Runtime Loader-Linker Technologies

by Pierre-Alain Darlet
Wind River Systems

ESC April 2001
Class #512

Abstract

Modern operating systems allow user applications to share code and dynamically add new extensions to themselves. Many operating systems allow for extending the kernel capabilities as well. It is the responsibility of the loader-linker sub-system to perform these operations. In this document we will describe the advantages of using the ELF format in this matter, and present alternate techniques for dynamic linking in the context of embedded systems. We will then describe how this can apply to dynamically extend the kernel of an operating system, and present a runtime software component replacement technique.

1. Introduction

Runtime loaders and linkers are essential parts of an operating system to load and run applications. They are responsible for locating the application code in secondary storage and installing it in the system's memory space. The application's code may require some adjustments in order to adapt to the location where it has been installed, as well as be able to make use of code external to the application.

Different programs within the operating system can handle these roles. However, for simplicity we will consider in this paper that a unique program called runtime loader-linker does this. Also, the link editor is a well-known part of any compiler tool chain. Its role is very comparable, albeit sometimes more complex, to that of the runtime loader-linker. In this paper we focus on the runtime aspect of the loading and linking, acknowledging that the binary files involved in this process are generated by the tool chain's link editor. A much more complete discussion of the relationship between link editors and runtime loader-linkers can be found in [1].

2. Runtime Loading and Linking

2.1. Principle

In order to execute in the memory of the system, the application has to be loaded, generally from a disc, but possibly from any kind of storage. In the case of the embedded systems, for which using a disc drive is often not convenient, the usage of flash memory cards is sometimes preferred to hold the application files. Network access to the file system of a host machine, when available, is also a good alternative. Aside from the obvious device management involved in the retrieval of the application from the secondary storage, the load operation involves the memory management sub-system since the runtime loader-linker transfers the application from a binary format file to the random access memory (RAM).

Typically, an application is made of executable instructions, also known as text, of initial values to be used in variables, commonly referred to as data, of values for constants, called read-only data, as well as an empty area prepared for the automatic, as yet uninitialized, variables, also called bss. Areas, or pages, of memory need to be allocated to fit the application's code and data. If the operating system makes use of the memory management unit (MMU), the appropriate memory protection attributes have to be set (typically `Read' and `Execute' for the code, `Read' and `Write' for the initialized and uninitialized data, and only `Read' for the data which must not be modified).

Once in the system's memory the application may require additional steps to be usable, namely changes in the code, depending on whether or not the tool chain has already prepared the application's code for its final execution address. This process, called relocation, applies when the application's internal references are absolute (i.e. refers to an address, instead of an offset). For instance when a global variable is manipulated in a routine, the variable's content is generally accessed via its address, not via an offset from the instruction that does the manipulation.

On some systems, such as Unix, the application may not need any relocation because the memory allocated for the application always starts at a known address. The MMU enables such operating systems to have many applications loaded, each in its own address space, but all starting at the same virtual address. So the actual execution address, in virtual memory, can be the same as the one used by default by the tool chain. In other systems, in particular those running without using the processor's MMU, the area of memory allocated for the application will be different each time and for each application, making it compulsory to adapt the application's code to a different base address. Interesting comparisons between some operating systems can be found in [1].

A special case, which we will not consider further in this document, is the loading of the operating system itself. In most cases, the operating system is also stored in a binary file, but the major difference is that the virtual memory context does not exist yet (the hardware is not yet fully initialized). Then the operating system is generally prepared to run at a given start address. The runtime loader involved at this phase is generally a simple program, which just knows how to read the system file and copies the system's code and data where it should be in physical memory. This simple loader is sometimes called the boot loader.

2.2. Static Linking and Fully-Linked Files

It is very uncommon that "real life" applications are held in only one source file. In general, dozens if not hundreds of files are involved - each providing the interface for a facet of the application. All these files need then to be combined, or linked, together to generate the executable file. The link editor generally does this during the build process. Since this binding can not be changed once made the term "static linking" is traditionally used.

The binary files holding code and data in their intermediate form, said to be relocatable, are known as object modules (which are normally named with a .o or .obj file name extension) generated by the compiler tool chain. These files hold all the information necessary to link them with other files (object modules or libraries) and generate the executable file.

