UNIX Programming

Shared Libraries in Sun OS

Douglas C. Schmidt

Washington University, St. Louis

http://www.cs.wustl.edu/∼schmidt/

schmidt@cs.wustl.edu

Overview

- *Shared libraries* share the code of functions that may be utilized by different processes running simultaneously
  - This extends the resource utilization benefits obtained from sharing code between processes running the same program to processes running different programs
- In general, the mechanisms in UNIX that support shared libraries are transparent to applications code and build procedures, as well as to library source code written in higher-level languages

Background

- UNIX has traditionally reduced memory utilization by sharing the *text* code of a program among all processes currently executing the program
  - However, on single-user workstations, the benefits from this type of sharing are rather small...
- Since program text typically contains *copies* of functions from one or more libraries there may be many duplicate copies of identical code competing for system resources
  - Obviously, this problem becomes worse with each system release!
- One approach to alleviating this problem is to extend the benefits of sharing by allowing different processes to share libraries common to them

Performance Issues

- Shared libraries represent a classic time vs. space trade-off, *e.g.*,
  - Primary and secondary storage utilization is reduced by deferring the incorporation of library code into an address space
  - However, this potentially increases program start-up time and may degrade certain types of data and function references
    - ∗ On the other hand, I/O time may be reduced and if the CPU time required to link the program dynamically does not exceed the I/O time saved, performance remains about the same
- Goals:
  1. Less than 10 percent degradation in performance for programs *not* dominated by start-up costs
  2. Less than 50 percent degradation in performance for programs that *were* dominated by start-up costs

Overview of Approach

- In Sun OS, shared libraries are built from several more basic mechanisms, *e.g.*,
  - Kernel-supplied file-mapping and "copy-on-writing" sharing at the page level
  - A revised link editor that supports dynamic linking
  - Compiler and assembler changes to generate position-independent code that need not be relocated for use in different address space arrangements and thus may be directly shared
- General goals (in order of priority)
  1. Maximize sharing
  2. Maximize flexibility
  3. Retain compatibility

Sharing Memory via File Mapping

- File mapping is the principle mechanism, accessed by programs and the dynamic linker via the `mmap` system call
  - A shared library is added to an address space using this general file mapping mechanism
    - *i.e.*, map the file as being "executable"
- Sharing occurs at the granularity of a "page"
- A per-page, copy-on-write facility allows the run-time modification of a shared object without affecting other users of the object

Overall Structure of New Linker

- The functionality of `ld` has been split into two parts
  1. Static `ld` -- runs at "static link-time"
  2. Dynamic `ld` -- runs at "dynamic link-time"
- Conceptually, the activities performed to link edit an executable program are independent of *when* the linkage occurs
- Therefore, the new `ld` facility is utilized at various times during the lifetime of a program to perform the link editing function over different scopes of material...

Overview of Old ld

- The old `ld` built all programs *statically*
  - Executable files contained complete programs, including all copies of necessary library routines
- Typically, a complete executable was built via a single batch operation
- `ld` would refuse to build incomplete executables unless the -r flag was given
- *e.g.*,

cc -o foo foo.c bar.o baz.a

Overview of New ld

- The new `ld` has been split into two parts
  1. *Batch link editor*, which generates
     - Simple object (.o) files by combining other .o's into a single larger one via -r
     - An executable file which is either a *program* (with a single application entry point) or a *shared object* (which does not have a single application entry point)
  2. *Dynamic link editor*
     - Performs additional run-time *symbolic* and *relative* reductions and relocations

New Behavior of ld

- The new `ld` will build "incomplete" executables
  - The incorporation of certain object files are deferred until some later time
    - Typically program execution...
  - The deferred link editing operations employ the system's memory management facilities to map to and thus share these objects directly
    - The extent of the changes that a link edit performs affects the degree to which sharing can occur
    - Modifications are accommodated automatically via the VM system using copy-on-write...

