An introduction to application development tools in Red Hat Enterprise Linux 7

Vladimír Slávik
Red Hat Customer Content Services
vslavik@redhat.com
Abstract

This document describes the different features and utilities that make Red Hat Enterprise Linux 7.5 an ideal enterprise platform for application development.
# Table of Contents

**PREFACE** ................................................................. 7.

**PART I. SETTING UP A DEVELOPMENT WORKSTATION** ................................................................. 8.

**CHAPTER 1. INSTALLING THE OPERATING SYSTEM** ................................................................. 9.
   Additional Resources 9

**CHAPTER 2. SETTING UP TO MANAGE APPLICATION VERSIONS** ................................................................. 10.
   Additional Resources 10

**CHAPTER 3. SETTING UP TO DEVELOP APPLICATIONS USING C AND C++** ................................................................. 11.
   Additional Resources 11

**CHAPTER 4. SETTING UP TO DEBUG APPLICATIONS** ................................................................. 12.
   Additional Resources 12

**CHAPTER 5. SETTING UP TO MEASURE PERFORMANCE OF APPLICATIONS** ................................................................. 13.
   Additional Resources 13

**CHAPTER 6. SETTING UP TO DEVELOP APPLICATIONS USING JAVA** ................................................................. 14.

**CHAPTER 7. SETTING UP TO DEVELOP APPLICATIONS USING PYTHON** ................................................................. 15.
   Python versions corresponding to Red Hat Software Collections packages 15
   Additional Resources 15

**CHAPTER 8. SETTING UP TO DEVELOP APPLICATIONS USING C# AND .NET CORE** ................................................................. 16.
   Additional Resources 16

**CHAPTER 9. SETTING UP TO DEVELOP CONTAINERIZED APPLICATIONS** ................................................................. 17.
   Additional Resources 17

**CHAPTER 10. SETTING UP TO DEVELOP WEB APPLICATIONS** ................................................................. 18.
   Ruby on Rails versions corresponding to Red Hat Software Collections packages 18
   Additional Resources 18

**PART II. COLLABORATING ON APPLICATIONS WITH OTHER DEVELOPERS** ................................................................. 19.

**CHAPTER 11. USING GIT** ................................................................. 20.
   Installed Documentation 20
   Online Documentation 20

**PART III. MAKING AN APPLICATION AVAILABLE TO USERS** ................................................................. 21.

**CHAPTER 12. DISTRIBUTION OPTIONS** ................................................................. 22.
   RPM Packages 22
   Software Collections 22
   Containers 22
   Additional Resources 22

**CHAPTER 13. CREATING A CONTAINER WITH AN APPLICATION** ................................................................. 23.
   Prerequisites 23
   Steps 23
   Additional Resources 24

**CHAPTER 14. CONTAINERIZING AN APPLICATION FROM PACKAGES** ................................................................. 25.
   Prerequisites 25
   Steps 25
PART IV. CREATING C OR C++ APPLICATIONS

CHAPTER 15. BUILDING CODE WITH GCC

15.1. RELATIONSHIP BETWEEN CODE FORMS

Prerequisites

Possible Code Forms
Handling of Code Forms in GCC
Additional Resources

15.2. COMPILING SOURCE FILES TO OBJECT CODE

Prerequisites
Steps
Additional Resources

15.3. ENABLING DEBUGGING OF C AND C++ APPLICATIONS WITH GCC

Enabling Creation of Debugging Information with GCC
Additional Resources

15.4. CODE OPTIMIZATION WITH GCC

Code Optimization with GCC
Additional Resources

15.5. HARDENING CODE WITH GCC

Release Version Options
Development Options
Additional Resources

15.6. LINKING CODE TO CREATE EXECUTABLE FILES

Prerequisites
Steps
Additional Resources

15.7. C++ COMPATIBILITY OF VARIOUS RED HAT PRODUCTS

Additional Resources

15.8. EXAMPLE: BUILDING A C PROGRAM WITH GCC

Prerequisites
Steps
Additional Resources

15.9. EXAMPLE: BUILDING A C++ PROGRAM WITH GCC

Prerequisites
Steps

CHAPTER 16. USING LIBRARIES WITH GCC

16.1. LIBRARY NAMING CONVENTIONS

Additional Resources

16.2. STATIC AND DYNAMIC LINKING

Comparison of static and dynamic linking
Reasons for static linking
Additional Resources

