Dynamic linker

Overview

The RPT group has divided RTEMS and RPT support code into a number of "dynamically shared objects" which are structurally the same as the shared libraries one uses under Linux [ELF]. There are two kinds of linking, or binding pieces of compiled code into a single functioning whole. The first kind of linking, static linking, creates the shared objects from the ".o" files produced by the compiler. The second kind of linking, dynamic linking, loads shared objects into memory on demand while the program is already running. As the name implies a single copy of a shared object's code and data may be used by a number of other shared objects which reduces the use of memory. There is also a potential reduction gained from loading only those shared objects that contain what's actually needed.

The system boot loader loads and starts the principle shared object containing RTEMS and the RPT dynamic linker. Then the application initialization code written by the RPT group uses the dynamic linker to load the remaining shared objects specified by the system configuration, itself contained in several shared objects.

Using the dynamic linker

The dynamic linker is available for use in applications as well as in system start-up.

From the RTEMS shell

The shell provides the "run" command in order to let you run a special kind of shared object called a Task. What makes a Task special is the way its code is organized and the name of the entry point. Most of the organization is taken care of by including a special Task-stub library when you create the shared object.

The "run" command allows you to specify the Task name (up to four characters), scheduling priority and the size of the stack in bytes [run].

From C or C++ code

The C function lnk_load() is the main entry point of the dynamic linker [linkerCall]. You give it the name of the shared object you want to load. The linker will attempt to find it and any other shared objects it needs according to the rules laid out in the next section.

SO names, dependencies, namespaces and installation

Whether you call the linker from the shell or from your own code it obeys the same rules:

- The shared object whose name you give as an argument is loaded but not "installed" (see below).
- If that shared object depends upon others then such of those others that are not already loaded are loaded.
- Shared objects that are loaded as dependencies are installed if they request it and if they are not already installed.
- A shared object isn't allowed to list itself as a dependency.

Each shared object has a name, the so-called SO name or soname, used as follows:

- Each shared object has its own soname built into it.
- Each shared object refers to the other shared objects that it depends on by their sonames.
- The system tracks installed shared objects by their sonames.

An installed shared object is a sort of public library. It's loaded only once; if it's named as a dependency the dynamic linker won't bother loading it again. The prototypical example is system:rtems.so which is needed by practically every other shared object. This way we don't have to load multiple copies of rtems.so.

Notice that the soname "system:rtems.so" has two components separated by a colon. The first component, "system" gives the namespace to which the shared object belongs. Objects in the same namespace are found in the same place. Where that place is is determined by the system's namespace table; each namespace name is associated with a directory in the RTEMS filesystem. The second component of the soname, "rtems.so" is the filename of the shared object after all directories have been removed. If the system namespace table associates "/mnt/rtems" with the namespace "system" then "system: rtems.so" will be looked for at /mnt/rtems/rtems.so when the time comes to load it.

Other shared objects besides system:rtems.so may be installed. In general, a shared object that the dynamic linker loads in order to satisfy a dependency it will also install if the shared object requests it. In such a case that single loaded copy is used whenever named as a dependency not only in that run of the linker but in all subsequent runs. In any given run of the linker it will load any object only once, but those objects that are not installed will be re-loaded if need be in subsequent runs, ignoring any copies loaded in previous runs. In this case shared objects are really unshared.

Resolving symbolic references

Though each shared object contains a list of the sonames of all the other shared objects it depends on, the so-called needed-list, this doesn't tell the dynamic linker just what the object requires from the other objects. It could be user data, functions, classes, etc. Each of these items has a name which appears in a symbol table built into the shared object that defines them. The object that needs to use them also has the names in its table, though marked as "undefined", that is to say "not defined in this shared object". There's no indication of which shared object is the definer. Instead, for each undefined symbol in a newly-loaded object the dynamic linker searches for a definition in each of the objects on the first object's needed-list [scope], taking the first definition that it finds. It doesn't check for duplication. The linker then puts the address of the definition it found into all the places in the first shared object that require it.

Initialization of newly loaded shared objects

The following initializations are performed in the order listed here. They're only performed once when the object is loaded.

Uninitialized variables

A shared object may define statically allocated variables that are given no explicit initial values in source code. Such variables take up no space in the shared object file; there's only a count of how much space they need. Dynamic linking has to allocate space for these variables and, in accordance with the C and C++ standards, initialize that space to all zeros.