This link stage generally consists of grouping together the text, data and bss of all the unit modules then writing the result to an output file. This phase often results in small changes, registered in the output file, from the original content of the unit modules. These changes are the result of two operations called relocation and symbol resolution, described later in this document (see also [1]).

It is very uncommon that the application's code is self-sufficient i.e. it does not need any code already written and stored in the form of libraries. In order for the application to take advantage of existing libraries of code, the link editor can be used to statically link these libraries with the application. This produces a single fully-linked executable file that can in turn be loaded in the system.

2.3. Impact on the System's Resources

Static linking has the obvious advantage of generating self-sufficient applications, which are not vulnerable to API or implementation changes in the libraries, once they have been built. The compiler tool chain technologies also ensure that the application code is tuned to its best since because the addresses are determined it can perform, for instance, address-specific optimizations such as replacing 32-bit absolute branches with 16-bit offset PC-relative branches. However, this also has the obvious disadvantage of inflating the size of the application significantly because a copy of the required library's modules is included in the statically linked file. The resulting binary file takes more room on disc and the application itself takes more of the system's RAM. This is especially annoying when the libraries, such as the C library or the standard I/O library, are used by virtually all applications. Also, building the application, i.e. generating the final executable file, can take a non-negligible amount of time and resources, especially when several big libraries are involved.

Traditional operating systems take advantage of the MMU when loading applications, by creating separate address spaces, in virtual memory, for each of them. Instead of allocating pages of physical memory for many instances of the same application, they can simply map the existing physical pages into the virtual memory space of the applications. This way, the instructions and the read-only data of an application can be shared between all the instances of this application. Only the modifiable data are actually allocated in new physical pages, possibly based on "copy on write" techniques (as an example, see [2] for the history of the virtual memory implementation in Sun's SunOS, for instance). However, sharing can not practically apply when non identical applications all use the same set of libraries because it is improbable that memory pages would have the same virtual address in processes running such fully-linked applications.

Operating systems based on a flat memory model, sometimes also called single address space operating systems (see [3] for a description of one of these systems) are even less able to handle duplicate code when stored in single executable files. Even if their runtime loader-linker can perform the required relocations to install all these applications in memory, it has no choice but to copy the entire code and data for each of the applications, regardless of whether the system is benefiting from the protection features of a MMU.

An alternative to these problems is based on loading, and linking, applications and libraries dynamically, as described later in this document.

3. The Executable and Linking Format (ELF)

There are many binary file formats. John Levine, in [1], gives a good retrospective of the more successful ones. In this document we will concentrate on the Executable and Linking Format (ELF) which, according to [6], originated in the UNIX System Laboratories. Not only is the ELF format the de-facto industry standard for the conventional Unix operating system, but this format is also remarkably flexible such that it is easy to adapt it to many operating system-specific needs while retaining compatibility with the ELF format definition.

3.1. Layout of an ELF file

This section provides an overview of the ELF format. A full description can be found in [6], however the reference document does not include the relocation types and some aspects of the program loading and dynamic linking. This information is processor architecture-dependent and can be found in separate documents for each processor family, such as [7].

ELF defines three main types of files:

1. Relocatable files - holding code and data intended to be linked with other object files.
2. Executable files - holding the program's code and data in a form suitable for execution.
3. Shared object files - intended to be used by the runtime linker.

A subtlety of ELF is that it defines two ways of looking at a file: the linking view and the execution view. The execution view applies to programs ready to execute and allows for an efficient load of the application. The linking view applies to the relocatable and shared object files and offers all the information necessary to handle the content of the file properly. These two views can be combined, although the elements relevant to one view may be optional in the other view.

An ELF file layout can be characterized as follows:

- ELF file header: gives information about what the file is and where the principal elements are in the file.
- section headers: this is a table of structures, each describing one of the sections. This is part of the linking view.
- sections: each section holds the actual information related to the program.
- program headers: this is a table of structures, each describing a set of sections logically connected. Such a set is called a segment. This is part of the execution view.

