Shared Libraries in SunOS

Robert A. Gingell
Meng Lee
Xuong T. Dang
Mary S. Weeks

Sun Microsystems, Inc.
2550 Garcia Ave.
Mountain View, CA 94043

ABSTRACT

The design and implementation of a shared libraries facility for Sun's implementation of the UNIX† operating system (SunOS) is described. Shared libraries extend the resource utilitization benefits obtained from sharing code between processes running the same program to processes running different programs by sharing the libraries common to them.

In this design, shared libraries are viewed as the result of the application of several more basic system mechanisms, specifically

- kernel-supplied facilities for file-mapping and ''copy-on-write'' sharing;
- a revised link editor supporting dynamic binding; and
- compiler and assembler changes to generate position-independent code.

The use of these mechanisms is transparent to applications code and build procedures, and also to library source code written in higher-level languages. Details of the use and operation of the mechanism are provided, together with the policies by which they are applied to create a system with shared libraries. Early experiences and future plans are summarized.

1. Introduction

The UNIX operating system has long achieved efficiencies in memory utilization through sharing a single physical copy of the *text* (code) of a given program among all processes that execute it. However, a program *text* usually contains copies of routines from one or more libraries, and occasionally a program consists *mostly* of library routines. Considering that virtually every program makes use of routines such as *printf*(3), then at any given time there are as many copies of these routines competing for system resources as there are different active programs.

In an environment containing single-user systems, such as workstations, the likelihood of achieving much benefit from sharing multiple copies of entire programs seems small. As the number of programs in a system increases (a guaranteed attribute of each new system release), so does the waste in file storage resources containing yet more copies of common library routines. Thus, there is increasing motivation to extend the benefits of sharing to processes executing *different* programs, by sharing the libraries common to them.

This paper describes the design and implementation of a shared libraries facility for Sun's implementation of the UNIX operating system, SunOS. We discuss our goals for such a facility, our approach to its design and implementation, and our plans for its use. We also discuss our early experiences, and our plans

† UNIX is a trademark of Bell Laboratories.

for the future.

2. Goals

Most of our goals were driven by a desire to have a facility that was as simple to use and evolve as possible. We also wanted to provide mechanisms that were as flexible as possible, so that the work we performed could be used to support other activities and projects. Providing mechanisms with great apparent simplicity would also help motivate their use. To that end, we arrived at the following specific goals:

- **Minimize kernel support**. Clearly, any support we put in the kernel would be very inflexible, and further complicate an already complex environment. We considered an ideal situation to be one involving no kernel changes.
- **Do not require shared libraries**. Although we might make the use of shared libraries the default system behavior, we felt we could not *require* their use or otherwise build fundamental assumptions requiring them into other system components.
- **Minimize new burdens**. The introduction of any new facility creates the potential for new burdens to be imposed upon its users. To minimize these, we decided how shared libraries should impact various groups:
    - **Application programmers**: The use of shared libraries must be transparent to application source code, program build procedures, and the use of standard utilities such as debuggers. It was also considered desirable to be able to use existing object files.
    - **Library programmers**: That a body of library code is to be built as a shared library must also be transparent to its source code. However, it need not be transparent to the procedures used to build the library, and such a goal appeared contradictory in any case – someone has to decide that a shared library will be built. The goal to not change library source was a direct consequence of not having the resources to change the large amount of library code already in existence. Even source alterations such as those used with System V shared libraries [ARNO 86] appeared more than we wished to do.
    - **Administrative**: There should be no requirement to administer and coordinate the allocation of address space. Libraries should be able to evolve and be updated without requiring rebuilding of the programs that used them as long as their interfaces are compatible, and mechanisms would have to be available to handle interface changes.
- **Improve the environment**. Where possible, we wanted our changes to provide functional benefits beyond the resource utilization ones we expected. This included having a great deal of flexibility in easily testing updates to libraries.
- **Performance**. Shared libraries represents a classic time vs. space trade-off opportunity. We were deferring the work of incorporating library code into an address space in order to save both secondary and primary storage space. Thus, we expected to pay a time penalty in programs using shared libraries. However, the expectation was that if sharing of library code really occurred, then the I/O (real) time required to bring in a program and get it executing would be greatly reduced. As long as the CPU time required to merge the program and its libraries did not exceed the I/O time we saved, the apparent performance would be the same or potentially even better. This approach fails if sharing does not occur, or if the system is CPU saturated already.

  Even though a moderate cut in I/O time offers a large window for computation, we felt that an attempt to equal the performance of current systems was unrealistic, and instead set two performance goals permitting a limited degradation in CPU performance for programs that used shared libraries. These goals were:

    - $\leq$ 10% for programs not dominated by start-up costs; and
    - $\leq$ 50% for programs that were dominated by start-up costs.

  A program was considered to be dominated by start-up costs if it took less than half a second to execute on a Sun-3/75.

3. Approach

Given our goals for flexibility, the most productive approach was not to build a mechanism *specific* to shared libraries. Rather, by abstracting the general properties we required of shared libraries and providing mechanisms to deliver those properties directly, we hoped to achieve the sought-for benefits and flexibility to address the needs of other projects. The mechanisms we chose were:

- a high degree of memory sharing of general objects (e.g., files) at a fine level of granularity (pages);
- a revised system link editor (*ld*) that supports dynamic loading and binding; and
- compiler changes to generate *position-independent* code (PIC) that need not be relocated for use in different address space arrangements and thus may be directly shared.

