Callable shared objects (CSO), self-contained callable shared objects (scCSO), COBOL shared libraries, shared libraries are all examples of what Unix thinks of as shared objects. There is no architectural difference between a shared object and shared library. There are simply different conventions about their naming and use. Much 3rd party code, e.g. Oracle, is supplied as one or more shared objects.

Only one copy of the text (code) of a shared object is present in memory, no matter how many processes and executables reference it. This code sharing makes shared objects efficient and attractive. Shared objects save disc space, as there is only one copy of the code on disc. Shared objects also allow old programs to pick up fixes without the need for recompilation or relinking.

This document details various features of shared objects of which you need to be aware.

**Definitions**

**module**
A partial executable or shared object.

**partial executable**
A partial executable, aka dynamic executable, aka dynamic-linked executable, does not contain all the code it requires to run. Undefined symbols within the partial executable must be resolved before they are used. A partial executable references shared objects that are required to be loaded when it is loaded. This relationship is known as a dependency, the referenced shared objects known as children and the referring module known as the parent. Note that parents depend upon children in this taxonomy, not the other way around.

**shared object**
A shared object is similar to a partial executable, but cannot be used to initialize a process. A shared object can be logically located at any address in a process. Shared objects may depend upon other shared objects. Shared objects can be loaded as a result of a dependency from another module, or explicitly by function call.

**PA-32**
32 bit PA-RISC HP-UX

**PA-64**
64 bit PA-RISC HP-UX

**ELF**
Executable and Linking Format. The object module format used on all Unix platforms except PA-32 and AIX. Not to be confused with DWARF or GNOME.

**XCOFF**
eXtended COFF (Common Object File Format). Object module format used on AIX. An IBM extension of COFF; a predecessor of ELF.

**SOM**
Spectrum Object Module. The object module format used on PA-32.

**Resolution/Binding**

When a module is loaded, after its dependencies (and their dependencies and so on) are loaded, then any symbols originally undefined need to be resolved. This can occur immediately, which is known as immediate binding, or can occur upon first use, which is known as deferred or lazy binding. Data symbols always use immediate binding. Some variations of Unix, notably AIX, have limited or no support for lazy binding.
Lazy binding means a process can continue even though some symbols are not defined. This is of benefit where it is known that at runtime a set of symbols will never be required. If an unresolved symbol is referenced at run-time an error occurs and the process terminates. Lazy binding leads to quicker initialization. There is a performance penalty when the symbol is resolved.

Immediate binding resolves all symbols when the module is loaded. Load time is increased. The possibility of run-time errors due to unresolved symbols is removed, but all the symbols do need to be defined.

Partial executables, shared objects listed in the LD_PRELOAD environment variable, and their direct and indirect dependencies use lazy binding by default, except on AIX. Linker options and environment variables can change this. Shared objects loaded through COBOL by set proc-ptr to entry use immediate binding. Shared objects loaded using the C API dlopen() specify the binding mode to use as part of the call.

Lazy binding and lazy loading are different, though some sources use the terms interchangeably. Lazy loading is often used in conjunction with lazy binding. A lazily loaded dependency is not loaded when its parent is loaded. Loading is deferred until a symbol it defines is referenced.

**Locating**

Shared objects are referenced by name. If that name includes "/" then only that path (relative or absolute as appropriate) is searched. Otherwise the dynamic linker searches directories for the shared object in the following order

1. a path embedded in the depending module (old ELF only – deprecated and no longer used by linkers but still supported by the runtime loader)
2. a path specified by an environment variable – LIBPATH (AIX only), SHLIB_PATH (PA-32), LD_LIBRARY_PATH or SHLIB_PATH (PA-64), LD_LIBRARY_PATH (all others)
3. a path embedded in the depending module
4. in the default list
5. in the list of directories specified in /etc/ld.so.conf (ELF only)

Note that, unlike Windows, the current directory is not automatically searched for shared object. The current directory needs to be explicitly specified by one of the above methods.

A path embedded in a module only affects the search for shared objects that are direct dependents. It does not affect grandchild shared objects, nor does it affect the search for shared objects loaded at run time by code in that module (e.g. by dlopen() or COBOL syntax). The checker directive initcall is a runtime artefact.

The path embedded in a module is constructed by the linker, typically from –L parameter passed to it, from specific other linker options, and appropriate environment variables. This varies greatly from platform to platform. Please read the ld documentation for your platform if you require more information. The embedded path is sometimes referred to as the rpath or runpath. The embedded path may contain the pseudo directory name $ORIGIN which evaluates at run time to the directory where this module was located. Including $ORIGIN in the embedded path of a module means that its dependents are searched for in the directory where their parent was located.