16.3. USING A LIBRARY WITH GCC

Compiling Code That Uses a Library
Linking Code That Uses a Library
Compiling and Linking Code Which Uses a Library in One Step
Additional Resources

16.4. USING A STATIC LIBRARY WITH GCC

Prerequisites
Steps

16.5. USING A DYNAMIC LIBRARY WITH GCC
Prerequisites
Linking a Program Against a Dynamic Library
Using a rpath Value Stored in the Executable File
Using the LD_LIBRARY_PATH Environment Variable
Placing the Library into the Default Directories

16.6. USING BOTH STATIC AND DYNAMIC LIBRARIES WITH GCC
Prerequisites
Introduction
Specifying the static libraries by file
Using the -Wl option
Additional Resources

CHAPTER 17. CREATING LIBRARIES WITH GCC
17.1. LIBRARY NAMING CONVENTIONS
Additional Resources
17.2. THE SONAME MECHANISM
Prerequisites
Problem Introduction
The soname Mechanism
Reading soname from a File
17.3. CREATING DYNAMIC LIBRARIES WITH GCC
Prerequisites
Steps
Additional Resources
17.4. CREATING STATIC LIBRARIES WITH GCC AND AR
Prerequisites
Steps
Additional Resources

CHAPTER 18. MANAGING MORE CODE WITH MAKE
18.1. GNU MAKE AND MAKEFILE OVERVIEW
Prerequisites
GNU make
Makefile Details
Typical Makefile
Additional resources
18.2. EXAMPLE: BUILDING A C PROGRAM USING A MAKEFILE
Prerequisites
Steps
Additional Resources
18.3. DOCUMENTATION RESOURCES FOR MAKE
Installed Documentation
Online Documentation

CHAPTER 19. USING THE ECLIPSE IDE FOR C AND C++ APPLICATION DEVELOPMENT
Using Eclipse to Develop C and C++ Applications
Additional Resources

PART V. DEBUGGING APPLICATIONS

CHAPTER 20. DEBUGGING A RUNNING APPLICATION
20.1. ENABLING DEBUGGING WITH DEBUGGING INFORMATION
20.1.1. Debugging Information
Additional Resources
20.1.2. Enabling Debugging of C and C++ Applications with GCC
   Enabling Creation of Debugging Information with GCC
   Additional Resources
20.1.3. Debuginfo Packages
   Prerequisites
   Debuginfo Packages
20.1.4. Getting debuginfo Packages for an Application or Library using GDB
   Prerequisites
   Procedure
   Additional Resources
20.1.5. Getting debuginfo Packages for an Application or Library Manually
   Prerequisites
   Procedure
   Additional Resources

20.2. INSPECTING APPLICATION INTERNAL STATE WITH GDB
20.2.1. GNU Debugger (GDB)
   GDB Capabilities
   Debugging Requirements
20.2.2. Attaching GDB to a Process
   Prerequisites
   Starting a Program with GDB
   Attaching GDB to an Already Running Process
   Attaching an Already Running GDB to an Already Running Process
   Additional Resources
20.2.3. Stepping through Program Code with GDB
   Prerequisites
   GDB Commands to Step Through the Code
   Additional Resources
20.2.4. Showing Program Internal Values with GDB
   Prerequisites
   GDB Commands to Display the Internal State of a Program
   Additional Resources
20.2.5. Using GDB Breakpoints to Stop Execution at Defined Code Locations
   Prerequisites
   Using Breakpoints in GDB
   Additional Resources
20.2.6. Using GDB Watchpoints to Stop Execution on Data Access and Changes
   Prerequisites
   Using Watchpoints in GDB
   Additional Resources
20.2.7. Debugging Forking or Threaded Programs with GDB
   Prerequisites
   Debugging Forked Programs with GDB
   Debugging Threaded Programs with GDB
   Additional Resources