3.1. Memory Sharing

The mechanism that provides our memory sharing is a new Virtual Memory (VM) system for SunOS. Although more completely described elsewhere [GING 87], the principal features of the new system include:

- file mapping as its principal mechanism, accessed by programs through the *mmap*(2) system call;
- sharing at the granularity of a file page; and
- a per-page copy-on-write facility to allow run-time modification of a shared object without affecting other users of the object.

The new VM system uses these features internally, so that the act of *exec*'ing a program is reduced to the establishment of copy-on-write mappings to the file containing the program. A ''shared library'' is added to the address space in exactly the same way, using the general file mapping mechanism. The use of files in this way originated with MULTICS [ORGA 72], and the use of file page mapping to incorporate library support at execution time was established with TENEX [MURP 72] and its evolution as Digital Equipment's TOPS-20. Comparable approaches have been applied with UNIX-based systems as described in [SZNY 86] and [DOWN 84].

3.2. New *ld*

The changes to *ld* reflect an observation that the activities that must occur to execute a program with shared libraries are no different than those to execute one without them, at least conceptually. All that has really changed is when, and over what scope of material, those activities occur. Conceptually, *ld* has been turned into a more general facility available at various times in the life of a program (in perhaps different guises) to perform its link editing function.

The old *ld* built all programs *statically*. Executable (*a.out*) files contained complete programs, including copies of necessary library routines. Executables were created by link editing the program in (usually) a single batch operation using *ld*. *ld* would refuse to build an incomplete executable file.

The new *ld* will build ''incomplete'' *a.out* files, deferring the incorporation of certain object files until some later time (generally program execution). These deferred link editing operations employ the system's memory management facilities to map to and thus share these objects directly. A ''shared library'' is simply the code and data constituting a library built as such a *shared object* (*.so*) file. A *.so* is simply one of these ''incomplete'' *a.out* files that lacks an entry point. It should be noted that a *.so* file can be *any* object, a ''library'' is simply one of many possible semantic uses for it.

As previously noted, dynamic link editing is still essentially the same operation as static link editing, but occurring at a different time. A link editing operation effects some change to either the material being added, or that to which it is added, or more likely both. However, when an object is changed as the result of such processing, it can no longer be shared with other users of the object, as the change is unlikely to be useful to any program other than the one in which it occurs. Such changes are accommodated automatically by the VM system, using its copy-on-write facilities to create per-process private copies of the pages of the file the process attempts to modify. Thus, the extent of the changes a link edit performs affects the degree to which sharing can occur.

Although a dynamic link edit operation may impact the degree to which sharing can occur in a system, it does not affect the *correctness* of the resulting program. A strong characteristic of our approach is this separation between ‘‘right and wrong’’ vs. ‘‘good and bad’’. Almost any legitimate combination of objects can be link edited into a program at any time (e.g., there are very few ‘‘wrong’’ combinations), but those that maximize sharing will be ‘‘best’’.

3.3. PIC

In the previous section it was observed that code that minimizes the amount of dynamic link editing promotes sharing and is thus ‘‘best’’. To increase the prospects for having the ‘‘best’’ code, we changed our C compiler to optionally generate *position-independent* code (PIC). PIC needs link editing only to relocate references to objects external to the body of code that has been built as PIC, and is thus more sharable. Again, it is not *necessary* to have PIC, just *better*.

However, PIC programs will be slower than non-PIC ones. To localize the link editing for references to global objects, the code refers to such objects indirectly through *linkage tables*. The specific amount of degradation is a function of the number of dynamic references to global objects.

4. Mechanisms

The previous section provided an overview of the approach we have employed, and briefly identified the mechanisms we would use. With this background, we describe the mechanisms in greater detail.

4.1. Compiler Changes

The C compiler has been altered to take a new option (**-pic**) that causes it to generate PIC. When **-pic** is specified, the code generated by the compiler changes in the following ways:

- Each function prologue is extended to include the initialization of a register that is used as the base address of a *linkage table* to global objects, this table is called the *global offset table* (*GOT*). For the Motorola 680x0 used in Sun’s workstations, this code is:

  ```
  movl #__GLOBAL_OFFSET_TABLE,a5     | Get offset to GOT
  lea pc@(0,a5:L),a5                 | Get absolute address
  ```

  which computes the absolute address of the *GOT* associated with this function based on a PC-relative offset from the function prologue to the table. The register `a5` is unavailable for the life of the function, and is one of those that the compiler expects called functions to preserve.

- Each reference to a global data object is generated as a dereference of a pointer in the *GOT*. For example, a reference to the external integer `errno` in C is generated as:

  ```
  movl a5@(_errno:w),a0              | Get address of _errno
  movl a0@,d0                        | Get contents
  ```

  Currently, the code generation scheme for static data objects is identical to that used for globals. This represents an area for future optimization work.

- Each function call is generated as an assembler pseudo-operation including a ‘‘free register’’, for example:

  ```
  jbsr _foo,a0                       | _foo()
  ```

  for an expression involving a call to the function `foo`. The assembler will, if `_foo` is undefined to it, expand the pseudo-operation to an instruction sequence that involves loading a PC-relative reference to an entry in a *procedure linkage table* into the ‘‘free register’’, and then issuing a subroutine call instruction involving the PC in the calculation of the effective address.

The code sequences generated for the 680x0 assume that the linkage tables are of a limited size, specifically no larger than 64K bytes. In the event the tables require a larger size, the compiler can be coerced into generating more clumsy code sequences permitting linkage tables to a full 32 bits in size (by expressing **-pic** as **-PIC**). However, we have yet to find a program that requires the use of this option.

