lookup table.
- The various file creation and destruction system calls are special-cased to use our inode allocation algorithm on the shared file system, to maintain the address-to-file-name table, and to remember and re-use the addresses of recently-deleted shared files. These calls include *creat*, *unlink*, *mknod*, *mkdir*, *rmdir*, *rename*, and *symlink*.
- The *link* system call refuses to work on the shared file system.
- A new system call, *iname*, translates an address to a path name.

We are in the process of adding the following changes to our prototype:

- The arguments to *open* will be overloaded to allow the user to open a file by address. Doing so will be equivalent to, but faster than, calling *iname* followed by *open*.
- The arguments to *rename* will be overloaded to allow the user to specify whether or not the file should retain its address.
- The arguments to *creat* will be overloaded to allow the user to request file creation at a particular address (e.g. for restoration from backups after the kernel has forgotten the file's address).

Both an address-retaining rename and creation at a specified address will conflict with the guaranteed recovery of addresses when re-creating a deleted file. We will probably specify that a file with a new name can be placed at a remembered address only if the user would have been able to overwrite the deleted file.

Serendipitously, *fork* already creates a new copy of every private *mmap*ed segment, and shares all public *mmap*ed segments. We expect that the *mprotect* system call will appear in a future release of IRIX, and that the SGI compilers will be able to generate PIC.

We plan to add the following to the 64-bit version of our system:

- /proc file system (see section 2.2).
- Addresses for all files, embedded in on-disk structures.

5. Example Applications

In this section we consider several examples of the usefulness of cross-application shared memory. We examine, in turn, the various Unix administrative files, utility programs that might be re-cast as functions, parallel applications, and programs with complicated persistent data structures. One of these examples was implemented under the early version of our tools under SunOS; some of the others are currently under construction on the SGI; the rest are hypothetical.

5.1. 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.[6] 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. We are currently porting the new server and utilities to our SGI-based system.

Our modified version of the Sun *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 per hour on each Sparcstation 1. *Rwho* saves over a second each time it is called. Many members of our department run a variant of *rwho* automatically every 60 seconds, so this time may be significant.

5.2. Utility Programs and Servers

Traditionally, UNIX has been a fertile environment for the creation and use of small tools that can be connected in different ways. One of the great attractions of UNIX is the way in which shell users can employ pipes to create powerful combinations of these building blocks on the fly. These combinations can be viewed as a kind of functionality sharing at the level of program executables. Our system encourages sharing at the level of functions as well as executables.

Calling a utility routine as a function has several potential advantages over *exec*ing it as a program. It is certainly going to be faster in the cases where protection is not required, since the overhead of process creation is avoided. Calling a function also has the advantage of allowing arguments to be passed as genuine data structures, rather than as character strings, which is also good for speed. With start-up overhead eliminated, one could imagine putting the power of *emacs* behind every fill-in blank in a pop-up widget. Instances of the editor located behind different windows could share per-user data structures, allowing key bindings, macros and other

---

[6] At the University of Rochester, utilities have been written to correlate the data in *rwhod*’s files with information on host types and locations, phone numbers, etc., and to filter or threshold the information for individual users, hosts, load averages, etc.

customizations established in one window to be used in another. Alternatively, one could construct the editor as an explicitly parallel program, into which multiple processes would make concurrent calls. In contrast to the ‘‘shared’’ libraries of SunOS, our tools use the same .o file format for both static and dynamic linking, and make it easy for programmers to mix the use of private and shared data.

The simplest way to use a function that someone else has written is to include its header files in the new source, call it by name, and count on the dynamic linker to incorporate it into the running program. Our lazy linking and hierarchical search mechanisms support an arbitrary chain of such incorporations, without fear of naming conflicts. One could also imagine writing a shell that calls programs as functions; it would use the programmable, non-automatic interface to ldl. For backward compatibility or additional flexibility, one could construct a driver for each utility function that would allow it to be executed as a stand-alone program.