20.3. RECORDING APPLICATION INTERACTIONS
20.3.1. Tools Useful for Recording Application Interactions
   Additional Resources
20.3.2. Monitoring an Application’s System Calls with strace
   Prerequisites
   Steps
   Additional Resources
20.3.3. Monitoring Application’s Library Function Calls with ltrace
CHAPTER 21. DEBUGGING A CRASHED APPLICATION ........................................... 70
  21.1. CORE DUMPS .................................................................................. 70
      Prerequisites .................................................................................. 70
      Description .................................................................................. 70
  21.2. RECORDING APPLICATION CRASHES WITH CORE DUMPS .......... 70
      Steps ............................................................................................ 70
      Additional Resources .................................................................. 71
  21.3. INSPECTING APPLICATION CRASH STATES WITH CORE DUMPS .. 71
      Prerequisites ................................................................................ 71
      Steps ............................................................................................ 71
      Additional Resources .................................................................. 73
  21.4. DUMPING PROCESS MEMORY WITH GCORE ................................. 73
      Prerequisites ................................................................................ 73
      Steps ............................................................................................ 73
      Additional resources ................................................................... 74
  21.5. DUMPING PROTECTED PROCESS MEMORY WITH GDB ................ 74
      Prerequisites ................................................................................ 74
      Steps ............................................................................................ 74
      Additional Resources .................................................................. 74

PART VI. MONITORING PERFORMANCE ......................................................... 75

CHAPTER 22. VALGRIND ............................................................................. 76
  22.1. VALGRIND TOOLS ......................................................................... 76
  22.2. USING VALGRIND .......................................................................... 77
  22.3. ADDITIONAL INFORMATION ........................................................ 77

CHAPTER 23. OPROFILE ............................................................................. 78
  23.1. USING OPROFILE ........................................................................... 78
  23.2. OPROFILE DOCUMENTATION ........................................................ 80

CHAPTER 24. SYSTEMTAP ....................................................................... 82
  24.1. ADDITIONAL INFORMATION ........................................................ 82

CHAPTER 25. PERFORMANCE COUNTERS FOR LINUX (PCL) TOOLS AND PERF ......................................................... 83
  25.1. PERF TOOL COMMANDS ................................................................ 83
  25.2. USING PERF ................................................................................ 83

APPENDIX A. REVISION HISTORY ................................................................. 86
This document describes the different features and utilities that make Red Hat Enterprise Linux 7 an ideal enterprise platform for application development.
PART I. SETTING UP A DEVELOPMENT WORKSTATION
Red Hat Enterprise Linux 7 supports development of custom applications. To allow developers to do so, the system must be set up with the required tools and utilities. This chapter lists the most common use cases for development and the items to install.
CHAPTER 1. INSTALLING THE OPERATING SYSTEM

Before setting up for specific development needs, the underlying system must be set up.

1. Install Red Hat Enterprise Linux in the **Workstation** variant. Follow the instructions in *Red Hat Enterprise Linux Installation Guide*.

2. While installing, pay attention to **software selection**. Select the **Development and Creative Workstation** system profile and enable installation of Add-ons appropriate for your development needs. The relevant Add-ons are listed in each of the following sections focusing on various types of development.

3. To develop applications that cooperate closely with the Linux kernel such as drivers, enable automatic crash dumping with **kdump** during the installation.

4. After the system itself is installed, register it and attach the required subscriptions. Follow the instructions in *Red Hat Enterprise Linux System Administrator’s Guide*, Chapter Registering the System and Managing Subscriptions.
   The following sections focusing on various types of development list the particular subscriptions that must be attached for the respective type of development.

5. More recent versions of development tools and utilities are available as Red Hat Software Collections. For instructions on accessing Red Hat Software Collections, see *Red Hat Software Collections Release Notes*, Chapter Installation.

**Additional Resources**

- Red Hat Enterprise Linux Installation Guide — *Subscription Manager*
- *Red Hat Subscription Management*
- *Red Hat Enterprise Linux 7 Package Manifest*
CHAPTER 2. SETTING UP TO MANAGE APPLICATION VERSIONS

Effective version control is essential to all multi-developer projects. Red Hat Enterprise Linux is distributed with Git, a distributed version control system.