The environment variable that specifies where to search for shared objects is analogous to the PATH environment variable used to locate executables and shares a similar syntax – a colon separated list of directories. In order to run COBOL, one generally needs to include $COBDIR/lib, so a typical setting would be

```
export LD_LIBRARY_PATH=$COBDIR/lib:/opt/oracle8.1/lib:$HOME/lib
```

which will cause the dynamic loader to search for shared objects in those 3 directories, in that order. References to LD_LIBRARY_PATH in this document generally apply equally to LIBPATH and SHLIB_PATH unless contra-indicated.

The default path varies from platform to platform, and can vary between 32 and 64 bit. It is generally something like /usr/ccs/lib:/usr/lib, or for 64 bit /lib64:/usr/lib64.
**Referencing and Preloading**

A shared object referenced in the link line is automatically included in the module produced, even if it does not satisfy an undefined reference. This is the easiest way to reference a shared object and cause it to be loaded. Shared objects can be referenced using the cob and ld options –L and –l, or referenced directly – eg:

```cob
cob -x a.o -L/home/jw/mylib -lsupp       # Produce a reference to libsupp.so
# and put /home/jw/mylib in the embedded path
```

```cob
cob -x fred.o shr.o                      # Produce a reference to shr.o
```

The second example does not work particularly well on HP-UX due to a ‘feature’ of the linker.

If you cannot relink then on Linux, SVR4 and HP-UX you can use the LD_PRELOAD environment variable. This is a space separated list of shared objects that will be searched for and loaded at process initialization, as if they had been specified on the link line of the executable. LD_PRELOAD_ONCE achieves the same, but is not inherited by child processes.

---

**Security Concerns and setuid/setgid**

A process on Unix may require more permissions than the user invoking the process has. For instance: we do not want a user to have general write permission to a database, but we do want the stock control program run by that user to be able to update the relevant files. Unix systems allow this using **setuid/setgid**. In this document the rules and guidelines for setuid programs also apply to setgid programs.

Most of the C runtime exists in shared objects. All processes, including setuid processes, access the C runtime. An attacker could use this to create their own shared object defining C library functions and set LD_LIBRARY_PATH to find this shared object.

Unix prevents this by restricting the functionality of environment variables and embedded paths for setuid executables.

- AIX ignores LIBPATH if the executable is setuid, and also clears the value of LIBPATH.
- HP/UX applies its restrictions if the effective user-id is not the same as the real user-id, or the effective group-id is not the same as the real group-id. This means the restrictions apply to a setuid process and its child processes. On HP/UX, enhanced privilege processes ignore LD_PRELOAD. Directories in SHLIB_PATH or LD_LIBRARY_PATH are only searched if those directories are also listed in /etc/dld.so.conf.
- SVR4 ignores LD_PRELOAD and LD_PRELOAD_ONCE if the executable is setuid. SVR4 will only search a directory specified in LD_LIBRARY_PATH if that directory is trusted.
- Solaris in addition to SVR4 restrictions, applies these restrictions to the children of setuid processes (cf HP/UX). Solaris will however load shared objects in LD_PRELOAD if the name does not contain “/”. These shared objects are only searched for in the trusted directories.
- Linux follows SVR4.

Security hardened systems can have even more restrictions.

In brief – setuid executables by default do not use the shared library search path environment variable. As COBOL executables need to locate the runtime in $COBDIR/lib this makes setuid COBOL executables problematic. In particular if the COBDIR used when to link the application is not in the same place as the COBDIR used at runtime then the COBOL libraries will not be found. Possible solutions are to link the COBOL shared libraries into the default search directories, to add $COBDIR/lib to the default list of directories to search, or to add $COBDIR/lib to the list of trusted directories. All of these require admin level typically root permission. Specification of default directory and trusted directory is very platform specific. See the links in Further Reading.
**Loading shared objects at run time**

At the OS level a shared object is loaded into the address space of a running process by using the `dlopen()` API. This is available on all Unix platforms on which Server Express 5.0 is available. Not all the functionality is available on all systems. HP-UX PA-32 also supports an older interface based around `shl_load()`. AIX additionally supports the `load()` interface.

dlopen() allows one to specify the visibility of symbols in the newly loaded shared object, how the shared object searches to resolve any undefined symbols, and whether to use immediate or lazy binding.

At the COBOL level, a shared object can be loaded by using

```
set proc-ptr to entry "fred"
```

This will search the shared library search path for a shared object with basename "fred". `set proc-ptr to entry needs to return a valid procedure-pointer`. Therefore `fred.so` must contain an entry "fred", or must be a CSO created by cob using the –e entry-point option. This entry point is then returned. Shared objects, and their dependencies, loaded using COBOL syntax will use immediate binding, the scope of the symbols they define is global, and they can search the entire process to resolve undefined symbols.