As instances of utility routines begin to share data structures or to be realized as concurrent code, the distinction between utilities and servers blurs. Just as function calls provide a faster and richer interface than *exec* for the invocation of utilities, so too are they attractive in comparison to message passing as an interface to servers. Clearly there will be circumstances in which a call to a server requires a change of protection domain, but there are also likely to be circumstances in which calls to servers can execute completely in the domain of the caller, with no overhead for parameter copying or context switching. Calls of this sort are reminiscent of *semi-model* operations in Psyche [38, 40]; the interface to a Psyche server consists of subroutines that transfer into the server’s domain only when one of a well-defined set of conditions makes such a transfer essential.

Even when a call into a server requires a change of protection domain, sharing between the client and server can make the call much faster. In their work on lightweight and user-level remote procedure calls, Bershad et al. argue that the speed of optimized interfaces permits a much more modular style of system construction than has been the norm to date [6, 7]. The growing interest in *microkernels* [67] 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.

5.3. 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 [5], 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 that were 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.[7] When the computation terminates the parent process performs the necessary cleanup, deleting the shared segment, template symlink, and temporary directory.

5.4. 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 [32, 41].[8] Similar work has occurred in the message-passing community [24].

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 [54, 56], designed around scanner and parser generators developed at the University of Wisconsin [19]. The Wisconsin tools produce numeric tables for separately-developed drivers. They come with drivers written in Pascal. At startup, these drivers read the tables from files. They translate them into appropriate data structures, and then begin to parse the user's program.

The cost of this initialization (during every compile run) prompted us to modify the drivers used in the Lynx compiler. Two auxiliary programs, *makescan* and *makeparse*, translate the scanner and parser tables into initialized C data structures that mirror the Pascal data structures that the drivers formerly built at start-up time. The compiler's *makefile* runs the output of makescan and makeparse through the C compiler and links it into the Lynx compiler via separate compilation.

Unfortunately, since Pascal lacks initialized static variables, the separate compilation trick depends on a non-portable, unsupported, and undocumented correspondence in data structure layouts between C and Pascal. Moreover, since makescan and makeparse are written in Pascal, they create exactly the data structures the Lynx compiler ought to contain; dumping those structures in C and then re-compiling the dump is a complete waste of time.

With the toolkit described in this paper, makescan and makeparse would share a persistent module (the tables) with the Lynx compiler. Makescan and makeparse would initialize the

---

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

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

tables; the compiler would link them in and use them. These changes would eliminate about a fifth of the code in makescan and quarter of the code in makeparse, both of which are about 400 lines long. 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. While editing, xfig maintains a set of linked lists that represent the objects comprising a figure. When saving figures to a file, it generates a pointer-free representation in ascii. About 1000 lines of code are dedicated to flattening the linked lists (when writing figures to disk) and to unflattening files back to linked lists (when reading figures from disk). We plan to adapt xfig to our system. It already includes code to copy the internal linked-list version of a figure. Once a segment has been mapped into xfig's address space, this same code can be used to read and write the figure in long-term storage, allowing the current input/output routines to be eliminated.

Storage of figures in long-lived segments will require that each segment contain a local heap. In addition, as noted in section 3.5.3, segments containing list-based figures will be position dependent. With the current version of xfig, users sometimes copy a file in order to initialize a new, similar figure. This will no longer be possible. Xfig itself, of course, can be used to make copies of figures.

6. Conclusion

We have proposed a set of extensions to the Unix programming environment that facilitate sharing of memory segments across application boundaries. We use dynamic linking to allow programs to access shared objects using the same syntax that they use for private objects. We exploit the availability of 64-bit architectures to assign a unique address to every byte of storage in primary and secondary memory, allowing processes to share pointer-based linked data structures without worrying that addresses will be interpreted differently in different protection domains. Our tools increase 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 April 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 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. Our SGI machine is scheduled to be upgraded to 64-bit R4000 processors in the fall of 1992, at which point we will extend our system to include all of secondary store.

Our tools maintain 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 do not believe that backward compatibility has cost us anything of importance, and has gained us a great deal. In particular, we believe that 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.

Questions for our ongoing work include:

(1) What is ultimately the best organization for /proc?

(2) How important is the ability to overload virtual addresses? Is it purely a matter of backward compatibility?