1. Select the Development Tools Add-on during system installation to install Git.

2. Alternatively, install the git package from the Red Hat Enterprise Linux repositories after the system is installed.

   # yum install git

3. To get the latest version of Git supported by Red Hat, install the rh-git29 component from Red Hat Software Collections.

   # yum install rh-git29

4. Set the full name and email address associated with your Git commits:

   $ git config --global user.name "full name"
   $ git config --global user.email "email_address"

   Replace full name and email_address with your actual name and email address.

5. To change the default text editor started by Git, set value of the core.editor configuration option:

   $ git config --global core.editor command

   Replace command with the command to be used to start the selected text editor.

Additional Resources

- Chapter 11, Using Git
- Red Hat Software Collections 3.0 Release Notes — 4.5 Git
CHAPTER 3. SETTING UP TO DEVELOP APPLICATIONS USING C AND C++

Red Hat Enterprise Linux best supports development using the fully compiled C and C++ programming languages.

1. Select the **Development Tools** and **Debugging Tools** Add-ons during system installation to install the **GNU Compiler Collection (GCC)** and **GNU Debugger (GDB)** as well as other development tools.

2. Latest versions of **GCC**, **GDB** and the associated tools are available as a part of the **Red Hat Developer Toolset** toolchain component.

   ```
   # yum install devtoolset-7-toolchain
   ```

3. The Red Hat Enterprise Linux repositories contain many libraries widely used for development of C and C++ applications. Install the development packages of the libraries needed for your application using the **yum** package manager.

4. For graphical-based development, install the **Eclipse** integrated development environment. The C and C++ languages are directly supported. **Eclipse** is available as part of Red Hat Developer Tools. For the actual installation procedure, see Using Eclipse.

**Additional Resources**

- Red Hat Developer Toolset User Guide — List of Components
CHAPTER 4. SETTING UP TO DEBUG APPLICATIONS

Red Hat Enterprise Linux offers multiple debugging and instrumentation tools to analyze and troubleshoot internal application behavior.

1. Select the **Debugging Tools** and **Desktop Debugging and Performance Tools** Add-ons during system installation to install the **GNU Debugger (GDB)**, **Valgrind**, **SystemTap**, **ltrace**, **strace**, and other tools.

2. For the latest versions of **GDB**, **Valgrind**, **SystemTap**, **strace**, and **ltrace**, install Red Hat Developer Toolset. This installs **memstomp**, too.

   ```bash
   # yum install devtoolset-7
   ```

3. The **memstomp** utility is available only as a part of Red Hat Developer Toolset. In case installing the whole Developer Toolset is not desirable and **memstomp** is required, install only its component from Red Hat Developer Toolset.

   ```bash
   # yum install devtoolset-7-memstomp
   ```

4. Install the **yum-utils** package in order to use the **debuginfo-install** tool:

   ```bash
   # yum install yum-utils
   ```

5. To debug applications and libraries available as part of Red Hat Enterprise Linux, install their respective debuginfo and source packages from the Red Hat Enterprise Linux repositories using the **debuginfo-install** tool. This applies to core dump file analysis, too.

6. Install kernel debuginfo and source packages required by the **SystemTap** application. See **SystemTap Beginners Guide**, section Installing SystemTap.

7. To capture kernel dumps, install and configure **kdump**. Follow the instructions in **Kernel Crash Dump Guide**, Chapter Installing and Configuring kdump.

8. Make sure **SELinux** policies allow the relevant applications to run not only normally, but in the debugging situations, too. See **SELinux User’s and Administrator’s Guide**, section Fixing Problems.

Additional Resources

- Section 20.1, “Enabling Debugging with Debugging Information”
- **SystemTap Beginners Guide**
CHAPTER 5. SETTING UP TO MEASURE PERFORMANCE OF APPLICATIONS

Red Hat Enterprise Linux includes several applications that can help a developer identify the causes of application performance loss.

1. Select the Debugging Tools, Development Tools, and Performance Tools Add-ons during system installation to install the tools OProfile, perf, and pcp.