The .init section and C++ static constructors

A shared object is divided into many named "sections" some of which have special meaning for the dynamic linker. One of these is named ".init" and contains pointers to functions that must be called before the shared object can be considered usable. Normally the .init section is filled by the compiler and contains pointers to functions that run static C++ constructors. With the "section" attribute [attr] you can place pointers of your own in the .init section.

Installation

If the shared object was loaded as a dependency, contains a global variable named lnk_options and that variable's (integer) value has the bit LNK_INSTALL set then the shared object's soname and location are recorded in the table of installed objects.

The prelude and preferences functions

Shared objects to be run on a RPT system may have two optional initialization functions which if present are called by the dynamic linker. The first, lnk_preferences() , returns a 32-bit preference datum which may be an int or a pointer and is passed as an argument to lnk_prelude(). The latter function is a general initialization function designed to be more easily used than the .init section. Since most other initialization for the shared object has been done, lnk_prelude() can do most anything: start tasks, load other shared objects, print messages, etc.

Memory access permissions

A shared object file's loadable content is divided into a small number of "segments". Each segment has a set of permission flags: (R)eadable, (W)riteable and e(X)ecutable. In shared objects built for RPT systems there's normally one RX segment containing instructions and read-only data and one RW section containing non-constant data. The dynamic linker uses the CPU's memory management unit (MMU) to set the access type of the memory allocated to each segment to match the segment's permission flags.

Linker actions step-by-step example

Suppose you use the dynamic linker to load a shared object A, which makes use of objects B and D. B uses object C which in turn uses E. D and E use no other objects. Then the needed-lists in the files containing these objects may look like this (the ordering depends on the order in which the shared objects are mentioned on the command lines used with the static linker):

A: B C D E

B: C E

C: E

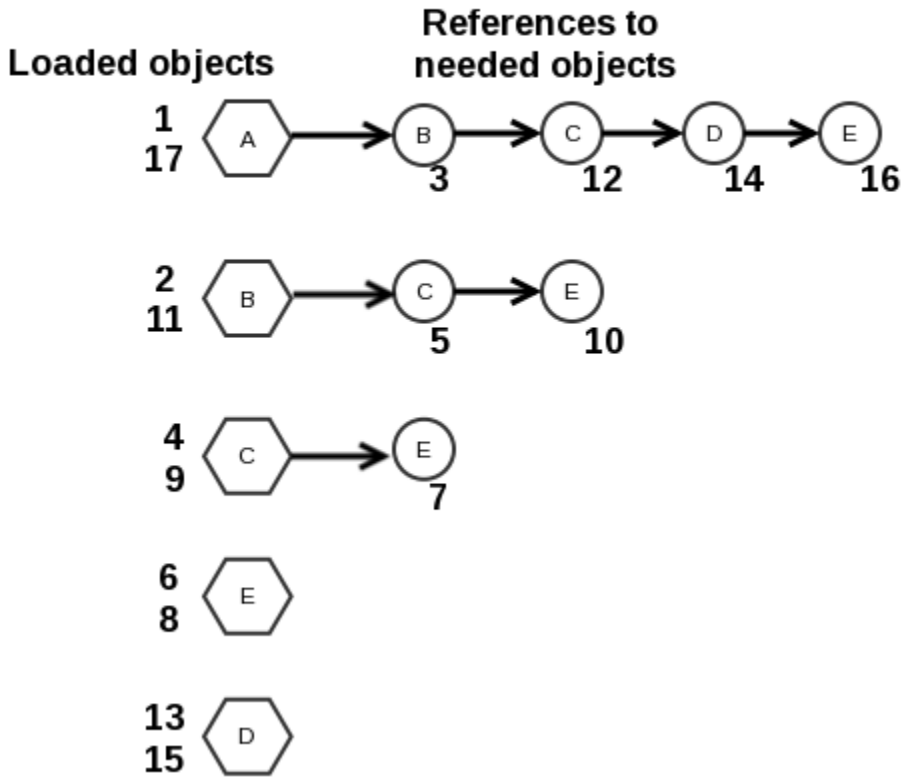
D:

E:

All the objects except for E request installation. If you ask the dynamic linker to load A it will perform the following actions:

1. Load A, create A-node.
2. Load B, create B-node.
3. Add a needs-B node for A.
4. Load C, create C-node.
5. Add a needs-C node for B.
6. Load E, create E-node.
7. Add a needs-E node for C.
8. Bind E, scope is E.
9. Bind C, scope is C, E. Install C.
10. Add a needs-E node for B (E is already loaded).
11. Bind B, scope is B, C, E. Install B.
12. Add a needs-C node for A.
13. Load D, create D-node.
14. Add a needs-D node for A.
15. Bind D, scope is D. Install D.
16. Add a needs-E node for A.
17. Bind A, scope is A, B, C, D, E.

