Pearls from the ABI — Dynamic Shared Libraries

I am sure you have heard the term Application Binary Interface (ABI). It is one of those buzzwords we cannot escape in these times of standards and "One UNIX system for all the masses." Simply stated, it means that an executable compiled on one machine architecture should run without recompilation on another machine of the same architecture. This seems intuitive and is likely true if the machines are from the same manufacturer because this implies the same UNIX® 'port'. But, different manufacturers using the same machine architecture are free to port the UNIX system in their own way. For example, ABC Microsystems' UNIX port to their Intel 80386 box is not guaranteed to be identical to XYZ Microsystems' port to their 386 machine. In pre-SVR4 days and with the various flavors of the UNIX system, this was chaos.

When an application makes a call to a system call such as `read()`, `read()` calls a function called `systrap()` with an index. This function, `systrap()`, is the real entry-point into the kernel. The index passed to the kernel via `systrap()` represents the service being requested (e.g. 'read' or 'write'). Different UNIX versions and ports can use different `systrap()` indices. That is, the ABC Microsystems' kernel recognizes 4 as the index for 'read' where the XYZ kernel uses 5. An executable compiled and statically linked on one company's machine would have undefined results if executed on the other company's box. The operative phrase here is *statically linked*. Statically linked executables contain the code for the functions they use. This includes system calls. The version of the system call code is appropriate only to the machine on which they were compiled and linked.

New to SVR4 are Dynamic Shared Libraries (DSL). DSLs are the fundamental solution to providing Application Binary Compatibility. Now an executable linked with the DSL version of `libc` (which is where the code for the system calls is located) is guaranteed to run across UNIX boxes of the same architecture. This is due to the machine specific version of `read()` being dynamically linked in at execution time.

A Solution with Utility

The beauty of DSLs is that not only do they facilitate the the ABI but that they are an elegant solution with utility far beyond that of static linking. Applications linked with DSLs:

- Are automatically updated when a new library is installed.
- Save physical disk space.
- Contribute to overall system performance.

Consider all the commands that reside in `/usr/bin`. Most of these commands are compiled and linked with the DSL `libc.so`.[1] When a command like `pr` is executed, the DSL is mapped into the process space, and then subsequent calls to `stdio` functions like `printf()` are dynamically resolved the first time they occur. If a bug were later found in the `stdio` portion of `libc.so`, a new library could be built and distributed to customers. Applying the bug fix would simply require replacing the `libc.so` on your system. All the applications that made use of the library would pick up the fix the next time they were executed.

Dynamic shared libraries also save space. Again, consider the commands that reside in `/usr/bin`. Let us assume they are all linked with `libc.so` and use the `stdio printf()` function. If these executables were all linked with the static version of `libc`, they would all contain the code for `printf()` and whatever other code `printf()` uses. Furthermore, let us suppose that they are all executed simultaneously. This would result in the kernel creating separate physical pages in memory containing identical `printf()` code. Excessive paging can have a significant detrimental effect on overall system performance. Since the `/usr/bin` commands are all linked with `libc.so` they all share the same code from the same physical location on disk as well as in the kernel. So, not only is space conserved but system performance is improved.

Easy to Use

Unlike the static shared libraries of pre-SVR4, DSLs do not require any special coding techniques. This is due to improvements in the SVR4 C compilation system. Aside from compiling ANSI C compliant code, the new CCS features a Position Independent Code (PIC) generator and a new object file format. The new object file format ELF (Executable and Linking Format), replaces the old COFF format and is also a fundamental part of an efficient and elegant ABI

1. The '`.so`' suffix stands for 'shared object'.

solution. ELF facilitates DSLs by providing a richer format for expressing what the process image of an executable should look like.

Creating a DSL is very simple. The command:

```
cc -G -K PIC file_1.c file_2.c file_3.c -o libfoo.so
```

creates the DSL `libfoo.so` from files 1, 2 and 3. The `-G` flag specifies that a DSL is to be created and the `-K PIC` option tells the compiler to generate position independent code. DSLs can be created out of non-PIC code but the benefits of sharing are lost.

Performance is Everything

The real win you get from using DSLs is improved overall system performance. By reducing the dynamic system memory requirements, improvements on the order of 20-30% in system performance can be achieved. This is because if the kernel needs to create and manage fewer pages, these page faults, which result in memory pages being read and written to disk, are less likely.

There is some overhead associated with the dynamic linker. The dynamic linker is the utility that maps the DSLs used by your process into your process space at execution time before `main()` receives control. As calls are made to functions in the DSL, the dynamic linker intervenes to resolve references. This *lazy-evaluation* occurs only once for each function called. Thereafter, function calls go directly to the function being called. The default lazy-evaluation method can be overridden at run-time via an environment variable. Setting `LD_BIND_NOW=1` will result in all function references being resolved before `main()` receives control. The down-side to this is that references get resolved for functions that may not be called.