4.2. Assembler Changes

The code generated by the compiler with the **-pic** option requires support from the assembler. The support required is that the assembler generate some new relocation information for certain constructs, and a change in interpretation for some syntactic forms. This support is enabled by the assembler flag **-k**, and is generated automatically by the C compiler driver when invoking the assembler for a compilation that contained the **-pic** or **-PIC** options.

When assembling a module with the **-k** flag enabled, the assembler:

- interprets a relocatable expression in an operand involving an ‘‘immediate’’ addressing mode as a PC-relative reference to any symbol involved and generates a PC-relative relocation record for the expression;
- interprets symbolic relocatable expressions in operands involving base-register relative addressing as a reference to the *GOT* entry for the symbol and generates a relocation record indicating such; and
- generates a ‘‘procedure call’’ relocation type for all `jbsr` pseudo-operations it assembles.

It should be noted that although examples have been provided using an assembler for the 680x0 processor employed in Sun workstations, the requirements for these special relocation types are architecture independent.

4.3. Link Editor changes

The most extensive changes have been performed to the link editor, *ld*. These not only include changes to the batch form of the link editor (embodied as *ld*), but also the creation of an execution-time version (*ld.so*).

4.3.1. Batch link editor (*ld*)

The batch link editor, *ld*, combines a variety of module types to produce an *a.out* file. How that *a.out* file can be used is very much dependent on what *ld* can determine to do with the information it has been fed. Whereas the previous version of *ld* had to determine *everything* about a program, the new version simply stops working when it runs out of information on the assumption that later events will provide more.

ld’s output can be one of two basic types, including:

- a ‘‘simple object’’ (*.o* file), produced by simply combining other *.o*’s into a single, larger one (**-r** flag);
- an ‘‘executable’’ (*a.out*), which is is either a ‘‘program’’ (has an entry point) or a *shared object* (*.so*) (does not have an entry point).

The production of a *.o* file through the use of the **-r** flag is a special use of *ld* that, while useful, is not relevant to the issues being discussed and will not be considered further.

Exactly what gets produced depends on what *ld* was fed in the way of input files and command line options. *ld* will process the following kinds of input files:

- simple object files, *.o*’s;
- *archives*, *.a*’s, conglomerates of simple objects and also referred to as *libraries*; and
- *shared objects*, *.so*’s, also known as dynamically bound executables and sometimes called *shared libraries*.

Each *.o* file is simply concatenated to previous *.o* files in the order it is encountered. In this respect, *ld* is unchanged except that it handles the new relocation operations required by code the assembler generated as PIC.

Each *.a* is searched exactly once as it is encountered – only those entries matching an unresolved external reference are extracted and concatenated. Again, this is exactly as *ld* has always done, with the addition of PIC handling.

Any *.so* encountered is (usually) searched for symbol definitions and references, but does not contribute any material to be concatenated except under certain conditions involving other options (described further below). However, their occurrence in the command line is stored in the resulting *a.out* file and utilized by the execution-time *ld.so* to effect dynamic loading and binding.

ld's **-l** flag is used to specify a short name for an object file to be used as a library. The full name of the object file is derived by adding the prefix *lib* and a suffix of either *.a* or *.so* (for archive or shared library, respectively). The specific suffix applied depends on the binding ''mode'' *ld* is operating in at the time the **-l** flag is processed. *ld*'s binding ''mode'' is specified by a new flag, **-B** that takes several keyword arguments:

dynamic	Allow dynamic binding, do not resolve symbolic references, and allow creation of execution-time symbol and relocation information. This is the default setting.
static	Force static binding, implied by options that generate non-sharable executable formats.

-Bdynamic and **-Bstatic** may be specified multiple times and may be used to toggle each other on and off. Like **-l**, their influence is dependent upon their location. When **-Bdynamic** is in effect, any **-l** searches may be satisfied by the first occurrence of either form of library (*.so* or *.a*), but if both are encountered the *.so* form is preferred. Since **-Bdynamic** is the default setting, the use of shared libraries in the construction of a program thus ''falls out'' from simply installing the *.so* that represents the shared library in the library search path used by *ld*.

If **-Bstatic** is in effect, however, *ld* will refuse to use any *.so* forms of libraries it encounters and continue searching for the *.a* form. Further, an explicit request to load a *.so* file is treated as an error.

After *ld* has processed all its input files and command line options, the form of the output it produces is based on the information it has been able to discern. *ld* first tries to reduce all symbolic references to relative numerical offsets within the executable it is building. To perform this ''symbolic reduction'', *ld* must know that either

- all information relating to the program has been provided, in particular, no *.so* will be added at execution time; and/or
- this program has an entry point and symbolic reduction can be performed for all symbols having definitions existing in the material it has been provided.

It should be noted that uninitialized ''common'' areas (essentially all uninitialized C globals) are allocated by the link editor *after* it has collected all references. In particular, this allocation can not occur in a program that still requires the addition of information contained in a *.so* file, as the missing information may affect the allocation process. Initialized ''commons'', however, are allocated in the executable in which their definition appears.

After *ld* has performed all the symbolic reductions it can, it attempts to transform all relative references to absolute addresses. *ld* is able to do this ''relative reduction'' only if it has been provided *some* absolute address, either implicitly through the specification of an entry point, or explicitly through other *ld* options. If, after performing all reductions it can, there are no further relocations or definitions to perform, then *ld* has produced a completely linked executable – essentially its old behavior.