2. Install the tools SystemTap which allows some types of performance analysis, and Valgrind which includes modules for performance measurement.

   # yum install valgrind systemtap systemtap-runtime

3. Run a SystemTap helper script for setting up the environment.

   # stap-prep

   **NOTE**
   
   Running this script installs very large kernel debuginfo packages.

4. For more frequently updated versions of SystemTap, OProfile, and Valgrind, install the Red Hat Developer Toolset package perftools.

   # yum install devtoolset-7-perftools

Additional Resources

- Red Hat Developer Toolset User Guide — IV. Performance Monitoring Tools
CHAPTER 6. SETTING UP TO DEVELOP APPLICATIONS USING JAVA

Red Hat Enterprise Linux supports development of applications in Java.

1. During system installation, select the **Java Platform** add-on to install OpenJDK as the default Java version. Alternatively, follow the instructions in Red Hat JBoss Developer Studio Installation Guide, Section Installing OpenJDK on Red Hat Enterprise Linux to install OpenJDK separately.

2. For an integrated graphical development environment, install the Eclipse-based Red Hat JBoss Developer Studio offering extensive support for Java development. Follow the instructions in Red Hat JBoss Developer Studio Installation Guide.
CHAPTER 7. SETTING UP TO DEVELOP APPLICATIONS USING PYTHON

The Python language version 2.7.5 is available as a part of Red Hat Enterprise Linux.

1. Newer versions of the Python interpreter and libraries are available as Red Hat Software Collections packages. Install the package with desired version according to the table below.

   # yum install package

<table>
<thead>
<tr>
<th>Python versions corresponding to Red Hat Software Collections packages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Version</strong></td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>Python 2.7.13</td>
</tr>
<tr>
<td>Python 3.4.2</td>
</tr>
<tr>
<td>Python 3.5.1</td>
</tr>
<tr>
<td>Python 3.6.3</td>
</tr>
</tbody>
</table>

2. Install the Eclipse integrated development environment which supports development in the Python language. Eclipse is available as part of Red Hat Developer Tools. For the actual installation procedure, see Using Eclipse.

Additional Resources

- Red Hat Software Collections Hello-World — Python
- Red Hat Software Collections 3.0 Components
CHAPTER 8. SETTING UP TO DEVELOP APPLICATIONS USING C# AND .NET CORE

Red Hat supports development of applications using the C# language targeting the .NET Core runtime environment.

- Install the .NET Core for Red Hat Enterprise Linux, which includes the runtime, compilers and additional tools. Follow the instructions in .NET Core Getting Started Guide, chapter Install .NET Core.

Apart from C#, the .NET Core 2.1 for Red Hat Enterprise Linux supports development in ASP.NET, F# and Visual Basic.

Additional Resources

- .NET Core for Red Hat Enterprise Linux Overview
CHAPTER 9. SETTING UP TO DEVELOP CONTAINERIZED APPLICATIONS

Red Hat supports development of containerized applications based on Red Hat Enterprise Linux, Red Hat OpenShift and a number of other Red Hat products.

- Install Red Hat Container Development Kit (CDK). CDK provides a Red Hat Enterprise Linux virtual machine single-node Red Hat OpenShift cluster. Follow the instructions in the Red Hat Container Development Kit Getting Started Guide, section Installing CDK.

- Additionally, Red Hat Development Suite is a good choice for development of containerized applications in Java, C, and C++. It consists of Red Hat JBoss Developer Studio, OpenJDK, Red Hat Container Development Kit, and other minor components. To install DevSuite, follow the instructions in Red Hat Development Suite Installation Guide.

Additional Resources

- Red Hat JBoss Developer Studio — Getting Started with Container and Cloud-based Development

- Product Documentation for Red Hat Container Development Kit

- Product Documentation for OpenShift Container Platform — Red Hat Customer Portal

- Red Hat Enterprise Linux Atomic Host — Overview of Containers in Red Hat Systems
CHAPTER 10. SETTING UP TO DEVELOP WEB APPLICATIONS

Red Hat Enterprise Linux supports development of web applications and being the platform for their deployment.