The execution of a process that is linked with a static library such as `libc.a`, may out-perform the execution of the same process linked with an equivalent DSL. The timing statistics, both system and user, of 20 users simultaneously executing a statically linked `pr` will be less than the timing statistics of 20 users executing a dynamically linked `pr`. All users share the same read-only memory pages in both examples, but the second group has the overhead of the dynamic linker. But, consider 20 users simultaneously executing 20 separate statically linked commands from `/usr/bin`. The timing statistics gathered here will be greater than 20 users simultaneously executing the same dynamically linked commands. The reason is that: the overhead of the dynamic linker is cumulatively small compared to the overhead of the kernel creating and faulting

on the 20 (or more) separate pages it must manage for each version of the library code in the statically linked processes. That is, each statically linked process will have a copy of the library code it uses in memory at execution time. The dynamically linked processes will be sharing one copy. Therefore, the kernel creates less pages, and less page faults are incurred.

Although creating a DSL does not require special coding techniques, to reap the full complement of benefits for using DSLs there are some guidelines one should follow when writing functions that are to be part of a DSL.

- *Minimize the Library's Data Segment*
  All executables and shared objects (i.e. DSLs) have a text segment (read-only data such as executable code) and a data segment (writeable data). The text segment of all executables and shared objects is shared amongst all processes executing those objects. However, the data segment is not shared and is considered private writeable space for each process. Furthermore, space is set aside in the executable for the references to global data in the DSLs with which it is linked. At run-time, this writeable data is copied into the process space. Therefore, minimizing the data segment of the shared library reduces the run-time overhead of copying this writeable data into the process' space and reduces the number of non-shared pages the kernel creates for the client process.

  There are several approaches to minimizing the data segment of a DSL.

  1. Use automatic (stack) variables versus static storage. For example,

     ```
     int
     foo (char *arg1)
     {
             char        buf [256];

             ...
     }
     ```

     performs better than

```
static        char        buf [256];
int
foo (char *arg1)
{
        ...
}
```

2. Use functional interfaces versus global variables. This serves two purposes: 1) in general it makes the code easier to maintain; and 2) it eliminates the copy of global data from the shared library into the users data segment at execution time. This latter point is especially important when there is tabular data embedded in the library. Consider an executable that is linked with a DSL that has a global table. A place for that table at run-time is statically reserved in the executable at link-time. There are two penalties for this: 1) there is a performance hit for copying the table from the DSL into the users data segment at run-time; and 2) if a new DSL should be installed with a larger table, the application, unless re-linked, will not see these changes beyond the previous size of the table.

   As an example,

```
static  const char *  table [] = {
        "foo string 0",
        "foo string 1",
         ...
        "foo string N",
};

const char *
fooString (int i)
{
        if (i < 0 || i >= sizeof (table)/sizeof (table[0]))
                return  0;

        return  table [i];
}
```

   is preferred over

```
char *  table [] = {
        "foo string 0",
        "foo string 1",
         ...
        "foo string N",
};
```

In addition, note the use of the type qualifier `const` in the preferred example. This identifies `table` as read-only data. Read-only data is stored in the shared object's text segment. This is another performance plus over writeable data. It is stored in the shared object's data segment and, as pointed out earlier, every user of the shared object gets a private copy of its data segment.

3. A corollary to the last item would be to exclude functions from the DSL that use large amounts of global data if they could not be rewritten.

4. Endeavor to make a DSL self-contained with respect to global data. If a shared library directly references global data in another shared library then even more data space is reserved and copied into the process' data segment at run-time (as per item 2).

- *Minimize Paging Activity*
  Processes that use shared libraries may still incur page faults on the shared read-only text pages. Page faults will degrade performance. There are a few techniques to use in writing a shared library to minimize paging activity. These methods are rather advanced and mostly related to the specific architecture of the machine on which the library will reside. Therefore, since such architecture specific topics can bring a lighthearted article like this to its knees, they will not be discussed here. They are discussed in [1].

Even More Crafty Uses

As if the things I have described already aren't enough reason to use DSLs I offer another example of their crafty use. Applications that are linked with DSLs can be said to use these DSLs 'implicitly' at run-time. But, it is also possible to use DSLs 'explicitly'. That is, you may have 2 or more DSLs that contain functions of the same names and interface syntax (i.e., calling arguments and return values) but the functions of one library do something semantically different than the same named functions in one of the other libraries. Your application is not dynamically linked with any one of these libraries. Instead, at run-time, the

application can decide which semantics are desired and call the dynamic linker directly to link in the appropriate library.

So, your probably saying to yourself: "Cool. But, why would I want to get this weird? I have a normal social life." Other than the fact that we sometimes get paid to be weird I offer the following real world example. A new feature to SVR4 is *Network Selection and Name-to-Address Mapping*. Given that SVR4 supports numerous network providers it provides a mechanism for finding a particular service on a specific host across a specific network provider. DSLs play a very clever part of this system service.