However, if any reductions remain, then the executable being produced will require further link editing at execution time in order to be useable. In the data spaces of such executables, *ld* creates an instance of a `link_dynamic` structure that has the label `__DYNAMIC`. The `link_dynamic` structure has the form:

```
struct link_dynamic {
        int     ld_version;             /* Version # */
        struct  link_map *ld_loaded;    /* Loaded objects */
        long    ld_need;                /* Needed objects */
        long    ld_got;                 /* Global offset table */
        long    ld_plt;                 /* Procedure linkage table */
        long    ld_rel;                 /* Relocation table */
        long    ld_hash;                /* Symbol hash table */
```

```
                long    ld_stab;                /* Symbol table itself */
                long    (*ld_stab_hash)();      /* Hash function */
                long    ld_buckets;             /* Number of hash buckets */
                long    ld_symbols;             /* Symbol strings */
                long    ld_text;                /* Size of text area */
        };
```

This data structure is used by *ld.so* to obtain *.so*'s on which this executable depends, and to find the symbolic and relative reduction operations that remain to be performed. The `link_dynamic` structure contains elements that allow evolution of the interfaces to occur without invalidating existing programs. These include the `ld_version` element, and the incorporation of the hash function for the execution-time symbol table as part of the executable.

4.3.2. Relocation of PIC

As described previously, code generated as PIC contains several new relocation record entries: PC relative, references to entries in a *global offset table* (*GOT*), and references to entries in a *procedure linkage table* (*PLT*).

PC relative relocations are easily handled by *ld*: the value replacing the relocation is simply the offset between the location reference and definition of its target.

GOT and *PLT* entry references are more complex, however. Both of these data structures are allocated by *ld* as part of creating an executable comprised of at least one PIC module, that is, a module containing either *GOT* or *PLT* or both relocation forms. *ld* is responsible for assigning entries in each of these tables for each unique symbol referenced in either a *GOT* or *PLT* reference, and creating a new relocation entry for the table entry. The resulting relocations are then processed just like any other handled by *ld*, by first attempting symbolic and then relative reductions. The table entries themselves are (at least conceptually) indirect pointers to the targets of global references.

4.3.3. *crt0*

Every main program produced by the standard languages is linked with a program prologue module, *crt0*. This module actually contains the program's entry point, and performs various initializations of the environment prior to calling the program's main function or logical starting point. *crt0* was modified to contain a reference to the symbol `__DYNAMIC`. As described above, when *ld* builds an executable requiring execution-time link editing, it defines this symbol as the address of a data structure containing information needed for execution-time link editing operations. If the structure is not needed, any reference to the symbol `__DYNAMIC` is relocated to zero.

Thus, at program start-up, *crt0* tests to see whether or not the program being executed requires further link editing. If not, *crt0* simply proceeds with the execution of the program as it always has – no further processing is involved. However, if `__DYNAMIC` is defined, *crt0* opens the file `/lib/ld.so` and requests the system to map it into the program's address space via the *mmap* system call. It then calls *ld.so*, passing as an argument the address of its program's `__DYNAMIC` structure. *crt0* assumes that *ld.so*'s entry point is the first location in its text. When the call to *ld.so* returns, the link editing operations required to begin the program's execution have been completed.

4.3.4. *ld.so*

After *crt0* transfers control to *ld.so*, *ld.so* executes a short bootstrap routine that performs any relocations *ld.so* itself requires. The process of building *ld.so*, described further below, results in only very simple forms of relocation that can be easily handled by this bootstrap routine. *ld.so* then processes the information contained in the `__DYNAMIC` structure of the program that called it, in order to perform the link editing required to start execution of the program.

ld.so's first action is to examine the `ld_need` entry of the program's `__DYNAMIC` structure. This entry contains an offset relative to the `__DYNAMIC` structure of an array of `link_object` structures. Each element of the array has the structure:

```
struct link_object {
        char    *lo_name;               /* Name of object */
        int     lo_library : 1;         /* Library search */
        short   lo_major;               /* Major version */
        short   lo_minor;               /* Minor version */
};
```

and identifies a *.so* that must be added to the program's address space and link edited. The identification is the name specified on the *ld* command line used to build the program, and includes a bit indicating whether the object was named explicitly or via an *ld* **-l** option. Some version control information is also recorded, however a discussion of the use of this information is deferred.

For each entry in the `ld_need` array, *ld.so* looks up the file identified and maps it into the process's address space. The location in the address space to which the *.so* is mapped is left to the system to decide, and a given *.so* may reside at different locations in the address spaces of different processes. Failure to find a needed object is a fatal error and results in the program's termination. At the end of the initial program's `ld_need` array, *ld.so* examines the `__DYNAMIC` structure of the first *.so* file it mapped in. It processes that *.so's* `ld_need` array, and proceeds likewise through all the loaded *.so*'s. Any references to already processed *.so* files are ignored.

For each *.so* that is loaded, *ld.so* builds a `link_map` data structure having the form:

```
struct link_map {
         caddr_t lm_addr;               /* Address mapped */
         char    *lm_name;              /* Absolute pathname */
         struct  link_map *lm_next;     /* Next .so */
};
```

Each such structure is placed on a singly linked list in the order it was loaded. The head of the list is rooted in the `ld_loaded` member of the initial program's `__DYNAMIC` structure. This ordering of the loaded *.so*'s is used to establish the search order for undefined symbol look-ups.