(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 [62] 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] G. T. Almes, A. P. Black, E. D. Lazowska and J. D. Noe, ‘‘The Eden System: A Technical Review,’’ *IEEE Transactions on Software Engineering SE-11*:1 (January 1985), pp. 43-59.

[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 10*:1 (February 1992), pp. 53-79. Originally presented at the *Thirteenth ACM Symposium on Operating Systems Principles*, 13-16 October 1991.

[3] B. N. Bershad, D. T. Ching, E. D. Lazowska, J. Sanislo and M. Schwartz, ‘‘A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems,’’ *IEEE Transactions on Software Engineering SE-13*:8 (August 1987), pp. 880-894.

[4] B. N. Bershad and C. B. Pinkerton, ‘‘Watchdogs — Extending the UNIX File System,’’ *Computing Systems*, Spring 1988. Also Technical Report 87-12-06, Department of Computer Science, University of Washington, December 1987.

[5] 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*, 19-21 July 1988, pp. 1-9. In *ACM SIGPLAN Notices 23*:9.

[6] B. N. Bershad, T. E. Anderson, E. D. Lazowska and H. M. Levy, ‘‘Lightweight Remote Procedure Call,’’ *ACM Transactions on Computer Systems 8*:1 (February 1990), pp. 37-55. Originally presented at the *Twelfth ACM Symposium on Operating Systems Principles*, 3-6 December 1989.

[7] B. N. Bershad, T. E. Anderson, E. D. Lazowska and H. M. Levy, ‘‘User-Level Interprocess Communication for Shared Memory Multiprocessors,’’ *ACM Transactions on Computer Systems 9*:2 (May 1991), pp. 175-198.

[8] D. L. Black, ‘‘Scheduling Support for Concurrency and Parallelism in the Mach Operating System,’’ *Computer 23*:5 (May 1990), pp. 35-43.

[9] P. Brinch Hansen, *Programming a Personal Computer*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[10] L. A. Call, D. L. Cohrs and B. P. Miller, ‘‘CLAM — an Open System for Graphical User Interfaces,’’ *OOPSLA’87 Conference Proceedings*, 4-8 October 1987, pp. 277-286. In *ACM SIGPLAN Notices 22*:12 (December 1987).

[11] J. B. Carter, J. K. Bennett and W. Zwaenepoel, ‘‘Implementation and Performance of Munin,’’ *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, 14-16 October 1991, pp. 152-164. In *ACM SIGOPS Operating Systems Review 25*:5.

[12] J. S. Chase, H. M. Levy, E. D. Lazowska and M. Baker-Harvey, ‘‘Lightweight Shared Objects in a 64-Bit Operating System,’’ Technical Report 92-03-09, Department of Computer Science and Engineering, University of Washington, March 1992.

[13] 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.

[14] D. Cheriton, ‘‘The V Kernel — A Software Base for Distributed Systems,’’ *IEEE Software 1*:2 (April 1984), pp. 19-42.

[15] D. Clark, ‘‘The Structuring of Systems Using Upcalls,’’ *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, 1-4 December 1985, pp. 171-180. In *ACM SIGOPS Operating Systems Review 19*:5.

[16] P. Dasgupta, R. J. LeBlanc, Jr. and W. F. Appelbe, ‘‘The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work,’’ *Proceedings of the Eighth International Conference on Distributed Computing Systems*, 13-17 June 1988, pp. 2-9.

[17] Dobberpuhl and others, ‘‘A 200mhz 64 Bit Dual Issue CMOS Microprocessor,’’ *Proceedings of the International Solid-State Circuits Conference*, February 1992.

[18] J. Edler, J. Lipkis and E. Schonberg, ‘‘Process Management for Highly Parallel UNIX Systems,’’ Ultracomputer Note #136, Courant Institute, N. Y. U., April 1988.

[19] C. N. Fischer and R. J. LeBlanc, Jr., *Crafting a Compiler*, Benjamin/Cummings, Menlo Park, CA, 1988.

[20] B. Fleisch and G. Popek, ‘‘Mirage: A Coherent Distributed Shared Memory Design,’’ *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 3-6 December 1989, pp. 211-223. In *ACM SIGOPS Operating Systems Review 23*:5.