Types of Files

- `ld` will process the following kinds of input files
  1. Simple object files (.o's)
  2. Archives (.a's) which are conglomerates of simple objects (often called libraries)
  3. Shared objects (.so's), called shared libraries
     - A *shared library* consists of code and data built as a *shared object* (.so) file
     - A .so is simply an "incomplete" object file that lacks a single application entry point
     - Note, a .so file can be *any* object, and a library is simply one of many possible semantic uses for it

Summary of ld's Behavior

- Each .o file is simply concatenated to previous .o files in the order encountered
- Each .a is searched exactly once as it is encountered
  - Only entries matching an unresolved external reference are extracted and concatenated
- Any .so encountered is (usually) searched for symbol definitions and references but typically does not contribute material to be concatenated
  - However, .so shared libraries on the command line stored *in order* in the executable file and used at execution time

Useful Options to ld

- -B [**static**|**dynamic**]
  - Toggle the binding mode between **static** and **dynamic**
    - **static** forces static binding
    - **dynamic** allows dynamic binding, does not resolve symbolic references, and allows creation of execution-time symbol and relocation information
  - -Bdynamic is the default setting, therefore the use of shared libraries occurs from simply installing the .so that represents the shared library in the library search path used by `ld`

Useful Options to ld (cont'd)

- -L <path_name>
  - Specifies the pathname of a library to load
- -l <short_name>
  - 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
    - The specific suffix applied depends on the binding "mode" `ld` is operating in at the time the -l flag is processed
- -R <list:of:directories>
  - Colon-separated list of directories used to specify library search directories to the run-time linker

Useful Environment Variables to ld

- LD_LIBRARY_PATH
  - A colon-separated list of directory names
  - Augments `ld`'s built-in rules for which directories to search for libraries specified via the -l option
  - Placed *before* built-in rules, but after command-line -L options
- LD_OPTIONS
  - Specifies a default set of options to pass to `ld`
  - *e.g.*,

    % ld $LD_OPTIONS other ld arguments

Detailed Behavior of ld

- *Symbolic reduction*
  - After processing command-line options and input files, `ld` first tries to reduce all symbolic references to relative offsets within the executable
- *Relative reduction*
  - Attempts to transform all relative offset references to absolute addresses
  - Note that this requires an "entry point"

Detailed Behavior of ld (cont'd)

- If no further relocations are required after performing the reductions, `ld` produces a completely linked executable
- Otherwise, the executable will require further link editing at execution time in order to be useable
  - To accomplish this, `ld` deposits a object called __DYNAMIC in the data space of the partially linked executable
  - This object is used by the run-time linker to obtain .so's upon which this executable depends and to find the symbolic and relative reduction operations that remain to be performed
  - The object also enables evolution of shared library interfaces and implementations

Run-time Linking

- When a program is `exec'd` the system calls the `crt0` program prologue module to perform any additional dynamic linking and to call the `main` function
- If dynamic linking is required, `crt0` contains a non-NULL reference to __DYNAMIC
- This value is passed to the dynamic linker, which is mapped into the program's address space and invoked to finish the link editing operations required to build a complete executable
- When the call to the linker returns, all link editing has completed and the program may begin execution

Run-time Linking (cont'd)

- Each .so that must be added to the program's address space and link edited by the run-time linker is first mapped into the process address space
  - Note, the mapped address may vary on each invocation
  - Therefore, each instance of the shared library may map to a different location in different processes
- Then the run-time linker attempts to complete the link editing begun by `ld`
  - *i.e.*, symbolic and relative reductions on all external references are performed

Run-time Relocation

- Dynamic linking implies that absolute addresses are unknown until run-time
  - Therefore, the run-time `ld` is responsible for relocating external references
- If the dynamic link editing relocated external references by updating the text this would greatly reduce the ability to share code!
- The solution involves the creation and use of:
  1. *Global Offset Table* and a *Procedure Linkage Table*
     - Handled by batch `ld`
  2. *Position-independent code*
     - Done by compiler/assembler