The sections hold not only the text and data of the program, but also the symbolic information (symbol table and name string tables), the relocation information, eventual debugging information, and more information necessary to properly link and execute the program. Some of these sections, such as the Global Offset Table and the Procedure Linkage Table, will be described further in this document. ELF differentiates the allocable sections, which use memory in the system, from the informational sections that are used by the linker when handling the allocable section contents.

A segment is made of one or more sections. If several sections are included in a segment, they have to be contiguous in the file and in the system's address space. Since only one program header describes this set of sections, they are handled as a unit by the runtime loader-linker and have to share the same memory attributes. The program header table makes sense only when the content of the file is ready for execution. Thus, it is optional in a relocatable file.

3.2. Loading an Executable ELF File

The ELF program header holds all the information necessary to copy any of the segments from file to memory:

- segment's location in the file (offset from the start of file).
- size of the segment both in the file and in memory.
- installation address in memory (virtual and/or physical).
- memory protection associated with the segment.

Loading an executable ELF file is then a simple matter: the runtime loader-linker just has to read the file's program header table to know the offset and size of each loadable segment (the type PT_LOAD characterizes a loadable ELF segment) in the file. The bss segment uses no room in the file but must be allocated in memory and zeroed out as expected by the programming environment. How these segments are allocated in the system memory actually depends on the operating system.

In systems such as Unix, the segments are arranged so that the segments can be loaded in one shot in physical memory. The mapping in virtual memory occurs in a second stage. Such an organization makes reading the file easier, and makes use of additional features such as sharing page content and swapping to disc easier (see [1] and [2] for a discussion on these aspects). The file and program headers are loaded at the beginning of the first text page, although they are not used by the application itself. Since it holds elements requiring different access rights, the physical page at the boundary between the text and the data segments is being mapped twice. It is mapped once in a page from the set of virtual pages attributed to the text segment (read and execute) and once in a page from the set of virtual pages attributed to the data segment (read and write). Of course, whenever a modification is performed on the data, the copy-on-write mechanism allocates a new physical page, copies the content of the original page, and re-maps the physical page in virtual space while committing the change.

On embedded systems where the usage of RAM is more critical, where discs are not often available and even a MMU is sometimes unavailable, the segment allocation strategy can be entirely different and only the segment's content is copied to memory. When the processors supported by these operating systems do not have a MMU, or their MMU is not used, the system may only allocate the physical memory for each of the segments, before copying them from file to memory. Sharing code in such systems is not easy, although it is possible as explained in the next section. When the MMU is used, it can be applied to provide the protection the segments need, after they are copied to memory. It can also be used for physical to virtual memory mappings, if this makes sense for the operating system.

3.3. Loading object modules instead of executable files

Loading object modules instead of executable files has several advantages for embedded systems. Since these systems can run on a wide variety of hardware configurations, it is more difficult to have pre-set start addresses for the applications. In addition, the possible absence of an MMU makes the usage of executable binary files potentially difficult. Indeed, they would have to be linked at different, pre-determined, addresses to avoid collision (see also [3] and [5] for a discussion of the advantages and limitations of linking in a single address space operating system). Some architectures (for instance Intel's i86) supports hardware segments which can then be used to handle applications stored as executable files, but this is common among processors.

Also, the limitation on the amount of memory is an important constraint in the embedded world. Various types of memory (DRAM, SRAM, Flash) can also be arranged in – possibly - intricate ways. Then, having more control over what is being installed in the system's memory is important. Although it is possible to achieve this control through the link editor, it is then up to the application developer to manage the location of each application in memory, making the evolution of the installed application difficult (often resulting in the use more memory…). Loading and unloading object modules is then an operation of finer granularity than loading complete executable files.

However, loading relocatable modules makes the notion of an application less clear. It is up to the user to know which object modules are parts of each application. Although this approach is not convenient for user applications executing on traditional systems, it is pertinent when the kernel needs to be extended (see [8] for metrics applying to kernel extension). In this situation, logical containers (separate address space or otherwise) are not applicable since the added code needs to be executed at kernel level, not at user/application level. Also, kernel extension technologies must not degrade the performance of the kernel. The runtime linking between the added code and the kernel code ensures the same performance as the static linking performed by the link editor. Although this technique does not bring a solution to the security issues involved when upgrading parts of a kernel (see [8] for a discussion on this subject), it allows for upgrades without stopping the system itself.