[21] P. Gautron and M. Shapiro, ‘‘Two Extensions to C++: A Dynamic Link Editor and Inner Data,’’ *Proceedings of the USENIX C++ Workshop*, November 1987, pp. 23-32.

[22] D. Gifford, P. Jouvelot, M. Sheldon and J. W. O’Toole, Jr., ‘‘Semantic File Systems,’’ *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, 13-16 October 1991, pp. 16-25. In *ACM SIGOPS Operating Systems Review 25*:5.

[23] R. A. Gingell, J. P. Moran and W. A. Shannon, ‘‘Virtual Memory Architecture in SunOS,’’ *USENIX Association Conference Proceedings*, June 1987, pp. 81-94.

[24] M. Herlihy and B. Liskov, ‘‘A Value Transmission Method for Abstract Data Types,’’ *ACM Transactions on Programming Languages and Systems 4*:4 (October 1982), pp. 527-551.

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

[26] M. B. Jones, R. F. Rashid and M. R. Thompson, ‘‘Matchmaker: An Interface Specification Language for Distributed Processing,’’ *Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages*, January 1985, pp. 225-235.

[27] E. Jul, H. Levy, N. Hutchinson and A. Black, ‘‘Fine-Grained Mobility in the Emerald System,’’ *ACM Transactions on Computer Systems 6*:1 (February 1988), pp. 109-133. Originally presented at the *Eleventh ACM Symposium on Operating Systems Principles*, Austin, TX, 8-11 November 1987.

[28] T. Kaehler, ‘‘Virtual Memory on a Narrow Machine for an Object-Oriented Language,’’ *OOPSLA’86 Conference Proceedings*, 29 September - 2 October 1986, pp. 87-106. In *ACM SIGPLAN Notices 21*:11.

[29] 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*, 13-16 October 1991, pp. 41-55. In *ACM SIGOPS Operating Systems Review 25*:5.

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

[31] E. J. Koldinger, H. M. Levy, J. S. Chase and S. J. Eggers, ‘‘The Protection Lookaside Buffer: Efficient Protection for Single Address-Space Computers,’’ Technical Report 91-11-05, Department of Computer Science and Engineering, University of Washington, November 1991.

[32] D. A. Lamb, ‘‘IDL: Sharing Intermediate Representations,’’ *ACM Transactions on Programming Languages and Systems 9*:3 (July 1987), pp. 297-318.

[33] P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson and B. L. Stumpf, ‘‘The Architecture of an Integrated Local Network,’’ *IEEE Journal on Selected Areas in Communications 5* (November 1983), pp. 842-857.

[34] 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.

[35] K. Li and P. Hudak, ‘‘Memory Coherence in Shared Virtual Memory Systems,’’ *ACM Transactions on Computer Systems 7*:4 (November 1989), pp. 321-359. Originally presented at the *Fifth Annual ACM Symposium on Principles of Distributed Computing*, 11-13 August 1986.

[36] K. Li and R. Schaefer, ‘‘A Hypercube Shared Virtual Memory System,’’ *Proceedings of the 1989 International Conference on Parallel Processing*, August 1989.

[37] 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.

[38] B. D. Marsh, ‘‘Multi-Model Parallel Programming,’’ Ph. D. Thesis, TR 413, Computer Science Department, University of Rochester, July 1991.

[39] B. D. Marsh, C. M. Brown, T. J. LeBlanc, M. L. Scott, T. G. Becker, P. Das, J. Karlsson and C. A. Quiroz, ‘‘The Rochester Checkers Player: Multi-Model Parallel Programming for Animate Vision,’’ *Computer 25*:2 (February 1992), pp. 12-19.

[40] 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*, to appear. Earlier version published as TR 374, Computer Science Department, University of Rochester, June 1991.

[41] C. R. Morgan, ‘‘Special Issue on the Interface Description Language IDL,’’ *ACM SIGPLAN Notices 22*:11 (November 1987).