Different transports (i.e., network providers like TCP and STARLAN) use different address formats as well as certain conventions. That is, network addresses have a human readable form and a machine readable form. The system file `/etc/inet/hosts` contains the human readable addresses and corresponding host names accessible using TCP/IP. The entry `192.11.109.144 snafu` represents the main address of the system `snafu`. Service ports are also transport dependent. The system file `/etc/inet/services` contains known port and service names. The entry `fubar 6666/tcp` represents that service `fubar` can be found on port `6666`. This address and port are combined in a transport specific manner to produce a transport specific address of the service `fubar` on the system `snafu`.

Now consider an application that would like to connect to service `fubar` on the system `snafu`. This application would start by using a set of functions called `netconfig` functions, to find a network provider on the local system that is suitable for use. `netconfig` functions parse the system file `/etc/netconfig` similar to the way the `getpwent()` functions parse `/etc/passwd`. Once a suitable network provider was found, the `netdir` functions would be used to find the address of service `fubar` on the system `snafu`. `netdir` functions do name-to-address mapping. Given a network provider, a host name and a service name `netdir_getbyname()` will search the appropriate system files and return the transport specific address of the requested service on the named host.

So, where do DSLs figure into all this? Consider the format of `/etc/netconfig`. The fields of this file are:

- The name of the transport (e.g., `tcp`, `starlan`).
- The semantics, like connection-oriented or connection-less, of the transport (e.g., `tpi_cots`, `tpi_clts`).

- Flags. The only flag currently defined is `v` for whether the transport is available for use or not.
- The protocol family of the transport (e.g., `inet`, `osi`).
- The protocol name of the transport (e.g., `tcp`, `udp`).
- The device to use for access to the transport (e.g., `/dev/tcp`, `/dev/starlan`).
- The path of the DSL to use to do the provider dependent name-to-address mapping (e.g., `/usr/lib/tcpip.so`, `/usr/lib/starlan.so`).

Now consider the following code fragment.

```
#include <netconfig.h>
#include <netdir.h>

struct nd_addrlist *
FindServiceAddress (char *hostp, char *servicep)
{
     void *  handlep;

     struct netconfig *   ncp;
     struct netbuf *      nbp;
     struct nd_hostserv   ndhs;
     struct nd_addrlist * ndalp = (struct nd_addrlist *) 0;


     handlep = setnetconfig ();

     while (ncp = getnetconfig (handlep))
     {
          /*
          **  'ncp' holds a parsed record from
          **  /etc/netconfig.
          **  Is it a transport we can use?
          */
          if (ncp->nc_semantics != SEMANTICS_WE_NEED)
          {
               /*
               **  Wrong communication semantics.
```

```
                */
                continue;
            }
            if (strcmp (ncp->nc_protofmly, PROTO_FAMILY_WE_NEED))
            {
                /*
                **  Wrong protocol-family.
                */
                continue;
            }
            ndhs.h_host = hostp;
            ndhs.h_serv = servicep;
            if (netdir_getbyname (ncp, &ndhs, &ndalp))
            {
                /*
                **  The service on the host we requested
                **  cannot be contacted on this provider.
                */
                ndalp = (struct nd_addrlist *) 0;
                continue;
            }
            /*
            **  We found a host with the service we want
            **  on the provider we can use.  So return
            **  the address of the service.
            */
            break;
        }
        endnetconfig (handlep);
        return  ndalp;
    }
```

The dramatic role that DSLs play in all this is in the `netdir_getbyname()` function. This function uses the `netconfig` record pointed to by `ncp` to find the correct DSL that searches the correct files (e.g., `/etc/inet/hosts` and `/etc/inet/services`) and does the transport dependent mapping of host and service, to network address. The important point here is that the application calling `FindServiceAddress()` will be dynamically linked with `libnsl.so`, the Network Services Library which contains the `netconfig` functions and the facade for the `netdir` functions. But, `libnsl.so` does not contain the text of

the functions `netdir` uses and the application is not linked with any of the DSLs that do. The explicit dynamic linking is done by the `netdir_getbyname` function based on the content of the `/etc/netconfig` entry passed to it.

The explicit use of DSLs saves space by relieving `libnsl.so` from having to contain the text of all the transport specific name-to-address mapping functions of all possible network provider. Furthermore, functionality is enhanced since network applications will be able to make use of new network providers as they are added to the system.

In Closing ...

Dynamic shared libraries are not an absolute panacea. In fact, DSLs can be used improperly. Some commands that are small and whose execution is typically short lived can negatively impact system performance by using DSLs. Despite this, DSLs are an excellent solution to many application engineering problems. Not the least of which is application binary compatibility.

Bibliography

[1] *UNIX System V Release 4: Programmer's Guide: ANSI C and Programming Support Tools,* Prentice Hall, Englewood Cliffs, NJ, 1990.

[2] *UNIX System V Release 4: Programmer's Guide: Networking Interfaces,* Prentice Hall, Englewood Cliffs, NJ, 1990.

[3] *UNIX System V Release 4: Programmer's Reference Manual,* Prentice Hall, Englewood Cliffs, NJ, 1990.