Directly loading object modules creates much more burden on the runtime loader-linker since it then has to do about the same work as the link editor does when building an executable file. Both the relocations and all the symbol resolutions that would be performed by the link editor need to be done by the runtime loader-linker as well. Also, the management of the memory allocated for the sections becomes more complex. The next section of this document describes this process.

3.4. Loading Relocatable ELF Files

Loading an object module in order to get an executable application implies the following steps:

- reading the section headers of the object modules
- allocating memory for the loadable sections
- copying the loadable (allocable) sections in memory
- performing the relocation stage
- performing the symbol resolution stage

Here the symbol resolution stage can, possibly, be performed against any of the already loaded modules. New elements are then required for:

- keeping track of the loaded modules
- keeping track of the available symbols

As for the segments, the section header table holds the offsets, sizes and attributes of the sections. The runtime loader-linker knows which sections are loadable (sections of types SHT_PROGBITS or SHT_NOBITS holding the attribute flag SHF_ALLOC), and can compute the total size of memory to be allocated in order to hold the program.

Copying the content of the loadable sections to memory follows similar logic as previously described for the segments: the section headers give the offset to the loadable sections. The bss sections occupy no room in the file but must be allocated and zeroed out.

The memory allocation scheme for the loadable sections can be based on a page mechanism if an MMU is available or on any memory management algorithm if the available memory is seen as a pool. In any case, in order to optimize the memory usage (reduce fragmentation), the runtime loader-linker should try to allocate the loadable sections of the same category together as one entity, roughly composing the same segments as the link editor would have done. The decision regarding which loadable sections are grouped together is based on the following section attribute flags: SHF_WRITE (writable content) and SHF_EXECINSTR (executable content). A small complication comes from the fact that the sections have alignment requirements that must be accounted for in the computation of the total size of memory required for each category.

3.5. Relocations and Symbol Resolution

In the object modules, the references to global routines as well as the references to global variables being used in routines are not fixed since the addresses of the loadable sections were not known at build time. The link editor produced sections holding the relocation information, which describe the locations in the loadable sections that need to be modified to make the program executable. Once the loadable sections are installed in memory, it is possible to compute the relocations. The relocation computation involves the address of the item being referred to, as well as the content of the instruction (text) or memory cell (data). For the text, the modification depends on the addressing modes valid for the instruction being relocated.

Note that this process involves symbolic information: these relocations are associated with references to variables or routines defined within the object file. This is closely related to the next stage - the symbol resolution that, in turn, implies reference resolutions involving the relocation information. Consequently the runtime loader-linker can handle these two stages together.

Very likely, the code in the object module is making references to routines or variables that are defined outside of the object module itself. Resolving these references will link the module's code to the code which defines the routines or variables. These references are associated with symbols referred to as being undefined. As for the defined symbols, these symbols are described in the object module's symbol table, and are associated with relocation information. Their resolution implies that the items they correspond to are known in the system so that these items' addresses can be used when performing the relocation computations. A standard method for linkers is to keep track of the global symbols declared by each of the loaded modules. In order to do this, the loader-linker generally implements a symbol management service based on hash tables, for speed, and linked list of symbols, for extensibility. Then when a module is loaded while holding references to undefined symbols, the runtime loader-linker goes through the appropriate list of symbols, found via the hash function, and can walk the list until it finds a defined symbol of same name and type. At this point, it can use the address of the symbol and compute the relocation to do the binding between the call and the item (function or variable).

Conversely, the symbols defined in the loaded module are registered in the symbol table for later use by other modules.

Keeping track of the loaded modules themselves, i.e. registering at load time where their sections are located in memory, allows for unloading the modules, i.e. removing the module's content from the memory. In order to fully clean up the memory, the unloaded module's symbols must also be unregistered. This implies that some level of loaded module management resides in the system.