After all of the modules comprising the complete program have been placed in the address space, *ld.so* attempts to complete the link editing operations begun by *ld*. Specifically, it attempts to perform first symbolic and then relative reductions on all the references *outside of procedure linkage tables* left in the program. In particular, this includes the allocation of any uninitialized commons (since all information regarding their use is finally present). If all non-procedural references can not be reduced to absolute addresses, then it is because a definition for a given symbol is not available, in which case *ld.so* terminates the program with an ''undefined symbol'' diagnostic.

All non-reduced references in any *PLT*'s in the loaded executables are not processed during program startup. Rather, all such references are initialized to cause the initial calls to the procedures they reference to result in the transfer of control to *ld.so*. Upon receiving control from such a reference, *ld.so* will reduce the original reference to the appropriate absolute address and modify the referencing *PLT* entry to direct future calls directly to the targeted procedure. Deferring the binding of procedure entry points until their first reference saves performing perhaps thousands of unnecessary bindings to entry points programs may never call.

4.3.5. Version Management of *.so*'s

The previous discussion of the handling of *.so* files in the course of processing an *ld* **-l** option was simplified with respect to *.so* version control. One of the goals of our project was to accommodate the evolution of shared libraries: to permit them to be updated without impacting the programs that used them so long as the interfaces remained compatible.

The *.so* files used as shared libraries actually employ a more complex name than has been described so far, involving a suffix that describes the version of the library contained in the file. Thus, interface version ''2'' of the C library, in its third compatible revision, would be placed in a *.so* having the name `libc.so.2.3`. The suffix may actually be an arbitrary string of numbers in Dewey-decimal format, although only the first two components are significant to the operation of the link editors at this time.

The first component is called the library's ‘‘major version’’ number, and the second component its ‘‘minor version’’ number. When *ld* records a *.so* used as a library, it also records these two numbers in the database used by *ld.so* at execution time. When *ld.so* finally searches for libraries, it uses these numbers to decide which of multiple versions of a given library is ‘‘best’’, or whether *any* of the available versions are acceptable. The rules it follows are:

- **Major Versions Identical**: the major version used at execution time must exactly match the version found at *ld*-time. Failure to find an instance of the library with a matching major version will cause a diagnostic to be issued and the program's execution terminated.
- **Highest Minor Version**: in the presence of multiple instances of libraries that match the desired major version, *ld.so* will use the highest minor version it finds. However, if the highest minor version found at execution time is less than the version found at *ld*-time, a warning diagnostic will be issued, although execution will continue.

The semantics of version numbers are such that major version numbers should be changed whenever interfaces are changed. Minor versions should be changed to reflect compatible updates to libraries, and programs will silently prefer the highest compatible version they can obtain. If minor version numbers drop, then although the interfaces should remain compatible, it is possible that certain bug fixes or compatible enhancements that the program builder wanted are unavailable: hence the warning diagnostic.

Although the mechanisms for supporting version evolution of shared libraries have been provided, we have not yet provided any tools to automate their use. As before, the detection of incompatibilities remains the responsibility of the library developer.

4.3.6. Link Editor Environment Variables

ld interprets the values of the environment variables LD_LIBRARY_PATH and LD_OPTIONS.

LD_LIBRARY_PATH augments *ld*'s built-in rules for directories to be used when searching for libraries specified with the **-l** option. If defined, the value of LD_LIBRARY_PATH should be a colon-separated list of directory names (as for the PATH variable of *sh*). The list specified by LD_LIBRARY_PATH is prepended to the list of *ld*'s built-in rules, and follows any further directories specified on the command line with **-L** options.

LD_OPTIONS specifies a default set of options to *ld*. LD_OPTIONS is interpreted by *ld* just as though its value had been placed on the command line immediately following *ld*'s invocation, as in:

```
% ld $LD_OPTIONS ... other ld arguments ...
```

ld.so also interprets the LD_LIBRARY_PATH environment variable, and may be used to substitute test versions of libraries in their own environments at execution time.

4.3.7. Considerations of Dynamically Linked Programs

Beyond providing a basis for improved sharing of system resources, the ability to defer the binding of library and other code offers a number of other potential advantages in terms of increased flexibility for maintenance and development. However, the environment they create is also inherently more complex, something that the policies governing the application of the mechanisms must address. Some aspects of this more complex environment include:

- **Multiple files**: a dynamically bound program consists not only of the executable file that is the output of *ld*, but also of the files referenced during execution. Moving a dynamically bound program *may* also involve moving a number of other files as well. Moving (or deleting) a file on which a dynamically bound program depends may prevent that program from functioning.
- **Ubiquitous link editor**: the previous behavior of *ld* was to produce only a fully linked executable. Link editing issues could be forgotten or ignored once the executable had been successfully produced. However, deferring some of the link editing means (potentially) deferring some of the errors that could occur. With the new facilities, it is possible for a running program to produce a link editor error.

  Consider the following example: a programmer misspelling in the use of the function call *printf* results instead in a reference to ‘‘pintf’’. During testing of the code in which the misspelling occurs, no path to the ‘‘pintf’’ reference is ever exercised. However, a later production user does exercise the path. The (no doubt surprised) user will find the program terminated with the message: ‘‘_pintf: undefined’’.

To deal with such problems, *ld* has been provided with an assertion-checking facility that (among other things) can be used to determine if a given program will encounter undefined symbols during execution *if used with the dynamic objects now on the system*. Later erroneous changes to such dynamic objects might still create this problem, however. Program builders wishing to isolate themselves from such problems should simply link their programs statically.