Third party shared objects may not satisfy either of these requirements. The easiest way to solve this is to create a stub CSO that references the third party shared object. Loading this CSO will load the third party shared object. So, for instance, to load at runtime third party shared library `libXYZ_supp` that resides in `/opt/XYZ/lib` create a stub CSO using the following cob commands

```
touch stub.cbl
cob –z stub.cbl –L/opt/XYZ/lib –lXYZ_supp
```

and then load the stub at runtime with the following COBOL statement

```
set proc-ptr to entry "stub"
```

**Environment Variables**

**LIBPATH**
A colon separated list of directories which the dynamic loader searches at run time to locate shared objects. LIBPATH is AIX specific. SHLIB_PATH is used on PA-32, and PA-64 if LD_LIBRARY_PATH is not specified.

**SHLIB_PATH**
LD_LIBRARY_PATH has the same syntax and function as LIBPATH/SHLIB_PATH, and is used on all platforms except AIX and PA-32. Additionally on strict SVR4 systems, including Solaris and Unixware, but excluding Linux and HP/UX, LD_LIBRARY_PATH is also used at link time. Its syntax here takes the form of two lists of directories. The lists themselves are separated by a semi-colon. e.g

```
LD_LIBRARY_PATH=dirlist1;dirlist2
```

If the semi-colon is omitted the list is taken as `dirlist2`. At link time the directories in `dirlist1` are searched first, then directories specified by `-L` options and then `dirlist2`. Both lists are searched when LD_LIBRARY_PATH is used at run time by the dynamic loader.

**LD_RUN_PATH**
Solaris and Unixware only. LD_RUN_PATH, like most *PATH environment variables, is a colon separated list of directories. The value of LD_RUN_PATH is used by the linker to create the embedded path for a module. It can be overridden by linker options: e.g. `ld –R on` Solaris.
LD_PRELOAD
LD_PRELOAD_ONCE
   Not available on AIX. A space separated list of shared objects loaded at process initialization. LD_PRELOAD is inherited by and applies to child processes. LD_PRELOAD_ONCE is not.

LD_BIND_NOW
   SVR4 and Linux only. If set to a non-null value, e.g. export LD_BIND_NOW=yes, then immediate rather than lazy binding is used.

LD_NOLAZYLOAD
   SVR4 and Linux only. When set to a non-null value lazy loading is disabled.

LD_WARN
   Linux only. If set to non-null value warn about undefined symbols.

LD_DEBUG
   Linux, Solaris. Output verbose debugging information about the dynamic linker. See the platform specific documentation for more details.

Solaris also has some variant environment variables with _32 and _64 suffixes which only apply to 32 bit and 64 bit executables – e.g. LD_LIBRARY_PATH_64. The bitism specific environment variable, if set, takes precedence over the generic variable.

Utilities

ldd
   All systems. List the dynamic dependencies of a module.

dump, odump, elfdump, objdump, readelf
   Display the contents of an object file or module. odump is PA-32 specific. elfdump is used on Solaris. objdump is used on GNU/Linux. Some Linux and SVR 4 systems use readelf. Different versions of dump exist on AIX and Solaris.

chatr
   HP-UX utility to display and changes the attributes of a module.

procldd, pldd
   procldd (AIX) and pldd (Solaris) display the shared objects mapped into a running process.

Other topics

A module depends on shared objects, which in turn can depend on other shared objects and so on. This conceptually creates a tree of dependencies with the original module at the root. These can logically be loaded and searched for symbols in different orders – either breadth-first, or depth-first – and searched for symbol. Generally speaking this should not make a difference, though it might be something to consider when migrating applications. Standard SVR4 behaviour is breadth-first order for symbol resolution. PA-32 uses depth first. Breadth-first can be specified for a PA-32 executable using the +compat linker option. COBOL executables will default to the platform default.

Miscellaneous gotchas

Beware of setting or clearing LD_PRELOAD or LD_LIBRARY_PATH in your shell environment file (typically .kshrc for Korn Shell, .cshrc for C shell, .bashrc for bash). Scripts, or anything that invokes a subshell, such as gdb, will inherit the values set in the environment file rather than your current environment. Only set LD_LIBRARY_PATH explicitly, or in your login file (e.g. .profile, .login etc).
Further Reading

Dynamic linking on AIX is significantly different to other Unix platforms. Please see here for all the details.

Both Sun and HP have excellent documentation sites. SCO has significant documentation about dynamic linking here. The Linux man pages are here, with specific info regarding the dynamic linker here.

NOTE: this is based on Knowledge Base article reference: 25638

1 OK – so it’s not quite that simple. Both AIX and Solaris allow you to have a shared object inside a static archive. The static archive is referenced on the link command line, and the library/shared-object combination saved in the module.

2 On AIX a partial executable can be loaded into a running process like a shared object. A shared object however cannot be used to initiate a process.

3 DWARF - Debug With Attributed Record Format (allegedly). A format for storing debugging information in an object file.

4 GNOME – A GUI desktop for Linux

5 Actually it will first search the currently loaded symbols for "fred", then it will search for a suitable shared object. After that it will search for .gnt, .int and .lbr files.