4. Static Shared Libraries

4.1. The Unix Approach

Linking the application code with shared libraries is a standard approach for the disc and memory usage problem that we described earlier. The application's references to the library symbols are bound to the symbol's addresses, but the library's code and data are not copied in the application's binary file, only their names are registered in this file, making the application leaner and faster to build.

When an application is loaded, and needs a shared library, the runtime loader-linker arranges to map the physical pages used by the library's text and data into the virtual address space of the application. This in effect shares the shared library between the applications that use it. Note that when the shared library is not already in use by some other application, the runtime loader-linker loads it in the system's memory, possibly creating a ripple effect of load operations of other libraries if this library makes use of others. Any change to the shared library data within the context of the application starts the copy-on-write mechanism.

Since the application's references to the library symbols must be resolved, the shared library has to be pre-linked against a known address. Although this is not a problem for the application itself, this has some consequences on the shared libraries: since they are going to be mapped in the application's address space and must be shared across applications, each of the static shared libraries must be assigned a specific pre-set location. Mapping the static shared library's content in the application's space raises the problem of sharing the process address space between the application's code and data and possibly several shared libraries that this application might be using, without creating conflicts. This implies a dedicated address space management for static shared libraries at the operating system level (John Levine terms this as being "black art" in [1]).

In order to be bound with the shared library, without having the library's code dragged in, the application is actually linked with a stub library which only holds the addresses of the symbols exported by the static shared library, but not code.

Additional elements must be considered when using static shared libraries: despite having many resemblance with static linking producing a full executable application file, this method produces incomplete executable application files (see details in [4]). So unlike with the full static linking method it is possible to fail running the application if the shared library is missing, or has been modified after the application was compiled. Also, the shared library's code may need initialization. In the full static linking scheme, this is done at application startup time. With static shared libraries some dedicated bootstrap code needs to be executed, before the application actually starts.

To give some freedom in upgrading static shared libraries without disturbing the applications, the applications are not linked to the actual addresses of the symbols in the shared library. Rather, they are linked to a jump table which gives access to the library's routines. Such an indirect reference method allows, up to a certain extent, to modify the content of the static shared library. Unfortunately, there is no such a simple method for the exported data: they must be grouped at the beginning of the data segment so that future evolutions of the library's private data do not disturb them. This requires cooperation from the library developers (see [1] for more comments on this aspect).

4.2. The Single Address Space Operating System Approaches

Single address space operating systems address very differently the sharing issue. The main advantage of a single address space is that the address of any object (routine or data) has a unique interpretation: whoever uses a given address always accesses the same object. The static binding between applications and shared libraries can be done in very much the same way as for Unix systems. So all static shared libraries can be attributed an area in this address space, and all applications can very naturally share them. The placement of the shared library seems however subject to the same troubles as for the Unix systems.

An advantage of a single address space is that the shared libraries are not transient: they can stay in memory even after the application is terminated. This makes the startup time of the application much smaller. Also the library's code and constants need no mapping in the application's space (although they do have to appear in the application's context if protection is to be granted by the MMU). So a simple linking scheme based on the relocatable object module, as previously described, provides "natural" static shared library support since such a module is linked once, at load time, and can be made visible to other modules.

However such a scheme (called Global Static Linking in [3]) makes difficult to have private data in the libraries. Every application shares the global variables, creating a potential code reentrancy issue. This can be solved via mechanisms based on per-client copy of the library's data segment (see [3], [5] and [11]).

5. Shared Libraries and Dynamic Linking

5.1. The Traditional Approach

Considering the complexity of creating static shared libraries, and the application address space management they involve, modern Unix version, as well as Microsoft Windows, implement a more flexible approach to shared libraries: the dynamic linking.

In such a scheme, the library is not bound to any specific address and is intended to be mapped at any address. Few static linking is involved: it is the responsibility of the runtime loader-linker to link the application with the shared library either at load time, or at run time. The code must then be position independent, also called PIC. Such a code does not have any absolute symbol references. All references are relative to the PC, or based on an address loaded in a register via the mechanism described below. Since the actual addresses of the library's symbols are not known before the library is loaded, most of the linking is deferred until an application needs the library. Aside of facilitating using and upgrading shared libraries, this scheme also allows the application to load and unload shared libraries on demand (run time).