The topic of web development is too broad to capture it with a few simple instructions. This section offers only the best supported paths to development of web applications on Red Hat Enterprise Linux.

- To set up your environment for developing traditional web applications, install the Apache web server, PHP runtime, and MariaDB database server and tools.

  # yum install httpd mariadb-server php-mysql php

  Alternatively, more recent versions of these applications are available as components of Red Hat Software Collections.

  # yum install httpd24 rh-mariadb102 rh-php71

- For development of web applications using Ruby on Rails, install the package from Red Hat Software Collections containing the desired version according to the table below.

  # yum install package

Ruby on Rails versions corresponding to Red Hat Software Collections packages

<table>
<thead>
<tr>
<th>Version</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ruby on Rails 4.1.5</td>
<td>rh-ror41</td>
</tr>
<tr>
<td>Ruby on Rails 4.2.6</td>
<td>rh-ror42</td>
</tr>
<tr>
<td>Ruby on Rails 5.0.1</td>
<td>rh-ror50</td>
</tr>
</tbody>
</table>

Additional Resources

- Red Hat Software Collections 3.0 Release Notes — Ruby on Rails
- Red Hat Software Collections — Hello World in Ruby
- Advanced Linux Commands Cheat Sheet (setting up a LAMP stack) — Red Hat Developers Portal Cheat Sheet
CHAPTER 11. USING GIT

Effective revision control is essential to all multi-developer projects. It allows all developers in a team to create, review, revise, and document code in a systematic and orderly manner. Red Hat Enterprise Linux 7.5 is distributed with an open-source revision control system, Git.

A detailed description of Git and its features is beyond the scope of this book. For more information about this revision control system, see the resources listed below.

Installed Documentation

- Linux manual pages for Git and tutorials:

  $ man git
  $ man gittutorial
  $ man gittutorial-2

  Note that many Git commands have their own manual pages.

- Git User’s Manual — HTML documentation for Git is located at /usr/share/doc/git-1.8.3/user-manual.html.

Online Documentation

- Pro Git — The online version of the Pro Git book provides a detailed description of Git, its concepts and its usage.

- Reference — Online version of the Linux manual pages for Git
PART III. MAKING AN APPLICATION AVAILABLE TO USERS

There are multiple ways of making an application available to its users. This guide describes the most common methods:

- Packaging an application as a RPM package
- Packaging an application as a software collection
- Packaging an application as a container
CHAPTER 12. DISTRIBUTION OPTIONS

Red Hat Enterprise Linux offers three methods of distribution for third-party applications.

**RPM Packages**
RPM Packages are the traditional method of distributing and installing software.

- A mature technology with multiple tools and widely disseminated knowledge.
- Applications are installed as part of the system.
- The installation tools greatly assist in resolving dependencies.
- Only one version of a package can be installed, making multiple application version installations difficult.

To create a RPM package, follow the instructions in RPM Packaging Guide, Chapter Packaging Software.

**Software Collections**
A Software Collection is a specially prepared RPM package for an alternative version of an application.

- A packaging method used and supported by Red Hat.
- Built on top of the RPM package mechanism.
- Multiple versions of an application can be installed at once.

For more information, see Red Hat Software Collections Packaging Guide, 1.2 What Are Software Collections?

To create a software collection package, follow the instructions in Red Hat Software Collections Packaging Guide, Chapter Packaging Software Collections.

**Containers**
Docker-formatted containers are a lightweight virtualization method.

- Application can be present in multiple independent versions and instances.
- Can be prepared easily from an RPM package or Software Collection.
- Interaction with the system can be precisely controlled.
- Isolation of the application increases security.
- Containerizing applications or their components enables orchestration of multiple instances.

**Additional Resources**

- Red Hat Software Collections Packaging Guide — 1.2 What Are Software Collections?
CHAPTER 13. CREATING A CONTAINER WITH AN APPLICATION

This section describes creating a docker-formatted container image from a locally built application. Making your application available as a container is advantageous when you wish to use orchestration for deployment. Alternatively, containerizing effectively solves conflicts of dependencies.

