Chapter 8. The incremental linker

Authors: *Joseph T. Buck*
Christopher Hylands

The incremental linker permits user written code to be added to the system at runtime. Two different mechanisms are provided, called a temporary link and a permanent link. With either a temporary link or a permanent link, code is linked using the incremental linking facilities of the Unix linker, the new code is read into the Ptolemy executable, and symbols corresponding to C++ global constructors are located and called. This means that such code is expected to register objects on Ptolemy's known lists (e.g. KnownBlock, KnownState, or KnownTarget) so that new classes become usable. *Warning:* if the executable containing the Linker class is stripped, the incremental linker will not work!

8.1 ld -A style linking vs. dlopen() style linking

There are two ways incremental linking is implemented: "ld -A" and "dlopen()" style linking.

The first type of implementation uses a BSD Sun-style loader with the -A flag to load in .o files. Usually, binaries that are to be dynamically linked must be built with the -N option. This is the older style of linking present in Ptolemy0.5 and earlier.

The second type of implementation uses the System V Release 4 `dlopen()` call to load in shared objects (.so files). SunOS4.1.x, Solaris2.x and Irix5.x support this style of dynamic linking. In Ptolemy0.6, only the sol2, sol2.cfront, and hppa architectures support dynamic linking of shared objects.

The interface to both styles of linking is very similar, though there are differences.

8.2 Temporary vs. Permanent Incremental Linking

Code that is linked in by the "temporary link" technique does not alter the symbol table in use. For that reason, subsequent incremental links, whether temporary or permanent, cannot "see" any code that was linked in by previous temporary links. The advantage is that the same symbols (for example, a Ptolemy star definition) may be redefined, which is useful in code development, as buggy star definitions can be replaced by valid ones without exiting Ptolemy. Code that is linked in by the "permanent link" method has the same status as code that was linked into the original executable. With "ld -A" style incremental linking, a permanent link creates or replaces the `.pt_symtable` file in the directory in which Ptolemy was started. This file contains the current symbol table for use by subsequent links, temporary or permanent. This file is deleted when the Ptolemy process exits normally. It is left around when the process crashes, as it is useful for debugging (as it contains symbols for object files that were incrementally linked using the permanent method as well as those in the original executable). With `dlopen()` style incremental linking, we keep track of all the files that have been permanently linked in. After a file has been permanently linked in, each successive link (permanent or not) includes all the permanently linking in files. That is, if we permanently link in foo.o, then when we link in bar.o, we will generate a shared object file that includes both foo.o and bar.o, and then call `dlopen()` on that shared object file. Note that with `dlopen()` style linking it is possible to relink in stars that have been permanently linked. When a file is to be linked in, we check against the list of permanently linked in file names and remove any duplicates. This method of checking will fail if one permanently links in a file, and then links in the same file as a different name, perhaps through the use of symbolic links. That is, if we permanently link in `./foo.o` and `./bar.o` is a link to `./foo.o,` when we link in `./bar.o,` we will have multiple symbols defined, and we will get an error. In other words, we only check for duplicate file names, we do not check for duplicate symbols in any files, the loader does this for us. Currently each `dlopen()` style link generates a temporary shared object in `/tmp.` If you are doing a large number of `dlopen()` style permanent links, you will have many files in `/tmp.` We hope to resolve this potential problem in a later release. Eventually, it would be nice if the code read the value of an optional environment variable, such as `TMPDIR.`

8.3 Linker public members

`static void init(const char*` *`execName`*);

This function initializes the linker module by telling it where the executable for this program is. For most purposes, passing it the value of `argv[0]` passed to the `main` function will suffice.

`static int linkObj(const char*` *`objName`*);

Link in a single object module using the temporary link mechanism (this entry point is provided for backward compatibility).

`static int multiLink(const char*` *`args`*, int *`permanent`*);
`static int multiLink(int` *`argc`*, char** *`argv`*);

Both of these functions give access to the main function for doing an incremental link. They permit either a temporary or a permanent link of multiple files; flags to the Unix linker such as `-l` to specify a library or `-L` to specify a search directory for libraries are permitted. For the first form, *`args`* are passed as part of a linker command that is expanded by the Unix shell. A permanent link is performed if *`permanent`* is true (non-zero); otherwise a temporary link is performed. The second form is provided for ease of interfacing to the Tcl interpreter, which likes to pass arguments to commands in this style. In this case, *`argv[0]`* indicates the type of link: if it begins with the character `p,` a permanent link is performed; otherwise a temporary link is performed. The remaining arguments are concatenated (separated by spaces) and appear in the argument to the Unix linker.

`static int isActive();`

This function returns TRUE if the linker is currently active (so objects can be marked as dynamically linked by the known list classes). Actually the flag it returns is set while constructors or other functions that have just been linked are being run.

`static int enabled();`

Returns true if the linker is enabled (it is enabled by calling `Linker::init` if that function returns successfully). On platforms that do not support dynamic linking, this function always returns false (zero).

`static const char* imageFileName();`

Return the fully-expanded name of the executable image file (set by `Linker::init).`

`static void setDefaultOpts(const char*` *`newValue`*`);`
`static const char* defaultOpts();`

These functions set or return the linker's default options, a set of flags appended to the end of the command line by all links.

8.4 Linker implementation

For each port of Ptolemy to a particular release, the Linker is implemented in one of two styles: "ld -A" style or "dlopen()" style. We discuss each style below.