According to [1], all the Unix-like implementations of the dynamic shared libraries are based on the ELF format and use position independent code, although according to [4] this is not compulsory. PIC reduces the amount of relocation necessary when the shared library's code is mapped in the process space. The library's code cannot directly reference global and static data symbols since their addresses are not known until after the library is loaded and mapped in the process space. Resolving these references by fixing up the text instructions (as for the relocatable object modules) is not possible since the code will then no longer be sharable. The solution is that the link editor produces a table of addresses (Global Offset Table) where each entry corresponds to one global or static variable used in the library. The content of the GOT is used via a base register as described in [1] and [9].

The linkage between the application and the shared library is partially done by the link editor: calls to outside routines do not remain unresolved as in a relocatable object module, but are actually statically linked to a jump table (procedure linkage table [PLT] in ELF terminology). The PLT is a set of stubs, one per outside routine used in the application. Each one of these stubs does a jump to an address found in an address table (a GOT too). The GOT holds the addresses of the outside routines but since these addresses are not known when the application is compiled, it is the role of the runtime loader-linker to fill it with the appropriate values. If the application makes direct data references to the library's symbols, then entries in the GOT are created for these symbols as well.

The exact process is fairly complex and involves interesting optimizations, such various ways to execute the runtime loader-linker itself, or the lazy binding of symbol references (see [1], [4], [6] and [9] for detailed explanations). Basically, the runtime loader-linker uses both the application binary file's relocation information and symbolic information in order to discover which of the undefined symbols corresponds to a given entry in the GOT.

Along with its obvious advantages, this traditional approach of the dynamically linked shared libraries has few drawbacks. For instance, like an object file, a shared library is actually a collection of functions. So the granularity of a shared library is the shared library itself, whereas individual modules could be extracted from static libraries. Although it is often mentioned in the literature that libraries using PIC tend to be slower and bigger than libraries using non-PIC, this is not our experience. At least with RISC CPUs the code of both PIC and non-PIC shared libraries is very comparable when generated by recent compiler tool chains. However, creating the GOTs and PLTs, and resolving them at load or run time, introduces some overhead.

For some languages (C++ in particular), the runtime binding can be slow because of the profusion of symbols they introduce, although this is counter balanced by the reduction in the number of relocations in PIC. This reduction is a side effect of the PLT: instead of having as many relocations as external function calls in the application, only one PLT entry and one GOT entry for each external function has to be bound. All the calls to a given function are statically linked to its dedicated PLT entry by the link editor (see [10]).

Dynamic shared library changes are very naturally handled since, the application is by default re-linked to the newest version of the shared library. A version number scheme allows for keeping track of changes, and lets application free to be bound to a specific version of the library if this is required.

5.2. The Single Address Space Operating System Approaches

As for the static shared libraries, the totally different paradigm implemented in many single address space operating systems brings forward new ways of supporting dynamically linked shared libraries. In Opal code modules are statically linked into the global address space, but can be shared and dynamically loaded ([3]). In Mungi, there is no true dynamic linking: instead the modules are made executable and have a “module descriptor object” which declares all the function imported from the module ([5]).

Dynamically linked libraries bring an answer to the following problem: sharing multiple libraries between applications, without location conflicts, and with enough flexibility for shared library code evolution. They also bring a useful “load on demand” scheme. Wide addressing range (64-bit) single address space operating systems do not have location conflicts because they have room for all (?) shared libraries. The flexibility aspect is obtained through the wide space available and the usage of jump tables to isolate the applications from the changes in the libraries. The shared library’s private data issue can be solved as described for the static shared library. The “load on demand” scheme is not an issue. So, although these systems can certainly support dynamically linked libraries, they probably don’t need it.

Small addressing range single address space operating systems (32-bit) may have an important location conflict issue. In this situation support for dynamically linked libraries following the Unix model is useful. Also, a simple loading and linking scheme based on the relocatable object modules, provides “natural” dynamically linked shared library support according to [3]. Indeed, when the memory space is flat for the entire system, that is no address overlapping exists, PIC is no longer necessary either because the library is prepared to be loaded at a given address or because the runtime loader-linker links it there.