[42] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse and H. van Staveren, ‘‘Amoeba: A Distributed Operating System for the 1990s,’’ *Computer 23*:5 (May 1990), pp. 44-53.

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

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

[45] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer and J. Thompson, ‘‘A Trace-Driven Analysis of the UNIX 4.2 BSD File System,’’ *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, 1-4 December 1985, pp. 15-24. In *ACM SIGOPS Operating Systems Review 19*:5.

[46] J. Ousterhout, ‘‘Push Technology, Not Abstractions,’’ *ACM SIGOPS Operating Systems Review 26*:1 (January 1992), pp. 7-11. Overhead slides from a panel presentation at the *Thirteenth ACM Symposium on Operating Systems Principles*.

[47] U. Ramachandran and M. Y. A. Khalidi, ‘‘An Implementation of Distributed Shared Memory,’’ *Proceedings of the First USENIX Workshop on Experiences Building Distributed and Multiprocessor Systems*, 5-6 October, 1989, pp. 21-38.

[48] 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.

[49] M. Rozier and others, ‘‘Chorus Distributed Operating Systems,’’ *Computing Systems 1*:4 (Fall 1988), pp. 305-370.

[50] M. L. Scott and A. L. Cox, ‘‘An Empirical Study of Message-Passing Overhead,’’ *Proceedings of the Seventh International Conference on Distributed Computing Systems*, 21-25 September 1987, pp. 536-543.

[51] 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*, V. II – Software, 15-19 August 1988, pp. 255-262.

[52] 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.

[53] 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.

[54] M. L. Scott, ‘‘The Lynx Distributed Programming Language: Motivation, Design, and Experience,’’ *Computer Languages 16*:3/4 (1991), pp. 209-233. Earlier version published as TR 308, ‘‘An Overview of Lynx,’’ Computer Science Department, University of Rochester, August 1989.

[55] M. L. Scott and W. Garrett, ‘‘Shared Memory Ought to be Commonplace,’’ *Proceedings of the Third Workshop on Workstation Operating Systems*, 23-24 April 1992.

[56] M. L. Scott, ‘‘LYNX Reference Manual,’’ BPR 7, Computer Science Department, University of Rochester, August 1986 (revised).

[57] 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.

[58] R. H. Thomas and W. Crowther, ‘‘The Uniform System: An Approach to Runtime Support for Large Scale Shared Memory Parallel Processors,’’ *Proceedings of the 1988 International Conference on Parallel Processing*, V. II – Software, 15-19 August 1988, pp. 245-254.

[59] W. F. Tichy, ‘‘Design, Implementation, and Evaluation of a Revision Control System,’’ *Proceedings of the Sixth International Conference on Software Engineering*, September 1982.

[60] 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.

[61] M. Weiser, A. Demers and C. Hauser, ‘‘The Portable Common Runtime Approach to Interoperability,’’ *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 3-6 December 1989, pp. 114-122. In *ACM SIGOPS Operating Systems Review 23*:5.

[62] 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.

[63] P. R. Wilson, ‘‘Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware,’’ *ACM SIGARCH Computer Architecture News 19*:4 (June 1991), pp. 6-13.

[64] N. Wirth, ‘‘From Programming Language Design to Computer Construction,’’ *Communications of the ACM 28*:2 (February 1985), pp. 159-164. The 1984 Turing Award Lecture.

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

[66] 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.

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

Appendix

The rest of this document consists of manual pages for lds, ldl, /proc, and the single-level store.

CONTENTS

NAME

lds - static portion of a dynamic link editing system

SYNOPSIS

lds [-Q pathname] [-F sharing class] [ld options]

DESCRIPTION

lds is a front end for ld that provides the ability to classify modules into one of four sharing classes. It may be called from cc by specifying -tl -h/usr/grads/bin -Bs. lds combines the object modules specified on the command line into an executable a.out file. It allows individual classification of the object modules into one of the following classes: static private, dynamic private, static public, and dynamic public. Options other than -F and -Q are passed on to ld. Additionally lds processes -L arguments. Search paths specified with the -Q option apply to all object modules. Path names specified with the -L option apply strictly to those modules starting with -l that follow the option.

lds verifies the existence of modules. Different search rules are used at static link time and at run time. At static link time, lds searches for modules in: 1) the current directory, 2) the path specified by the -Q option, 3) the path specified by the LD_LIBRARY_PATH environment variable, and 4) the default library directories (/lib:/usr/lib:/usr/local/lib). If there are multiple static modules with the same name, the one that will be used is the first one found by the above rules. If a static module cannot be found, lds aborts execution. If a dynamic module cannot be located, a warning is given and linking continues. Lds passes the list of directories in which it searched to ldl, which prepends the then-current value of LD_LIBRARY_PATH to the list.