Prerequisites

- Understanding containers
- An application built locally from sources

Steps

1. Decide which base image to use.

   **NOTE**
   
   Red Hat recommends starting with a base image that uses Red Hat Enterprise Linux as its foundation. Refer to Base Image in the Red Hat Container Catalog for further information.

2. Create a workspace directory.

3. Prepare your application as a directory containing all of the application’s required files. Place this directory inside the workspace directory.

4. Write a Dockerfile that describes the steps required to create the container. Refer to the Dockerfile Reference for information about how to create a Dockerfile that includes your content, sets default commands to run, and opens necessary ports and other features.

   An example of a minimal Dockerfile that contains the `my-program/` directory:

   ```
   FROM registry.access.redhat.com/rhel7
   USER root
   ADD my-program/ .
   ```

   Place this Dockerfile into the workspace directory.

5. Build a container image from the Dockerfile:

   ```
   # docker build .
   (...)
   Successfully built container-id
   ```

   During this step, note the `container-id` of the newly created container image.

6. Add a tag to the image, to identify the registry where you want the container image to be stored. See Getting Started with Containers — Tagging Images.

   ```
   # docker tag container-id registry:port/name
   ```

   Replace `container-id` with the value shown in the output of the previous step.
Replace \textit{registry} with address of the registry where you want to push the image, \textit{port} with the port of the registry (omit if not needed), and \textit{name} with the name of the image.

For example, if you are running a registry using the \texttt{docker-distribution} service on your local system with an image named \texttt{myimage}, the tag \texttt{localhost:5000/myimage} would make that image ready to put to the registry.

7. Push the image to the registry so it can be pulled from that registry later by someone who wants to use it.

\begin{verbatim}
# docker push registry:port/name
\end{verbatim}

Replace the tag parts with the same values as these used in the previous step.

To run your own Docker registry, see \textit{Getting Started with Containers — Working with Docker registries}

\textbf{Additional Resources}

- OpenShift Container Platform — \textit{Creating Images}
- Red Hat Enterprise Linux Atomic Host — \textit{Recommended Practices for Container Development}
- \texttt{Dockerfile Reference}
- Docker Documentation — \textit{Get Started, Part 2: Containers}
- Red Hat Enterprise Linux Atomic Host — \textit{Getting Started with Containers}
- \texttt{Base Images} — Red Hat Container Catalog listing
CHAPTER 14. CONTAINERIZING AN APPLICATION FROM PACKAGES

For multiple reasons, it may be advantageous to distribute an application packaged in an RPM package as a container, too.

Prerequisites

- Understanding containers
- An application packaged as one or more RPM packages

Steps

To containerize an application from RPM packages, see Getting Started with Containers — Creating Docker images.

Additional Information

- OpenShift Container Platform — Creating Images
- Red Hat Enterprise Linux Atomic Host — Getting Started with Containers
- Product Documentation for Red Hat Enterprise Linux Atomic Host
- Docker Documentation — Get Started, Part 2: Containers
- Docker Documentation — Dockerfile reference
- Base Images — Red Hat Container Catalog listing
PART IV. CREATING C OR C++ APPLICATIONS

Red Hat offers multiple tools for creating applications using the C and C++ languages. This part of the book lists some of the most common development tasks.
CHAPTER 15. BUILDING CODE WITH GCC

This chapter deals with situations where source code must be transformed into executable code.

15.1. RELATIONSHIP BETWEEN CODE FORMS

Prerequisites

- Understanding the concepts of compiling and linking

Possible Code Forms

When using the C and C++ languages, there are three forms of code:

- **Source code** written in the C or C++ language, present as plain text files. The files typically use extensions such as .c, .cc, .cpp, .h, .hpp, .i, .inc. For a complete list of supported extensions and their interpretation, see the gcc manual pages:

  ```
  $ man gcc
  ```

- **Object code**, created by compiling the source code with a compiler. This is an intermediate form. The object code files use the .o extension.

- **Executable code**, created by linking object code with a linker. Linux application executable files do not use any file name extension. Shared object (library) executable files use the .so file name extension.

**NOTE**

Library archive files for static linking also exist. This is a variant of object code and uses the .a file name extension. Static linking is not recommended. See Section 16.2, “Static and dynamic linking”.