Position-independent Code

- Code that minimizes:
  1. The amount of dynamic link editing
     - To improve speed...
  2. The extent of “changes” resulting from this editing
     - To promote sharing

- To enhance opportunities for sharing, the C compiler has been modifyed to generate position-independent code (PIC) optionally

- PIC needs editing only to relocate references to objects external to the body of code that has been built as PIC and is thus more sharable

Position-independent Code (cont’d)

- However, PIC programs will execute more slowly since global data and functions are accessed indirectly via *linkage tables*
  - This technique localizes the extent of the link editing process

- The specific amount of degradation is a function of the number of dynamic references to global objects

Global Offset Table (GOT)

- The GOT provides a level of indirection for external references

- `ld` generates a GOT for every dynamic executable and shared library

- The GOT is loaded in writable data since each process must relocate its external references

Procedure Linkage Table (PLT)

- Relocation may result in a significant start-up cost for programs using shared libraries

- However, many programs use only a small portion of the external symbols appearing in their text

- Therefore, *lazy evaluation* of procedure relocation is often an effective optimization

- The PLT is used to allow the static and dynamic link editors to perform lazy external function relocation

Procedure Linkage Table (PLT) (cont'd)

- The PLT avoids relocating the GOT entry for an external function until it's first referenced
  - The first call is routed via the PLT, which relocates the GOT entry
  - Subsequent calls use the GOT function entry directly
  - There's is an option that causes eager evaluation of all function references...
- `ld` generates a PLT for each shared library and dynamic executable
- The PLT is stored in read-only text.
  - Thus, it's shared among all library users

Compiler Changes

- The C compiler now takes a new option (-**pic**) that informs it to generate PIC
  - Each function prologue is extended to include the initialization of a register used as the base address of the shared libraries GOT, *e.g.*,

    movl #__GLOBAL_OFFSET_TABLE, a5
    lea pc@(0, a5:L), a5
  - Each reference to a global data object is generated as a dereference of a pointer in the GOT, *e.g.*, external integer **`errno`** is generated as

    movl a5@(_errno:w), a0
    movl a0@, d0
  - Each function call is generated as an assembler pseudo-operation including a free-register that will expand (if necessary) into a label in the PLT

    jbsr _foo, a0

Managing Different Versions of .so's

- The Sun dynamic linking facilities accommodate the evolution of shared libraries
  - In particular, it permits updates without impacting existing programs as long as interfaces remain compatible
- This is handled via major and minor "version numbers"
  - *e.g.*, /lib/libc.so.1.1

Managing Different Versions of .so's (cont'd)

- *Major numbers*
  - The major version used at execution time must exactly match the version found at `ld`-time
  - Major numbers change when interfaces change
- *Minor numbers*
  - The run-time linker will choose the highest minor version if multiple libs have the same major number
  - Minor numbers change to reflect compatible updates to libraries
    - The linker silently prefers highest compatible version possible

Developing Software using Dynamically Linked Programs

- In addition to improving the sharing of system resources, dynamic linking also offers several potential advantages in terms of increased flexibility for software maintenance and development
- Some aspects to consider, however
  - Dependence on multiple files
  - Semantic differences in order of library searching
  - Unusual debugging behavior

Three general types of problems with dynamic linking

1. *Deferred errors*
   - *e.g.*, unresolved symbols appearing at run-time
   - Solved via assertion mechanism
2. *Potential for chaos if important shared library is removed*
   - Solved by not building certain commands using shared libraries
     - *e.g.*, **`mv`**, **`ln`**, **`ls`**, **`tar`**, **`restore`**, **`init`**, **`getty`**, **`shells`**
3. *Potential security problems with setuid programs*
   - Solved by not building setuid programs with shared libraries

Assertions in Sun OS 4.x

- To help deal with potential complexities of dynamic linking, a set of assertion facilities have been developed
- Assertions in Sun OS 4.1.x
  - -assert definitions
    - If program was run now there would be no undefined symbols
  - -assert pure-text
    - The executable requires no further relocations

Assertions in Solaris 2.x

- -z defs
  - Force a fatal error if any undefined symbols remain at the end of the link (default for executable)
- -z nodefs
  - Allow undefined symbols (default for shared objects)
- -z text
- In dynamic mode only, force a fatal error if any relocations against non-writable, allocatable sections remain.