- **Semantic Differences**: there are some semantic differences between the dynamic and static binding algorithms. The differences are not expected to manifest themselves as problems with existing programs, unless such programs engaged in questionable practices in their use of library search ordering. The major semantic difference that can create a problem involves old programs built from several components, where several of those components suddenly become dynamically loadable and others remain static.

  Consider the *ld* command:

  ```
  % ld -o x ... <dc> <sc>
  ```

  The executable `x` consists of several objects including a dynamic component (`<dc>`) and a static component (`<sc>`). `<dc>` was, prior to the introduction of the new mechanisms, an unordered archive file. `<dc>` and `<sc>` both contain definitions for the symbol `bar`. In addition, `<dc>` contains a reference to `bar`. If, in `<dc>`’s prior existence as an unordered static archive, the definition of `bar` preceded its reference, the `ld` operations to build `x` may have satisfied `<dc>`’s reference with the definition from `<sc>`. However, in its dynamic form, `<dc>`’s own definition will be used. This is a consequence of the fact that at execution time, all searches for a symbol definition start with the main program and then all *.so*’s in load order. This behavior preserves the ability to *interpose* on library entry points.

4.4. Debuggers

The debuggers used in the SunOS environment, *adb* and *dbx*, have been modified to deal with the dynamic linking environment provided by the new *ld*. In particular, they understand that symbol definitions may appear after a program starts executing. Such dynamically added symbols are found by noting the creation of the `link_map` structure list in the initial program’s `__DYNAMIC` structure, and adding the symbols for the *.so*’s that have been added to the debugger’s database of symbols.

Despite our goal for transparency in the tools application programmers use, debugger users must also have some awareness of the use of dynamic linking. For example, if they reference the symbol `printf` in a program that uses a shared C library but has not yet started executing, the debugger will fail to find it. If, however, such a reference has been made after the same program has executed far enough to call the program’s `main()`, then `printf` will appear.

5. Policies: Applying the Mechanisms

The previous sections have provided descriptions of our approach to providing a shared library capability through the application of basic mechanisms. We have also described the basic mechanisms involved. In this section, we describe the policies by which we use the mechanisms to build a system that provides and uses shared libraries. In general, the considerations applied in setting these policies were (in decreasing order of priority):

- maximize sharing (resource utilization performance);
- maximize flexibility (enriched environment);
- ‘‘Principle of Least Astonishment’’ (user compatibility).

This is to say that a conflict between something that was completely compatible and something that improved sharing or flexibility, generally favored the latter. However, in many cases, it has been possible to accomplish all three considerations.

5.1. System Construction

To meet the goals for resource reduction in the system, the system itself should be built to use shared libraries, and thus, dynamic link editing. This creates the potential for three sorts of problems:

- deferred errors (the so-called ‘‘pintf’’ problem) that are manifested after the system is installed;
- the potential for chaos if an important shared library is deleted; and
- the potential for security problems with ‘‘setuid’’ programs.

To deal with the problem of deferred errors, a set of programs that are supposed to be self-consistent should be built using the assertion-checking facilities previously described.

To deal with the chaos that would result if (for example) a shared C library were deleted from the system, a number of commands and utilities will not be built with shared libraries. These include but would probably not be limited to: *init*(8), *getty*(8), the shells, *mv*(1), *ln*(1), *ls*(1), *tar*(1) and *restore*(8) – essentially programs that would be necessary to restore the missing library from some other other source.

Finally, programs that are built as ‘‘setuid’’ (or ‘‘setgid’’ for that matter) are not built to use shared libraries. Such programs could be easily subverted by incorporating a ‘‘trojan horse’’ into a library on which they depend.

5.2. Dynamic Binding

To maximize the benefits of shared libraries, we have decided to make their use the default by having the default binding mode for *ld* be **-Bdynamic**. This creates the potential for users of the programs *ld* builds to be ‘‘surprised’’ by the special considerations of dynamically linked executables the next time they rebuild their programs. In this case, our preference for maximizing sharing took precedence over the potential for user surprise, a choice we made because we believe:

- most users want the benefits; and
- the mechanisms are sufficiently transparent that the ‘‘potential’’ for surprise is not considered to be the same as ‘‘likelihood’’.

The greatest impact is expected to be on those users who create programs for shipment to other systems. Such users probably want to be isolated from the various problems that a dynamically linked program can have, and should force their programs to be linked statically. While this may impact existing build procedures, such developers usually take special steps when building production programs (such as removing debugging features and employing extra optimization). The addition of another consideration appeared to be a small cost relative to the benefits obtained by the community through maximizing sharing.

5.3. Use of assertions

To help deal with the potential complexities created by dynamic linking, *ld* has been provided with the ability to validate some assertions about an executable it builds. The assertion checking is invoked with the *ld* flag **-assert**, followed by a keyword argument from one of:

definitions	if the resulting program were run now, there would be no run-time undefined symbol diagnostics;
nosymbolic	there are no symbolic relocation items remaining to be resolved; and
pure-text	the resulting executable requires no further relocations to its text.

Together, these assertions are intended to support the development of production programs by allowing the verification of important properties: for instance that a program will not produce run-time link edit diagnostics, or that a piece of code intended to be a ‘‘shared library’’ is in fact sharable.

5.4. PIC Generation

As has been pointed out, PIC in dynamically linked objects improves their ability to be shared and is thus a more efficient use of system resources. However, PIC executes slower than non-PIC, the degree of degradation being dependent on the number of dynamic indirect references the code incurs. Although refinements to the generated code may ultimately make the performance impact of PIC negligible, we have chosen to make the use of PIC an option. Our expectation is that only code intended to be part of a shared library will be compiled as PIC.

