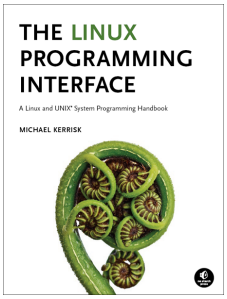
Building and Using Shared Libraries on Linux

Course code: M7D-SHLIB04

This course provides a thorough understanding of the process of designing, building, and using shared libraries on Linux. Detailed presentations coupled with carefully designed practical exercises provide participants with the knowledge needed to understand, design, create, and deploy shared libraries.



Audience and prerequisites

The primary audience comprises designers and programmers building and deploying shared libraries on Linux. Systems administrators are likely to also find the course of benefit for the purpose of troubleshooting problems with shared libraries.

Participants should have a good reading knowledge of the C programming language and some programming experience in a language suitable for completing the course exercises (e.g., C, C++). No previous experience of working with shared libraries is required.

Course materials

- A course book (written by the trainer) that includes all course slides and exercises
- An electronic copy of the trainer's book, *The Linux Programming Interface*
- A source code tarball containing all of the example programs written by the trainer to accompany the presentation

Course duration and format

2.5 days, with around 40% devoted to practical sessions.

Course inquiries and bookings

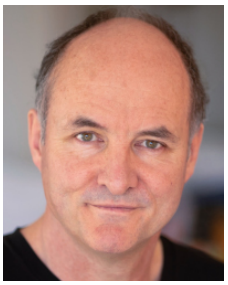
For inquiries about courses and consulting, you can contact us in the following ways:

- Email: training@man7.org
- Phone: +49 (89) 2488 6180 (German landline)

Prices and further details

For course prices, upcoming course dates, and further information about the course, please visit the course web page, `http://man7.org/training/shlib/`.

About the trainer



Michael Kerrisk has a unique set of qualifications and experience that ensure that course participants receive training of a very high standard:

- He has been programming on UNIX systems since 1987 and began teaching UNIX system programming courses in 1989.
- He is the author of *The Linux Programming Interface*, a 1550-page book acclaimed as the definitive work on Linux system programming.
- He has been actively involved in Linux development, working with kernel developers on testing, review, and design of new Linux kernel–user-space APIs.
- Since 2000, he has been the involved in the Linux *man-pages* project, which provides the manual pages documenting Linux system calls and C library APIs, and was the project maintainer from 2004 to 2021.

Building and Using Shared Libraries on Linux: course contents in detail

Topics marked with an asterisk (*) are optional, and will be covered as time permits

1. **Course Introduction**
2. **Fundamentals of Shared Libraries**
   - Background
   - The static linker and the dynamic linker
   - Static vs shared libraries
   - Basics of shared library creation and use
   - Position-independent code (PIC)
   - The shared library soname
   - In pictures: library creation, linking, and loading
3. **Versioning and Installation**
   - Shared library versioning
   - Shared library real names, sonames, and linker names
   - Installing shared libraries
   - *ldconfig*
4. **ELF (Executable and Linkable Format)**
   - ELF file layout
   - The program header table (PHT)
   - The section header table (SHT)
   - Program header table vs section header table
   - ELF sections
   - Useful commands: *readelf* and *objdump*
5. **The Dynamic Linker**
   - Rpath: specifying library search paths in an object
   - Dynamic string tokens
   - Finding shared libraries at run time
   - How programs get run
6. **Symbol Interposition and Library Load Order**
   - Symbol resolution and symbol interposition
   - Symbol resolution and library load order
   - Link-map lists
   - The global look-up scope
   - `LD_DEBUG`: tracing the dynamic linker
7. **Dynamically Loaded Libraries (*dlopen*)**
   - Opening a shared library: *dlopen()*
   - Obtaining the address of a symbol: *dlsym()*
   - The *dlopen* API: example
   - Use cases
   - The *dlopen* API: further details
8. **Shared Libraries and the Static Linker**
   - Recording dynamic dependencies
   - How the static linker finds library dependencies
   - Handling secondary dependencies at link time
9. **Symbol Visibility**
   - Controlling symbol visibility
   - Controlling symbol visibility: *–Bsymbolic*
   - Symbol attributes: binding and visibility
   - Controlling visibility on a per-symbol basis
   - Using version scripts to control symbol visibility
   - Summary of techniques for visibility control
   - Run-time visibility control: *dlopen()*-ed libraries
10. **Look-up Scopes**
    - Look-up scopes
    - `LD_DEBUG=scopes`
11. **Preloading**
    - Preloading shared libraries
    - Preloading example
    - Use cases for preloading
12. **Weak Symbols (*)**
    - Weak symbols
    - Linker rules for strong and weak symbols
    - Use cases for weak symbols
    - Use case: testing whether a symbol definition exists
    - Use case: overridable weak implementation
    - Use case: overridable weak alias
13. **Symbol Versioning**
    - Creating a symbol-versioned library
    - An aside: version "dependencies"
    - Summary: assigning a version to a symbol
    - Symbol versioning and symbol resolution
    - ELF and symbol versioning
    - Advantages of symbol versioning
    - Referencing a nondefault symbol version
    - Removing a public versioned symbol
    - The library base version
    - Appendix: version node syntax
    - Appendix: the `.symver` assembler directive
14. **Symbol Versioning: Further Topics (*)**
    - Symbol versioning design approaches
    - Transitioning an existing library to symbol versioning
    - Further details on symbol versioning
    - Symbol-version matching rules
    - Addendum: a few C++ details
15. **Lazy Binding (*)**
    - Lazy binding
    - Immediate binding
    - Lazy binding versus immediate binding
16. **GOT and PLT (*)**
    - The GOT and PLT
    - Relocation and the PLT: in pictures
    - Relocation and the PLT: code
    - Observing the effect of lazy binding on the GOT
    - Performance considerations