Examples

- Building a shared library under Sun OS 4.1.x

```
% cc -pic -o foo.c -c foo.c
% /bin/ld -assert pure-text foo.o -o foo.so.1.0
```

- Building a shared library under Solaris 2.x

```
% cc -pic -o foo.c -c foo.c
% /usr/ccs/bin/ld -G -o foo.so.1.0
```


A Simple C Example

- *e.g.*,

```
#include <stdio.h>
int main (void)
{
      printf ("hello world\n");
}
```

- This is compiled as

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

- Which generates the `ld` command

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


A Simple C Example (cont'd)

- `hello` is built as "incomplete executable"
- When `hello` is executed, `crt0` discovers __DYNAMIC structure `ld` left behind and maps in the run-time linker
- `ld` will map in the appropriate version of `libc.so`, allocate uninitialized commons, and cause unresolved procedure references in `hello` and `libc.so` to call the run-time linker
- When `printf` is called, the linker is invoked again to search the symbol table of `hello` and `libc.so` for an appropriate definition

Interposition example

- Consider a variation on the previous example where another shared library is linked ahead of libc.so
  - /usr/lib/interpose.so
- *e.g.*,

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

- The interpose library defines entry points for various system calls (such as `read` and `write`) that invoke the required system calls *and* keep track of certain statistics
- Since `interpose.so` appears before `libc.so`, the code that implements `printf` will call the `write` function in `interpose` rather than `libc.so`

Mixing static and dynamic linking

- Consider the following

  % cc -o foo foo.c -llib_dynamic -llib_static

- If `foo` references a function `bar` defined in both `lib_dynamic` and `lib_static` which gets called?
  - Note that material from `lib_dynamic` is *not* incorporated into `foo` until run-time!!!!
  - However, the order of libraries on the command-line still determines which definition takes precedence

Direct Access to Dynamic Linking

- The `dl*` family of functions provide user programs direct access to dynamic linking
  - **void** *dlopen (**char** *pathname, **int** mode);
    - Maps *pathname* into the address space of the program
    - *mode* is either RTLD_LAZY or RTLD_NOW
  - **void** *dlsym (**void** *handle, **char** *name);
    - Returns a pointer to a *named* symbol in the library indicated by handle
  - **void** *dlclose (**void** *handle);
    - Deletes the shared library associated with *handle* from the process address space
  - **char** *dlerror (**void**);
    - Returns an error-reporting string

Direct Access to Dynamic Linking (cont'd)

- *e.g.*,

```
/* file decimal.c */
int print (int a) { printf ("%d\n", a); }

/* file hex.c */
int print (int a) { printf ("%x\n", a); }

/* file main.c */
char *mylibs[] = {"./decimal.so", "./hex.so", NULL};
int main (void) {
  void (*func)(int), *handle;
  int i = 0;
  while (mylibs[i]) {
    handle = dlopen (mylibs[i], RTLD_NOW);
    if (!handle) printf ("%s\n", dlerror ()), exit (1);
    func = (void (*)(int)) dlsym (handle, "print");
    if (!func)
      printf ("%s\n", dlerror ()), exit (1);
    (*func) (38);
    (void) dlclose (handle);
    i++;
  }
  exit (0);
}
```


Summary

- The support for dynamic linking in Sun OS is characterized by
  1. No kernel support specific to shared libraries or dynamic linking
  2. Transparency to application source code and build procedures
  3. Transparency to library source code in higher-level languages
  4. No complicated administrative procedures required to create or use shared libraries