We also expect that few users will enable the generation of PIC in their application programs, simply because it takes extra effort to do so. However, this raises the issue of the binding of non-PIC code to the PIC shared libraries it uses. The binding that must occur involves all references to:

- **commons**: allocated after the program is completely assembled;
- **initialized data**: imported from the shared libraries; and
- **entry points**: supplied by the shared libraries.

The implication of these binding operations is simply that the link editing that implements the binding will render the edited code unsharable.

To improve the degree of sharing for such programs, *ld* can be made to force the allocation of commons and to create aliases for library entry points. These allocations and aliases are created as part of the non-PIC executable, and result in ''pure-text'' non-PIC programs even if they have dynamically linked components. These options (**-dc** to force the definition of common storage, and **-dp** to force the definition of procedure aliases), are included by the C compiler driver automatically in the *ld* command line it generates.

6. Examples: *.so* Construction

6.1. Shared C Library

The construction of the shared C library involved:

- compilation of all of its C source modules using the **-pic** option;
- modifications of some assembly-language source files so that the assembly source was also position-independent; and
- *ld*'ing the resulting collection of *.o* files to create `libc.so.1.0`.

The modification of the assembly-language source files was, of course, not a requirement for the library to function − simply to make it more sharable. In this area, we fell short of our goal for having shared libraries be transparent to library source code, though happily the amount of assembly source in the system is relatively small.

The *ld* operation, although in reality involving more complex operations resulting from the way we build the various versions of the C library, is conceptually just:

```
% ld -o libc.so.1.0 -assert pure-text *.o
```

assuming the current directory for the command contained only the *.o* files comprising the C library. Since no entry point is supplied, and lacking any other clue to an absolute address, *ld* simply stops processing after it has combined all the object files, built the *GOT* and *PLT*'s, and performed any intra-library PC-relative relocations. The assertion request will cause *ld* to issue a diagnostic if the library requires further relocation to the code contained within it, a sign that a non-PIC object has found its way into the library.

6.2. *ld.so*

The execution-time link-editor, *ld.so* is built with an *ld* command that has the form:

```
% ld -o ld.so -Bsymbolic -assert nosymbolic ... list of modules ...
```

and is conceptually just like other *.so* files. However, it also involves the use of a special binding control option **-Bsymbolic** and the assertion **nosymbolic**.

Normally when *ld* builds a program lacking an entry point or other absolute addressing information, it is unable to perform its symbolic reduction operations simply because it can not assume that symbols from other executable files will not be added later. However, *ld.so* must be self-contained, or else it would require itself to operate and would otherwise pollute the symbol space of the programs it link edits.

The **-Bsymbolic** flag forces *ld* to perform symbolic reduction operations using the information it has now, leaving only relative reductions to be performed − something *ld.so* resolves as part of its bootstrapping operations. The result should be a completely self-contained program, in which all symbolic references are satisfied by its own internal definitions. The **nosymbolic** assertion tests whether or not this is in fact the case.

7. Examples: Application Construction

To illustrate the use of the mechanisms by applications users, we will consider several simple examples of application program construction.

7.1. ‘‘Hello World’’

The classic simple C program is the one that simply prints ‘‘Hello world’’ on its standard output using *printf* and then exits. In an environment where the standard library path includes a *.so* form of the C library, the command

```
% cc -o hello hello.c
```

generates the *ld* command

```
% ld -e start -dc -dp -o hello /lib/crt0.o hello.o -lc
```

This *ld* command will cause the creation of the executable file `hello`. Since the default behavior of *ld* is to prefer the use of shared libraries, `hello` will be built as an ‘‘incomplete’’ executable requiring the inclusion of the library file `libc.so[.v]` (where `[.v]` represents the required version string) at execution time.

When the program is executed, *crt0* will discover the `__DYNAMIC` structure *ld* left behind and map in the execution-time linker, *ld.so*. *ld.so* will map in the appropriate version of `libc.so`, allocate any uninitialized commons required by the program, and cause unresolved procedure references in both `hello` and `libc.so` to call *ld.so*. The user’s call to *printf* invokes such a call, causing *ld.so* to search first the symbol table of `hello` and then `libc.so` for a definition of *printf*. The definition is found in `libc.so` and the *PLT* entry for the original call is updated to cause future references to go directly to *printf*. *printf* internally makes other calls to various parts of the C library, each of these intercepted and relocated by *ld.so*.

Although it might be argued that the relocations of intra-C-library calls could have been optimized by prebinding them. However, this would break interposition, as demonstrated by the next example.

7.2. Interposition

Consider the building of the program `hello` again, this time involving a special library, `libinterpose`. This library, like `libc`, is available in a *.so* form. The command used to build `hello` is:

```
% cc -o hello hello.c -linterpose
```

transparently invoking an *ld* command referencing `libinterpose` before `libc`. `libinterpose` defines entry points for various system calls, such as *read* and *write*, that in addition to invoking the required system call also take various statistics on the use of the system calls they surround.

As before, *ld.so* is invoked and maps in the two libraries, first `libinterpose` and then `libc`. The program calls *printf* requiring a relocation to the entry point in `libc`. Eventually, the code that implements *printf* and its descendents issues a call to *write*.

