Dynamic Loading of Native L4 Programs

Frank Mehnert

Dresden University of Technology
Department of Computer Science
D-01062 Dresden, Germany
email: fm3@os.inf.tu-dresden.de

1 Introduction

In this paper, I introduce the design and implementation of an L4 loader which can start native L4 applications. Although some services of the existing resource manager (RMGR) are still needed, the loader replaces the functionality to start tasks. While the RMGR can only start L4 applications talking the *Sigma0* protocol [3] at the system boot time, the L4 loader is much more flexible. It can start *L4 environment* applications which can depend on shared libraries as well as legacy *Sigma0* applications such as L^{4}Linux [2]. L4 applications started by the RMGR have to be linked to disjunct virtual addresses. In contrast, multiple instances of an application started by the loader can run at the same time[1]. Using shared libraries, the memory usage can be reduced.

2 Structure

The L4 loader depends on services of the L4 environment. Besides the loader server, there exists an *exec layer* which can interpret *ELF* binaries (Figure 1).

2.1 Components

The Loader scans a configuration file describing requirements of the task and prepares all resources that are needed to start an L4 task. It can be controlled by command line arguments as well as by the IDL interface.

The Exec Layer is responsible for interpreting the program binary. Although at the moment the only binary format the *exec layer* can interpret is *ELF* [6], it should be possible to extent the server to work with other executable formats, such as Windows' *Portable Execution Format* [5].

[1]This is not possible with *Sigma0* clients which depend on direct-mapped program sections

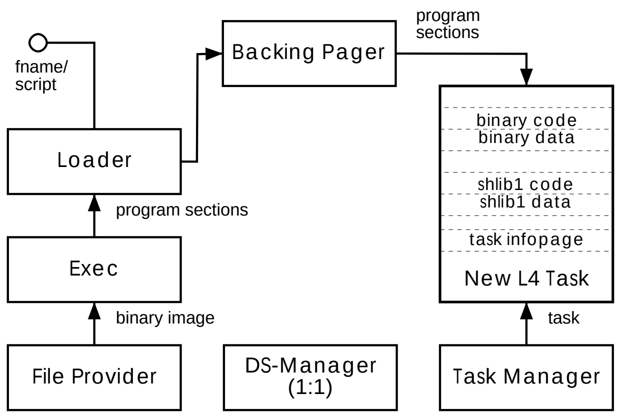

Figure 1 L4 Loader scenario

The *exec* server is a *from-scratch* implementation in C++ – no code was taken from Linux or anywhere else.

The File Provider offers a file access service. It delivers a dataspace which contains the binary image of the requested file. There are two arguments for the *open* function: The filename and the thread identifier (thread ID) of the dataspace manager which should be used to allocate the dataspace.

Currently, there are two implementations of file providers available: A simple L4 TFTP server developed from GRUB and an L^{4}Linux file provider.

A future version of a file provider could be implemented as a dataspace manager allowing paging on-demand.

The Backing Pager provides paging support in particular for program sections which are needed during the applications startup process (region manager, startup library). The loader registers program section dataspaces of these libraries together with an L4 task identifier. Incoming page faults of a task will be translated into the proper offset of the appropriate dataspace. The pagefaults will be handled by the manager of the dataspace. While this is currently a simple dataspace manager for pinned (not swap-able) memory, this could be a file provider which loads data from hard disk or network on demand in future implementations.

Currently, this service is included in the L4 loader but it is planned to move that functionality to an own server and to use the same algorithm that is used in the region manager to find the dataspace appropriate to an page fault address inside a task.

2.2 Starting an L4 Task

The loading process of an L4 application consists of the following steps:

1. Preparation: At first the loader gets the filename of the application which should be started. The loader allocates a new *task infopage* and fills in important information needed during the task's loading process (thread ID of the *name server*, thread ID of the default *dataspace manager*, thread ID of the file provider, ...).

The filename and a the *task infopage* are delivered to the binary interpreter *exec*.

2. Load binary: To interpret the binary image, *exec* reads the thread identifier of the file provider from the *task infopage* and requests the file image of the binary from that server. Then it parses the ELF header of the binary image and saves data which is needed later for loading and executing the binary (text sections, data sections, shared symbols together with their memory offset used in the linking process, debug symbols). Then it creates a dataspace for each program section. These sections contain the program code and static data.

3. Load shared libs: Furthermore, all shared libraries the binary depends on get loaded if not yet done.

4. Create program sections: After that, logical entries for each program section of the L4 task will be created and written to the *task infopage*.

Each program section (text and data sections of both the binary and all shared libraries) will be duplicated by a *copy-on-write* dataspace server. The dataspaces IDs will be included together with their relocation address into the *task infopage*. The GNU linker relocates shared libraries to address 0 so they have to be relocated to the target address space before they can be used. This is done in the next step.

5. Boot L4 task: Now *exec* passes the control back to the loader. All program sections of the new task which have a known relocation address get registered at the *backing pager*.

There are two special shared libraries which must have a fixed relocation address: the *loader library* and the *L4 region manager*. Because both libraries are needed during the task's startup process to register program sections and to initialize basic *L4env* services (thread service, semaphore service), they have to be relocated to a predefined position of the new task's address space. The *region manager* cannot page itself, so both libraries are paged by the *backing pager*.

A new task is started at the *task server* and the initial instruction pointer is set to the entry of the *loader library*.

6. Relocate program sections: The new L4 task starts executing the *loader library* initializing the *L4env* services and locating each program section which do not have any predefined relocation address. Finally, each program section is registered at the *region manager* but symbol references of some program sections are not yet solved because their relocation address was not yet known during the first linker step (see step "Create program sections").

