Appendix D. Shared Libraries

Authors: *Christopher Hylands*
Alain Girault

D.1 Introduction

Shared libraries are a facility that can provide many benefits to software but have a slight cost of additional complications. In this appendix we discuss the pros and cons of shared libraries. For further information about shared libraries, you should consult the programmer's documentation that comes with your operating system, such as the Unix `ld` manual page.

D.1.1 Static Libraries

A static library file is a file that consists of an archive of object files (`.o` files) collected into one file by the `ar` program. Static libraries usually end with `.a` (i.e., `libg++.a`). At link time, static libraries are searched for each global function or variable symbol. If the symbol is found then the code for that symbol is copied into the binary. In addition, any other symbols that were in the original `.o` file for the symbol in question are also copied into the binary. In this way, if we need a symbol that is dependent on other functions in the `.o` file in which it is defined, at link time we get the dependent functions. There are several important details about linking, such as the order of libraries, that should be discussed in your system documentation.

D.1.2 Shared Libraries

Most modern operating systems have shared libraries that can be linked in at runtime. SunOS4.x, Solaris2.x and HPUX all have shared libraries.

Shared libraries allow multiple programs to share a library on disk, rather than copying code into a binary, resulting in smaller binaries. Also shared libraries allow a binary to access all of the symbols in a shared library at runtime, even if a symbol was not needed at link time.

A shared library consists of an archive of object files (`.o` files) collected into one file by either the compiler or the linker. Usually, to create a shared library, the `.o` files must be compiled into Position Independent Code (*PIC*) by the compiler. The compiler usually has a special option to produce PIC code, under `gcc`/`g++`, the `-fPIC` option produces PIC code. Shared libraries have suffixes that are architecture dependent: under SunOS4.1 and Solaris, shared libraries end with `.so` (i.e., `libg++.so`); under HPUX, shared libraries end with `.sl` (i.e., `libg++.sl`).

In addition, shared libraries can also have versioning information included in the name. Shared library versioning is architecture dependent, but a versioned shared library name might look like `libg++.so.2.7.1`. Note that the version of a shared library can be encoded in the shared library in the `SONAME` feature of that library. Usually, the `SONAME` of a library is the same as the filename (i.e., the `SONAME` of `/users/ptolemy/gnu/sol2/lib/libg++.so.2.7.1` would be `libg++.so.2.7.1`). Interestingly, if you rename a shared library without changing the `SONAME` and then link against the renamed shared library, then at runtime the binary may report that it cannot find the proper library.

The constraint with shared libraries is that the binary be able to find the shared libraries at run time. Exactly how this is done is architecture dependent, but in general the runtime linker looks for special environment variable that contains pathnames for directories to be searched. Under SunOS4.1.x and Solaris2.x, this environment variable is named `LD_LIBRARY_PATH`. Under HPUX, the variable is named `SHLIB_PATH`. A binary can also have a list of pathnames to be searched encoded inside it. Usually this is called the `RPATH`. In general, asking the user to set the `LD_LIBRARY_PATH` or `SHLIB_PATH` is frowned upon. It is better if the binary has the proper `RPATH` set at link time.

D.1.3 Differences between static and shared libraries: Unresolved symbols

A library consists of `.o` files archived together. A `.o` file inside a library might contain symbols (functions, variables etc.) that are not used by your program.

At link time, a static library can have unresolved symbols in it, as long as you don't need the unresolved symbols, and you don't need any symbol that is in a `.o` file that contains an unresolved symbol. However, with shared libraries, you must resolve all the symbols at link time, even if you don't necessarily use the unresolved symbol.

As an example, say you have a program that uses a symbol from the `pigi` library (`$PTOLEMY/lib.$PTARCH/libpigi.*`), but does not use Octtools which is used by other files that make up the `pigi` library

If you are linking with a static library, you can have some unresolved symbols in the static library, as long as you don't reference the unresolved symbols. So, in our example, you could just link with the static `libpigi.a`.

If you are linking with a shared `libpigi`, you must resolve all the unresolved symbols. So, if you need a symbol from the `libpigi` library, then you must also include references to the Octtools libraries that pigilib uses, even though you are not using Octtools. So you would have to link in `liboct.so` and `libport.so` and the other Octtools libraries.

One positive benefit of this is that *all* the symbols in pigilib are available at run time, which makes incremental linking much easier, especially if we have a shared g++ library.

D.1.4 Differences between static and shared libraries: Pulling in stars

If you are using static libraries, then for a symbol to be present in the binary, you must explicitly reference that symbol at link time. When building Ptolemy with static libraries, each star directory contains a *xxx*`stars.c` file (where *xxx* is the domain name, an example file is `$PTOLEMY/src/domains/sdf/stars/sdfstars.c`) which gets compiled into *xxx*`stars.o`. At link time, the *xxx*`stars.o` file is included in the link command and the linker searches `lib`*xxx*`stars.a` for the symbols defined in *xxx*`stars.o`, and pulls in the rest of the star definition.