The following diagram represents the state of the linker's internal data structures, with the nodes numbered according to the steps above.



While scanning needed-lists the linker employs a top-down approach; as soon as it sees that an object is needed it loads the object (if needed) then starts scanning the needed-list of the new object. Once the needed-list for a shared object has been completely scanned, the linker then performs the binding for that object.

To bind an object:

1. Perform local relocations.
2. Search the objects in this object's scope, in order, for symbols not defined in this object.
3. Run the .init code for this object.
4. Read lnk_options, if this object has one.
5. Call lnk_preferences() if this object has one.
6. Call lnk_prelude() if this object has one.
7. Set the memory access rights for this object's loaded segments.

Note that object A, because it's the object whose name is passed to the linker as an argument, is not installed.

Notes

[ELF] The file format used is the standard Executable and Linkable Format. The ELF standard defines three kinds of objects: compiler output (not directly executable), executable main programs and executable dynamic shared objects. RPT systems allow the loading and execution only of files in the third format.

[scope] For any given shared object the set of objects searched to satisfy its undefined references is called its scope. For RPT code the scope is the original shared object followed by the objects named in its needed-list. It may seem strange to search an object for its own undefined symbols but it's necessary to handle some unusual cases. This scope rule is much simpler than the one employed by a Linux dynamic linker.

[attr] Attributes are a language extension offered by GCC.

[run] Shared object files containing Task code are given the filename extension ".exe". Argument strings may be passed to a task using an interface resembling the argc/argv interface for C main programs. Here's a run of our standard example task which prints its arguments and then suspends itself.

```
[/] # run -N foo -P 200 -S 10240 system:hello.exe -- doe re mi
2014/06/04 18:20:12.081941: hello_task was called with 4 arguments @ AA0D25C: system:hello.exe doe re mi
```

```
[/] # stop foo
2014/06/04 18:20:59.750071: ending hello_task

[/] # help run

run          - Usage:  run [[-N taskName] [-P priority] [-S stacksize]] <namespa
            ce>:<exe> [-- task arguments]
              Execute the code in file pointed to by namespace:exe
              with RTEMS task name as <taskName> (4 characters)
              Optional arguments:
              -N taskName:     RTEMS task name
              -P <priority>:   RTEMS priority.
              -S <stacksize>:  RTEMS stack size.
```

[linkerCall] In this example code we load and execute a task in much the same way that the "run" shell command does, though with much simpler error handling: dbg_bugcheck() prints a message and then calls the RTEMS fatal error handler.

```
#include <inttypes.h>
#include <rtems.h>

#include "debug/print.h"
#include "elf/linker.h"
#include "task/Task.h"

void launch(void) {
  uint32_t status = STS_K_SUCCESS;
  Task_Attributes myAttr;
  myAttr.name = rtems_build_name('M', 'Y', 'T', 'K');
  myAttr.stack_size = 40 * 1024;
  myAttr.priority = 200;
  myAttr.attributes = RTEMS_DEFAULT_ATTRIBUTES;
  myAttr.modes = RTEMS_NO_PREEMPT;
  myAttr.image = NULL;
  myAttr.argc = 0;
  myAttr.argv = NULL;
  Ldr_elf* mytask = lnk_load("myspace:mytask.exe", NULL, &status, NULL);
  if (!mytask) {dbg_bugcheck("mytask.exe did not load. DIE!\n");}
  rtems_id id;
  Task_status tstat = Task_Run(mytask, &myAttr, myAttr.argc, myAttr.argv, &id);
  if (tstat) {dbg_bugcheck("mytask.exe did not run. DIE!\n");}
  // At this point you'll have launched a new thread running your task code.
}
```