OPTIONS

-F*class*

Static private, dynamic private, static public, and dynamic public sharing classes are specified by spr, dpr, spu, and dpu respectively. Modules specified as static are linked together during the static link phase. The dynamic class indicates that the module is to be lazily linked at run time. Public further specifies that the module will be persistent, that is, it will continue to exist after the program has terminated.

-L*dir*

As in ld, allows a search path to be specified for all modules starting with -l. This option applies to only -l modules following the specification by -L and is appended to the end of the search path specified by -Q.

-L *dir*

Same as -L*dir*.

-Q*path*

Specifies a search path to be used for finding modules given on the

command line of lds, as described above.

EXAMPLE

```
lds -Fdpu f1.o -L/u/dir -lm -Fspr f2.o -Q/u/dir:/u/dir/dir f3.o  -Fshared
f4.o
```

DIAGNOSTICS

lds issues a warning for unlocated dynamic modules and an error for unlocated static modules.

AUTHOR

Ricardo Bianchini and Robert Wisniewski (University of Rochester, April 1992)
ricardo@cs.rochester.edu and bob@cs.rochester.edu

SEE ALSO

cc(1), ld(1), ldl(1)

NAME

ldl - lazy dynamic link editor

DESCRIPTION

ldl provides a dynamic linking facility to running executables. Programs may specify modules to be dynamically included, and directories where those modules are searched for, by using lds at static link time. At run time, ldl searches for the specified modules in: 1) the path specified by the LD_LIBRARY_PATH environment variable, 2) the directory in which lds was run at static link time, 3) the path specified by the -Q option to lds at static link time, 4) the path that the LD_LIBRARY_PATH environment variable held at static link time, and 5) the default library directories (/lib:/usr/lib:/usr/local/lib). Once these modules are found ldl links them into the address space. The type of linking done depends on the sharing class specified to lds at static link time. Modules declared public are linked into the address space at the address corresponding to the file in the single level store, and any changes made to the module are seen by others having mapped the file in as public. Modules declared private are linked into the private address space at a location determined by ldl, and all changes are local.

The modules linked in using the search rules above are used to resolve undefined references in the parent module. If these modules have any undefined references they are marked as untouchable by ldl. When the running program actually attempts to touch one of these modules, a signal handler catches the segmentation violation that results and activates ldl, which then repeats the above process of bringing in modules and resolving references.

When a module is brought in, its internal symbols are first resolved against the symbols defined in the modules from its own module and search path (these modules are searched for and brought in using the rules detailed above). If unresolved references still exist, the module's symbols are then resolved using the module and search path of the module that caused it to be loaded in (its parent). If unresolved references remain, the module is resolved using the module and search path of its parent's parent, and so on, until either all symbols have been resolved or ldl runs out of ancestors. If there are still unresolved references ldl leaves them untouched. User defined signal handlers may handle these references later.

The signal handler installed by ldl also catches references to unlinked objects in the systems single level store (see sls(5)). Attempting to access any of these objects by its address causes that object to be automatically linked into the calling program's address space.

AUTHOR

Bill Garrett and Jeff Thomas (University of Rochester, April 1992)
garrett@cs.rochester.edu and thomas@cs.rochester.edu

SEE ALSO

lds(1), sls(5), ld(1)

NAME

iname - inode to filename translation

SYNOPSIS

```
#include <sys/types.h>
#include <sys/iname.h>

int iname(inum,buf)
int inum;
char *buf;
```

DESCRIPTION

iname() returns the file name in the single level store that corresponds to the inode number specified by inum. Read, write, or execute permissions on the file are not required.