If you are using shared libraries, then all the symbols in the `lib`*xxx*`stars` file are present at runtime, so you need not include the *xxx*`stars.o` file at link time.

D.2 Shared library problems

- Start up time of a binary that uses shared libraries is increased. We believe that some of the increased startup time comes from running star constructors. We are working on modifying Ptolemy so that startup time of binaries that use shared libraries is decreased. See "Startup Time" below for more information.
- The time necessary to start up a debugger is sometimes increased. When the debugger starts up, it usually has to load all the shared libraries so that the debugger knows where to find symbols.
- Building is more complex. Unfortunately, shared libraries are very architecture dependent. Also, the commands and command line arguments differ between architectures. Finally, how different versions of the same shared library are handled, along with the shared library naming conventions also vary between architectures.
- You need to keep track of where the shared libraries are, either by using `RPATH` at link time or setting `LD_LIBRARY_PATH` or `SHLIB_PATH`. The problem is that if you are building a C Code Generation (CGC) application that uses shared libraries, then at runtime the user needs to either have the necessary shared libraries in their `LD_LIBRARY_PATH` or `SHLIB_PATH`, or the binary needs to have the `RPATH` to the shared libraries encoded into it. This can be done with an option of the linker. The various commands to do this are architecture dependent, and `Default-CGC` target usually fails. The `Makefile_C` target and the `TclTk_Target` which is derived from `Makefile_C` is much more likely to work with shared libraries.
- It could be the case that binaries that use shared libraries might use slightly more memory.

D.2.1 Startup Time

In Ptolemy you can build a `pigiRpc` that has only the domains you are interested in with Jose Pino's `mkPtolemyTree` script in `$PTOLEMY/bin`, see the Programmer's Manual for more information. The startup time for a full `pigiRpc` is greater than for a `pigiRpc.ptrim` (SDF, DE, CGC and a few other small domains). If you use either `pigiRpc.ptrim` or `pigiRpc.ptiny` (SDF and DE only), then the start up time is quite reasonable. If you regularly use some of the other less common domains, then you can build special `pigiRpc` with just your domains.

One reason that startup time is increased might be because Ptolemy constructs a lot of objects and processes many lists of things like domains. We may be able to decrease startup time by carefully managing the star constructors.

One way to speed things up might be to create a large shared library that has the domains in which you are interested. Startup time might be faster if everything is in one file. Currently we have about 80 different shared libraries.

Combining these libraries into a few big libraries for ptiny, ptrim and pigi binaries might help. Of course, we could leave the 80 libraries and just add the new libraries. We have not tried this, but it might be interesting.

D.3 Reasons to use shared libraries

- All the symbols in a shared library are available at runtime. This is especially important with incremental linking of stars. If you have a shared `libg++`, then you can use all the symbols in `libg++` in a star for which you did not use the `libg++` symbol at link time. If you use static linking, then when you incrementally link, you only have symbols that you used when you linked the binary. The same is true for symbols in the Ptolemy kernel and the domain kernels.
- You don't need to write dummy functions to pull in code from a library. `$PTOLEMY/src/domains/sdf/stars/sdfstars.c` is an automatically generated C file that pulls in the stars from `libsdfstars.a`. If you use shared libraries, then you need not have a `sdfstars.c` file. Also, more than one person has been confused because they added new file containing new functionality to the Ptolemy kernel, and then when they tried to link in a star, the symbols they just wrote couldn't be found. Usually this is because they are not using the new symbols anywhere at link time, so the new symbols are not being pulled into the binary. If the Ptolemy kernel is a shared library, then this problem goes away, as the new symbols are present at incremental link time.
- Smaller binary size on disk. Shared library binaries are smaller on disk, so it is possible to have many versions of `pigiRpc` that include different domains, without using up a lot of disk space. If you use shared libraries, a `pigiRpc` is about 1.5Mb; if you use static libraries, then a `pigiRpc` is about 8Mb.
- Link time is greatly decreased with shared libraries. Under Solaris with shared libraries, it takes almost no time to link a binary. Under SunOS with static libraries it can take 8 minutes to link. Using a tool like Pure Inc.'s `purelink` can help, but `purelink` is expensive and is not available everywhere.
- If you are running multiple `pigis` on one machine, the memory usage should be reduced because of all the pigi binaries are sharing libraries. In theory, if a binary is built with static libraries, you should get some sharing of memory, but often the shared libraries result in better memory usage. If you use shared libraries for X11 and Tcl/Tk, then your memory usage should be lower.
- Using `dlopen()` to incrementally link in new stars is usually faster than the older method of using `ld -A`. Eventually, we may be able to link in entire domains at runtime using `dlopen()`.