Although [5] considers that the libraries are smaller and the runtime binding has been measured as significantly faster when non-PIC shared libraries and applications are used, this is not our observation. As said earlier PIC routines and non-PIC routines look very much alike for RISC processors when produced by recent compilers (position independent data support does introduce a small difference though).

An important advantage of single address space systems is that the libraries can easily be made permanent (see the concept of library caching in [10]), speeding up the application binding time.

5.3. An Alternate Approach

An alternate solution, developed by our company, is a hybrid between a single address space operating system and a private address space operating system, such as Unix. In this system, the applications have their own address space, but shared libraries, as well as the kernel, share a single address space. In such a system, it is possible to have per-application copies of the shared library’s global and static data, while keeping the advantages of loading relocatable object modules. When used, the shared libraries are parts of the application’s private space but present the same base address to all applications during their lives. Thus, no PIC is necessary so the generated code stays somewhat simpler. Shared libraries, as well as applications are not transient in this system. Their code stays even if the application’s thread of execution exited, allowing for a quick re-start of the application. The kernel itself is seen as a special, privileged shared library in this system. The per-application data copy does not apply in this case to ensure a complete isolation of the kernel.

In this model, the shared library can be generated statically, based on a description of what the library exports, and where it goes in memory. This is close, conceptually, from what has been described earlier for the Unix operating systems. Dynamic shared libraries are also supported but unlike for the static linking situation, the dynamic linking of a shared library is not transparent for the applications. In the current version of this system: the new shared library domain has to be explicitly created, and the runtime loader-linker needs to be called by the application's code or directly by the user through the shell. Future versions will see the addition of schemes making the support of dynamically linked libraries transparent for the applications.

In this system, both shared libraries and applications can be seen as containers for units of code (called component modules). The runtime loader-linker knows how to link these units, which are in essence relocatable files, so that applications or shared libraries can be easily built incrementally. Outbound calls from applications and shared libraries are made through a table analogous to the ELF Procedure Linkage Table. The shared library's routines visibility outside of the library can be controlled so that only the selected routines (called entry points) are usable by an application when it is statically linked to the shared library. The runtime loader-linker performs the same control when dynamically linking application's code and shared library's entry points. Since the kernel is handled as a restricted shared library it is very easy to dynamically extend it, just by downloading component modules into the kernel's address space.

The shared library data are automatically copied in per-client physical pages when the shared library is attached by an application. These pages are mapped to the same start address for all the clients and the shared library itself, so that non-PIC code can be used. The system ensures that the appropriate pages are made visible for each client application in turn.

Although this is not yet implemented, applications can easily be loaded as executable code in such a model. However, shared libraries still need to be relocatable since they can not share overlapping range of addresses and an application might want to use several shared libraries at a time. Note that it would be possible to load them as executable, but this would greatly reduce the flexibility of the system.

As said earlier, an important feature of the ELF format is its flexibility. This quality allows for the addition of new sections, which can be used by the system's runtime loader-linker. Such sections can be added by way of assembly code, with special extensions to the C language (such as the GNU __attribute__), as described in [9], or with the linker's command language (for instance, see [12] for the GNU link editor's command language). In our hybrid system, this allows for declaring the shared library's attributes as well as its public interface (the entry points).

6. Runtime Software Component Replacement

To some extent, traditional dynamically linked shared libraries can be used to implement a runtime software component replacement mechanism, but this seems to be somewhat cumbersome. Software component replacement generally applies to the operating system itself so is sensitive to issues such as granularity, configuration, trust, etc. (see [8] for a discussion on this subject). ELF shared objects are generally big because they gather many routines in one executable file. Although nothing prevents for building module-size ELF shared objects, PIC would imply having many PLTs and GOTs in the operating system's space. Also, in order to take advantage of the update facility allowed by PIC, many of the system services would have to be themselves position independent code.

As noted in [13], using dynamically linked libraries for software replacement is limited by the fact that the code using the shared library has to be aware of the replacement mechanism itself. For instance, to test the presence of the library or doing the load requests if the library is not there. This can introduce undesirable latencies in the service replacement. The dependencies on other dynamically linked libraries introduced by the traditional runtime loader-linker may yet increase this latency. Also there is neither built-in interface verification mechanism, nor is there any protection against updating components while they are in use.