8.4.1 Shared Objects and dlopen() style linking

If a Ptolemy release on a platform supports `dlopen` style dynamic linking, then the ptcl `link` command can be called with either a `.o` file or a `.so` file. If the `link` ptcl command is passed a `.o` file, then a `.so` file will be generated. If link ptcl command is passed a `.so` file, then the `.so` file will be loaded. If the `.so` file does not exist, then an error message will be produced and the link will return. There are several ways to specify the path to a shared object.

1. Using just a file name `link foo.so` will not work unless LD_LIBRARY_PATH includes the directory where `foo.so` resides. The man pages for `dlopen()` and `ld` discuss LD_LIBRARY_PATH Interestingly, using `putenv()` to set LD_LIBRARY_PATH from within ptcl has no effect on the runtime loader.
2. 2If the file name begins with `./`, then the current directory is searched. `link ./foo.so` should work, as will `link ./mydir/foo.so`.
3. If the file name is an absolute path name, then the shared object will be loaded. `link /tmp/foo.so` should work.
4. Dynamic programs can have a run path specified at link time. The run path is the path searched at runtime for shared object. (Under Solaris2.3, the `-R` option to `ld` controls the run path. Under Irix5.2, the `-rpath` option to `ld` controls the run path). If ptcl or pigiRpc has been compiled with a run path built in, and the shared object is in that path, then the shared object will be found. The Sun Linker Manual says: "To locate the shared object foo.so.1, the runtime linker will use any LD_LIBRARY_PATH definition presently in effect, followed by any runpath specified during the link-edit of prog and finally, the default location /usr/lib. If the file name had been specified ./foo.so.1, then the runtime linker would have searched for the file only in the present working directory."

8.4.2 Porting the Dynamic Linking capability

This section is intended to assist those that attempt to port the Linker module to other platforms. The Linker class is implemented in three files: `Linker.h,` specifying the class interface, `Linker.cc,` specifying the implementation, and `Linker.sysdep.h,` specifying all the machine dependent parts of the implementation. To turn on debugging, compile `Linker.cc` with the DEBUG flag defined. One way to do this would be:

```
cd $PTOLEMY/obj.$PTARCH/kernel; rm -f Linker.o; make OPTIMIZER=-DDEBUG
```

The Linker class currently uses "ld -A" style dynamic linking on the Sun4 (Sparc) running SunOS4.1 and `g++`, the Sun4 (Sparc) running SunOS4.1 and Sun's `cfront` port, DECStations running Ultrix, HP-PA running `g++` or HP's `cfront` port. The Linker class currently uses "dlopen()" style dynamic linking on the Sun4 (Sparc) running Solaris2.4 and `g++`, the Sun4 (Sparc) running Solaris2 and Sun's `cfront` port (Sun `CC-3.0`), the Sun4 (Sparc) running Solaris2 and Sun's native C++ compiler `CC-4.0`, and SGI Indigos running IRIX-5.2 and `g++`. The intent is to structure the code in such a way that no `#ifdefs` appear in `Linker.cc`; they should all be in `Linker.sysdep.h`.

8.4.3 ld -A Style Dynamic Linking

The linker reads all new code into a pre-existing large array, rather than creating blocks of the right size with `new`, because the right size is not known in advance but a starting location must, as a rule, be passed to the loader in advance. This means that there is a wired-in limit to how much code can be linked in. The symbol `LINK_MEMORY`, which is set to one megabyte by default, is easily changed if required. Here are the steps taken by the linker to do its work:

1. Align the memory as required.
2. Form the command line and execute the Unix linker. Only certain flags in the command line will be system-dependent.
3. Read in the object file. This is heavily system-dependent.
4. Make the read-in text executable. On most systems this is a do-nothing step. On some platforms (such as HP) it is necessary to flush the instruction cache and that would be done at this point.
5. Invoke constructors in the newly read in code. Constructors are found by use of the `nm` program; the output is parsed to search for constructor symbols, whose form depends on the compiler used.
6. If this is a permanent link, copy the linker output to file `.pt_symtable`; otherwise delete it.

8.4.4 dlopen() Style Dynamic Linking

Here's how we link in an object using `dlopen()` style linking.

1. Generate a list of files to be linked in. If we have not yet done a permanent link, then the list of files to be linked in will consist of only the files in this link or multilink command. If the link is a permanent link, then we save the object name. For each successive link, we check the name of the object to be linked in against the list of objects permanently linked for duplicate file names. For each link after a permanent link, we include the names of all the unique permanently linked in objects in the generation of a temporary shared object file.

2. Generate a shared object `.so` file from all the objects to be linked in. The `.so` file is created in /tmp.
3. Do a `dlopen()` on the shared object.
4. Most architectures use `nm` to search for constructors, which are then invoked. Currently, sol2.cfront does not need to search for, or invoke constructors. gcc-2.5.8 has patches that allow similar functionality, but apparently these patches are not in gcc-2.6.0. Shared libraries in the SVR4 implementation contain optional `__init` and `__fini` functions, called when the library is first connected to (at startup or `dlopen()`) and when the library is disconnected from (at `dlclose()` or program exit), respectively. Some C++ implementations can arrange for these `__init` and `__fini` functions to contain calls to all the global constructors or destructors. On platforms where this happens, such as sol2.cfront, there is no need for the Linker class to explicitly call the constructors, as this will happen automatically.