As previously noted, `libinterpose` defines an entry point for *write*. However, so does `libc`, as the standard interface for the *write* system call. *ld.so* resolves the ambiguity by using the ordering it established when mapping in *.so*’s, which places `libinterpose` first. Thus, `libinterpose` is effectively interposed for all uses of *write* in this program. If `hello` itself had defined a *write* entry point, it would have taken precedence over both `libinterpose` and `libc`.

7.3. Mixing Static and Dynamic Binding

Consider a program linked with two shared libraries, `liba` and (automatically) `libc`. A third library, `libb`, however it is only available in an archive, or *.a*, form. These are combined with the program `foo.c` with the command

```
        % cc -o foo foo.c -la -lb
```

`foo` references a procedure `bar` defined in both `liba` and `libb`. *ld* handles this problem by recognizing that `liba` contains a definition for `bar`, and ignoring the one provided in `libb`. Thus, even though the material from `liba` is not incorporated into the program until execution time, `libb` is prevented from contributing a definition.

However, suppose `foo` did not reference `bar`, but `liba` did and further, had no definition for it? In this case, *ld* would incorporate the definition from `libb`, and again the intent of the ordering on the command line is followed despite the difference in binding times.

8. Conclusions and Future Work

We have described the design of a shared libraries facility satisfying most of our goals, including:

- no kernel support specific to shared libraries or dynamic linking;
- transparency to application source code and build procedures;
- transparency to library source in higher-level languages; and
- no administrative procedures required to create or use shared libraries.

Some goals for transparency were only partially achieved, the most significant being the potential confusion to those using the system's debugging tools. The need to change some library assembly source is considered an acceptable minor shortcoming.

Although we have only limited experience with the implementation, early performance measurements indicate that we should meet our performance goals for ''average'' programs. These early measurements reveal:

- **PIC degradation**: the use of PIC in libraries does degrade execution time, although in many programs the degradation in negligible. The degradation is most noticeable in those programs that execute primarily in the libraries, and in some cases the degradation fails to meet our limits of 10%. However, we believe there are opportunities for improving the generation of PIC.
- **Start-up costs**: programs previously dominated by start-up costs and that use only a few libraries fall within our 50% goals. We have identified several areas for optimizing the start-up process, including caching the results of library searches. The start-up overhead for programs that use many libraries is unacceptably large, and is an area we are investigating.
- **Space reduction**: measurements over most of the system's standard utilities suggest that the average per-program savings from the use of shared libraries will be approximately 24K bytes.

During the time we obtained these early measurements, the new VM system on which the work was performed was also being debugged and shaken-out. The measurements were taken in a worst case environment where only the test programs employed shared libraries. Thus, any benefits or problems created by the dynamics of an environment that is based on shared libraries have not been determined, though it is expected that the sharing of the C library will have a positive impact.

Like most technologies, shared libraries and the mechanisms from which we build them can be abused. The execution-time loading we perform clearly has a cost, and excessive use of it in production programs may produce unacceptable performance. However, extensive use during program development adds a new element of flexibility that developers can use to enhance their development environment. An additional consideration is that a library is now a more powerful construct. Previously, the benefits of libraries were in the packaging they provided commonly used facilities. However, that packaging can now be used to provide performance and functional benefits as well.

Our future plans include:

- **Performance enhancement**: continuing efforts in this area for the foreseeable future. An area of particular interest is work to provide different space/time trade-off points than the two provided by the current implementation.
- **Common Link Editor source**: although they can be conceptually viewed as one, at present the two link editors *ld* and *ld.so* are implemented as separate programs. *ld.so* is particularly simplified, a short-cut taken to speed implementation of a first cut at the facility. We would

like to build both programs from a common source. Ideally, *ld* should just be an executable jacket to the common code in *ld.so*.

- **Programmatic interface**: some programs, particularly based on interpretive languages such as LISP, can dynamically generate dynamic references. We would like to support the handling of such references through a common mechanism, and thus wish to provide a program-accessible interface to the services now provided invisibly.
- **Different exception handling**: the current disposition of execution-time errors is to abort the program in which they occur. We would like to investigate the program development environments that might be created with other exception handling policies.

9. Acknowledgements

David Goldberg worked on the initial design for a shared libraries prototype that was the basis of this effort. Evan Adams and particularly Richard Tuck spent many hours discussing and helping to refine the philosophies of link editing employed in this design. Bill Joy provided advice and helpful criticism as well as several alternative and interesting implementation strategies. Bill Shannon was helpful in reviewing drafts of this paper.

10. References

[ARNO 86]	Arnold, J. Q., ‘‘Shared Libraries on UNIX System V’’, *Summer Conference Proceedings, Atlanta 1986*, USENIX Association, 1986.
[DOWN 84]	Downing, C. B., F. Farance, ‘‘Transparent Implementation of Shared Libraries’’, *UniForum Conference Proceedings, Washington DC*, /usr/group, January 1984.
[GING 87]	Gingell, R. A., J. P. Moran, W. A. Shannon, ‘‘Virtual Memory Architecture in SunOS’’, *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.
[MURP 72]	Murphy, D. L., ‘‘Storage organization and management in TENEX’’, *Proceedings of the Fall Joint Computer Conference*, AFIPS, 1972.
[ORGA 72]	Organick, E. I., *The Multics System: An Examination of Its Structure*, MIT Press, 1972.
[SZNY 86]	Sznyter, E. W., P. Clancy, J. Crossland, ‘‘A New Virtual-Memory Implementation for UNIX’’, *Summer Conference Proceedings, Atlanta 1986*, USENIX Association, 1986.