Our system enables the implementation of a more complete and secure runtime software component mechanism, based on the component modules. The runtime loader-linker sub-system is able to replace a given component module, by unloading the current version and reloading a replacement one. Since our technology is based on standard relocation and symbol resolution stages, the runtime loader-linker implements a patented technique for flexible software linking. This technique allows for unloading and reloading respectively old and new versions of a component module while ensuring that the linkages with other component modules are re-established. If well controlled this allows for very reduced latencies in the replacement of services. Although this is not available yet, this feature could be extended to allow replacement of entire shared libraries in one shot.

In its current state, the interface verification done by the runtime loader-linker is minimal since the ELF format does not allow, by default, for identification of the type and interface of an undefined symbol. Although languages such as C++ would probably be of some help, this is not a generic enough solution in the embedded industry where C is still the preferred programming language. In addition, the existing update mechanism requires that the threads possibly executing the code being replaced be stopped, or paused in a safe place.

Future implementations will take advantage of ideas coming from the High Availability area, in particular the notion of “fail over”. In this case, the new version of the component is loaded while the old version is still in use, so that after the transition between these two versions is done, it is still possible to come back quickly to the previous version. Cooperation between the application threads and the runtime loader-linker is still necessary however to ensure the safety of the transition period.

7. Conclusion

We have seen that the traditional approach for the runtime loader-linker concepts keeps its advantages in some situations, with particular regard to per-process copy of the shared library data. We observed that the notion of a shared library takes on a new meaning in the context of single address space operating systems. We have seen that loading relocatable code (object files) is an interesting alternative to the standard model of the executable files for both application and shared library. The hybrid approach developed by Wind River tries to take the best of both the traditional operating system world and the single address space operating system world and brings greater flexibility, in particular with regard to software component replacement.

References

[1] John Levine. *Linkers & Loaders*. Morgan Kaufmann Publishers, 2000.
[2] Robert Gingell, Joseph Moran, William Shannon. *Virtual Memory Architecture in SunOS*. In USENIX Conference Proceedings. Summer 1987.
[3] Jeffrey Chase, Henry Levy, Michael Feeley and Edward Lazowska. *Sharing and Protection in a Single Address Space Operating System*. ACM Transactions on Computer Systems. May 1994.
[4] Robert Gingell, Meng Lee, Xuong Dang and Mary Weeks. *Shared Libraries in SunOS*. In USENIX Conference Proceedings. Summer 1987.
[5] Luke Deller and Gernot Heiser. *Linking Programs in a Single Address Space*, in USENIX Conference Proceedings. Summer 1999.
[6] Intel. *Executable and Linkable Format (ELF)*. Tool Interface Standards (TIS). Portable Formats Specification, Version 1.1.
[7] Steve Zucker and Kari Karhi. *System V Application Binary Interface. PowerPC Processor Supplement*. . Sunsoft Documents Part No: 802-3334-10, Revision A. September 1995.

[8] Margo Seltzer, Yasuhiro Endo, Christopher Small and Keith Smith. *Issues in Extensible Operating Systems*. Technical Report TR-18-97, Harvard University, 1997

[9] Hongjiu Lu. *ELF: From the Programmer's Perspective*. http://citeseer.nj.nec.com/lu95elf.html. May 17, 1995.

[10] Michael Nelson and Graham Hamilton. *High Performance Dynamic Linking Through Caching*. In USENIX Conference Proceedings. Summer 1993.

[11] Anders Lindström, John Rosenberg and Alan Dearle. *The Grand Unified Theory of Address Spaces*. In Proceedings of the 5th workshop on hot topics in operating systems (HotOS). IEEE. May 1995.

[12] Steve Chamberlain. *Using ld. The GNU linker. ld version 2*. http://www.gnu.org/manual/ld-2.9.2/ld.html. January 1994.

[13] Michael Hicks. *Dynamic Software Updating*. http://citeseer.nj.nec.com/336947.html. October 1999.