7. Second Linker step: Therefore the *exec layer* performs a second linker step linking all program sections which are still not linked. This is now possible because all sections are relocated.

8. Start L4 task: Now the control is passed back to the L4 task and either the function *multiboot_main()* (if exists) or `multiboot_main()` is called. The startup code is not executed. Because all L4 environment library services are initialized now, the task can use them without initializing them explicitly.

With the separation of the linker process in two steps ("Creating program sections" and "Second Linker step"), the region manager of the new L4 task has full control over the relocating process of the program sections.

2.3 Terminating an L4 task

The current basic approach for resource management is very simple: The loader knows of each server of the system which provides resources (name server, task server, dataspace manager ...). When an L4 task is to be killed, each of these server is asked to free all resources the killed task owns. This centralized strategy has the disadvantage, that the loader has to have knowledge of all resource servers of the L4 system.

In the future, we want to develop a resource model that avoids these drawbacks.

Currently, a L4 program can only be killed by the loader. In future versions, there will be a library function `exit()` in the *loader library* which can be called by the L4 program to terminate itself.

3 Compatibility

For compatibility issues, there exists a special emulation mode of the L4 loader which allows to load legacy L4 applications talking the *Sigma0* protocol. A prominent representative of such an application is L^{4}Linux.

Using a configuration file, the reservation of resources the loader does for L4 tasks can be controlled. The most important keywords are

`direct_mapped` ensures that all program sections (text, data, bss) are mapped one-to-one. In the current version of the L4 loader, direct mapped memory is allocated at the RMGR. In future versions, special dataspace managers will offer such services.

`memory` allows to reserve an amount of memory which can be accessed using the *Sigma0* protocol.

`dma_able` ensures that memory reserved by the `memory` keyword can be used for ISA-DMA, i.e. lays below 16 MB. In Linux, the `scsi_init_malloc()` function for instance needs such memory.

See [4] for the full syntax description.

L^{4}Linux can handle non-contiguous memory so the program sections, the DMA-able memory, and the regular memory don't need to be contiguous. In future versions of L^{4}Linux is could be needless to map code and data sections one-to-one.

4 Real Life

The loader can be used either to start L4 applications while the system is booting (as the RMGR does) or to dynamic start L4 applications from other L4 applications, e.g. from L^{4}Linux.

Here is an example GRUB menu file how to start L^{4}Linux at boot time:

```
kernel (nd)/tftpboot/fm3/rmgr -sigma0
module (nd)/tftpboot/fm3/fiasco -nowait
module (nd)/tftpboot/fm3/sigma0
module (nd)/tftpboot/fm3/log
module (nd)/tftpboot/fm3/names
module (nd)/tftpboot/fm3/buffermgr
module (nd)/tftpboot/fm3/simple_dm
module (nd)/tftpboot/fm3/simple_ts
module (nd)/tftpboot/fm3/tftp
module (nd)/tftpboot/fm3/l4exec
module (nd)/tftpboot/fm3/loader \
       (nd)/tftpboot/fm3/cfg_linux
```

The `cfg_linux` configuration script defines some special requirements for L^{4}Linux:

```
verbose 0
modpath "(nd)/tftpboot/fm3"

task "glinux"
  "root=/dev/sda5 mem=48M" direct_mapped
  memory  2MB dma_able
  memory 18MB
```

The tftp server stops working when L^{4}Linux is up because the Linux network driver re-initializes the network card.

After L^{4}Linux is up, it can start other L4 tasks using the Linux file provider `fprov-l4` and the loader client `run-l4`. `Fprov-l4` uses the Linux infrastructure to access the requested files.

5 Fiasco extensions

The L4 loader is able to provide debugging symbols and debugging line information for Fiasco in a special format. The keywords `fiasco_symbols` and `fiasco_lines` do that job. Fiasco uses an `int3` extension for registering such debugging information.

6 Future work

The following features are planned for future work:

- Implementation of an L4 loader client that allows to dynamic start L4 applications without L^{4}Linux
- Implementation of an L4 environment resource and access right maintenance model: With the L4 environment, the current RMGR-based resource model does not make sense anymore because resources are not maintained by one central server but by several servers. Furthermore resources like dataspaces can be transfered between clients.

- Paging-on-demand in file provider.
- Determining the loading and running costs of shared libraries.
- Current versions of GNU cc and g++ produce lots of expensive run-time relocations. This translates into slow startup of large applications with many symbol references to shared libraries. By collecting all code references from an ELF object to another and replacing them by an additional indirect jump, the linker overhead could be reduced dramatically [1].
- PE-like optimization of ELF link process: Shared libraries could be pre-relocated to default locations. If the address range of the library's program sections are not used, that could save the cost of relocation. First candidates for pre-relocation are the *load library* and the *region manager*.

References

[1] Waldo Bastian. Making C++ ready for the desktop, version 1.2. http://www.suse.de/~bastian/Export/linking.txt, May 2001.

[2] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Saint-Malo, France, October 1997.

[3] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996.

[4] Frank Mehnert. *L4 Loader Reference Manual*, 2001. http://os.inf.tu-dresden.de/ fm3/doc/loader.

[5] Tool Interface Standard Committee (TIS). *Formats Specification for Windows, Version 1.0*, February 1993. http://developer.intel.com/vtune/tis.htm.

[6] Tool Interface Standard Committee (TIS). *Portable Formats Specification, Version 1.1*, October 1993. http://developer.intel.com/vtune/tis.htm.