D.4 Architectural Dependencies

In this section, we discuss shared library architectural dependencies

Table 1: Commands to use to find out information about a binary or library

Architecture	Command(s) that prints what libraries a binary needs	Library Path Environment Variable
`hppa`	`chatr` *file*	`SHLIB_PATH`
`irix5`	`elfdump -Dl` *file*	`LD_LIBRARY_PATH`

Architecture	Command(s) that prints what libraries a binary needs	Library Path Environment Variable
sol2	ldd *file* /usr/ccs/bin/dump -Lv *file*	LD_LIBRARY_PATH
sun4	ldd *file*	LD_LIBRARY_PATH

D.4.1 Solaris

Under Solaris the /usr/ccs/bin/dump -Lv *file* will tell you more shared library information about a *binary*. Under Solaris2, binaries compiled with shared libraries can have a path compiled that is used to search for shared libraries. This path is called the RPATH. The ld option -R is used to set this at compile time. Use /usr/ccs/bin/dump -Lv *binary* to view the RPATH for *binary*. The RPATH for a library can be set at the time of creation with the -L flag:

```
g++ -shared -L/users/ptolemy/lib.$PTARCH -o librx.so *.o
```

or by passing the -R flag to the linker:

```
g++ -shared -Wl,-R,/users/ptolemy/lib.$PTARCH -o librx.so *.o
```

Constructors and Destructors between SunOS4.x and Solaris2

The Solaris2 SPARCompiler c++4.0 Answerbook says

> On SunOS 5.x, all static constructors and destructors are called from the .init and .fini sections respectively. All static constructors in a shared library linked to an application will be called before main() is executed. This behavior is slightly different from that on SunOS4.x where only the static constructors from library modules used by the application are called.

D.4.2 SunOS

The SunOS4.x port of Ptolemy uses BSD ld style linking, which will **not** work with a binary that is linked with **any** shared libraries. For incremental linking of stars to work, the ldd command must return statically linked when run on a SunOS4.x pigiRpc or ptcl binary.

```
ptolemy@mho 2% ldd ~ptolemy/bin.sun4/pigiRpc
/users/ptolemy/bin.sun4/pigiRpc: statically linked
```

D.4.3 HPUX

Under HPUX, shared libraries must be executable or they will not work. Also, for performance reasons, it is best if the shared libraries are not writable.

Under HPUX, shared libraries have a .sl suffix, and HPUX uses the SHLIB_PATH environment variable to search for libraries.

Under HPUX10, when you are building shared objects, you need to specify both -fPIC and -shared. (-fpic -shared will also work). The reason is that the temporary files that are generated by g++'s collect program need to be compiled with -fPIC or -fpic. Other platforms don't need both arguments present.

D.5 GateKeeper Error

If there are problems with shared libraries, then you may see

```
ERROR: GateKeeper error!
```

message when you exit `pigi`.

`GateKeepers` are objects that are used to ensure atomic operations within the Ptolemy kernel. The Ptolemy error handling routines use `GateKeepers` to ensure that the error messages are not garbled by two errors trying to write to the screen at once.

Say we have two files that make up two different libraries, and both contain the line:

```
KeptGate gate;
```

With static libraries, the linker will resolve gate to one address and call the constructor once. The destructor will also be called once.

With shared libraries, there are two instances of this variable, so the constructor and the destructor get called twice.

The problem is that there are several different implementations of the error routines depending on if we are running under `pigi`, `ptcl` or `tycho`. The static function `Error::error` is defined in several places, and which definition we get depends on the order of the libraries. (See `$PTOLEMY/src/kernel/Error.[cc,h]`, `$PTOLEMY/src/pigilib/XError.cc`, `$PTOLEMY/src/ptcl/ptclError.cc` and `$PTOLEMY/src/tycho/tysh/TyError.cc`). Each implementation defines a static instance `gate` of `KeptGate`.

You will see the `ERROR: GateKeeper error!` message when you exit if there is more than one `KeptGate gate`, and the destructor is called twice for `gate`.

Under HPUX10.x, the error message is produced if `libptolemy` is static and the other lib (`libpigi`, `libptcl`, `libtysh`) is shared. Here the work around is to make the other library static too.

The way to debug `GateKeeper Error!` problems is to set breakpoints in `Error::error`, and then trigger an error and make sure that the right error routine in the right file is being called. One quick way to trigger an error is to set the current domain to a non-existent domain. Try typing `domain foo` into a `pigi -console`, `ptcl` or `tycho -ptiny` prompt.