The parameters passed to iname are the inode number of the file whose name is needed and a character buffer where the result will be returned.

RETURN VALUES

iname() returns:

0 on success.

-1 on failure and sets errno to indicate the error.

ERRORS

iname() will fail if one or more of the following is true:

EFAULT — buf points to an invalid address.

ENOENT — The file referred to by inum does not exist in the single level store, or inum is outside the inode number range.

BUGS

iname() is an unsupported interface, and will change in future versions of the kernel. Programmers should use the addr2path() library routine.

AUTHOR

Leonidas Kontothanassis (University of Rochester, April 1992)
kthanasi@cs.rochester.edu

SEE ALSO

path2addr(1), addr2path(1), stat(2), sls(5)

NAME

sls - single level store

SYNOPSIS

#include <sys/iname.h>

DESCRIPTION

The single level store is a special file system type known to the Unix mount system call. For a single level store file system the kernel maintains a mapping between file names and their corresponding inode numbers. This mapping makes it possible to retrieve the filename given the inode number, in contrast to normal filesystems in which the conversion is possible only in the other direction.

Single level store filesystems support the iname system call. They also support modified versions of the creat, open, link, symlink, mkdir, mknod, rmdir, rename, unlink and smount system calls that maintain the inode-filename correspondence.

To mount a file system as a single level store one needs to use the mount command with the special sls flag.

EXAMPLES

Mounting a file system as a single level store:

```
example% mount -t sls device directory
```

Unmounting a single level store file system:

```
example% umount directory
```

BUGS

Currently only one filesystem can be mounted as a single level store flag at any time. Also, the kernel tables are big enough for only 1024 files. Creation of more files will result in failure to maintain the inode-filename mapping and inconsistent behavior.

AUTHOR

Leonidas Kontothanassis (University of Rochester, April 1992)
kthanasi@cs.rochester.edu

SEE ALSO

iname(2), stat(2)

NAME

addr2path - translate from an address to a filename

SYNOPSIS

```
#include <sys/types.h>
#include <sys/pathaddr.h>

int addr2path(addr,buf)
int addr;
char *buf;
```

DESCRIPTION

addr2path finds the filename of the file whose address is addr and returns it in the buffer buff. Read write or execute permissions on the file are not required.

addr2path calculates the inode number that corresponds to addr and calls iname with the inode number and the user supplied buffer.

RETURN VALUES

addr2path() returns:

The offset in the file as specified by addr

-1 on failure and sets errno to indicate the error.

ERRORS

path2addr() will fail if one or more of the following is true:

EFAULT buf points to an invalid address.

ENOENT No file in the single level store corresponds to address addr or addr is out of the range of addresses allocated for the single level store

AUTHOR

Leonidas Kontothanassis (University of Rochester, April 1992)
kthanasi@cs.rochester.edu

SEE ALSO

iname(2), path2addr(3), sls(5)

NAME

path2addr - translate from a filename to an address

SYNOPSIS

```
#include <sys/types.h>
#include <sys/pathaddr.h>

int path2addr(path)
char *path;
```

DESCRIPTION

path2addr returns the starting address in the single level store of the file path Read write or execute permissions on the file are not required, but all directories listed in the path name leading to the file must be searchable.

path2addr calls stat with path and then uses the inode number of the file to calculate the address. path2addr and addr2path are the approved interface for translating back and forth between path names and addresses. The association between inodes and addresses will be abandoned in future versions of the kernel.

RETURN VALUES

path2addr() returns:

The address of the file on success

-1 on failure and sets errno to indicate the error.

ERRORS

path2addr() will fail if one or more of the following is true:

EFAULT path points to an invalid address.

ENOENT The file referred to by path does not exist or its address clearly indicates that it is out of the single level store.

BUGS

path2addr will return an address for a file even when the file is not part of the single level store if its inode number is less than or equal to 1024.

AUTHOR

Leonidas Kontothanassis (University of Rochester, April 1992)
kthanasi@cs.rochester.edu

SEE ALSO

stat(2), sls(5)