Handling of Code Forms in GCC

Producing executable code from source code requires two steps, which require different applications or tools. GCC can be used as an intelligent driver for both compilers and linkers. This allows you to use a single command gcc for any of the required actions. GCC automatically selects the actions required (compiling and linking), as well as their sequence:

1. Source files are compiled to object files.
2. Object files and libraries are linked (including the previously compiled sources).

It is possible to run GCC such that only step 1 happens, only step 2 happens, or both steps 1 and 2 happen. This is determined by the types of inputs and requested type of output(s).

Because larger projects require a build system which usually runs GCC separately for each action, it is better to always consider compilation and linking as two distinct actions, even if GCC can perform both at once.

Additional Resources

- Section 15.2, “Compiling Source Files to Object Code”
15.2. COMPILING SOURCE FILES TO OBJECT CODE

To create object code files from source files and not an executable file immediately, GCC must be
instructed to create only object code files as its output. This action represents the basic operation of the
build process for larger projects.

Prerequisites

- C or C++ source code file(s)
- GCC installed on the system

Steps

1. Change to the directory containing the source code file(s).

2. Run `gcc` with the `-c` option:

   ```
   $ gcc -c source.c another_source.c
   ```

   Object files are created, with their file names reflecting the original source code files: `source.c`
   results in `source.o`.

   **NOTE**
   With C++ source code, replace the `gcc` command with `g++` for convenient
   handling of C++ Standard Library dependencies.

Additional Resources

- Section 15.5, “Hardening Code with GCC”
- Section 15.4, “Code Optimization with GCC”
- Section 15.8, “Example: Building a C Program with GCC”

15.3. ENABLING DEBUGGING OF C AND C++ APPLICATIONS WITH GCC

Because debugging information is large, it is not included in executable files by default. To enable
debugging of your C and C++ applications with it, you must explicitly instruct the compiler to create it.

Enabling Creation of Debugging Information with GCC

To enable creation of debugging information with GCC when compiling and linking code, use the `-g`
option:

```
$ gcc ... -g ...
```

- Optimizations performed by the compiler and linker can result in executable code which is hard
to relate to the original source code: Variables may be optimized out, loops unrolled, operations
merged into the surrounding ones etc. This affects debugging negatively. For improved
debugging experience, consider setting the optimization with the `-Og` option. However, changing the optimization level changes the executable code and may change the actual behaviour so as to remove some bugs.

- The `-fcompare-debug` GCC option tests code compiled by GCC with debug information and without debug information. The test passes if the resulting two binary files are identical. This test ensures that executable code is not affected by any debugging options, which further ensures that there are no hidden bugs in the debug code. Note that using the `-fcompare-debug` option significantly increases compilation time. See the GCC manual page for details about this option.

Additional Resources

- Section 20.1, “Enabling Debugging with Debugging Information”
- Using the GNU Compiler Collection (GCC) — Options for Debugging Your Program
- Debugging with GDB — Debugging Information in Separate Files
- The GCC manual page:

  $ man gcc

### 15.4. CODE OPTIMIZATION WITH GCC

A single program can be transformed into more than one sequence of machine instructions. A more optimal result can be achieved if more resources are allocated for analysis of the code during compilation.

**Code Optimization with GCC**

With GCC, it is possible to set the optimization level using the `-Olevel` option. This option accepts a set of values in place of the `level`.

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Optimize for compilation speed - no code optimization (default)</td>
</tr>
<tr>
<td>1, 2, 3</td>
<td>Increasing optimization effort for code execution speed</td>
</tr>
<tr>
<td>s</td>
<td>Optimize for resulting file size</td>
</tr>
<tr>
<td>fast</td>
<td>Level 3 plus disregard for strict standards compliance to allow for additional optimizations</td>
</tr>
<tr>
<td>g</td>
<td>Optimize for debugging experience</td>
</tr>
</tbody>
</table>

For release builds, the optimization option `-O2` is recommended.

During development, the `-Og` option is more useful for debugging the program or library in some situations. Because some bugs manifest only with certain optimization levels, ensure to test the program or library with the release optimization level.

GCC offers a large number of options to enable individual optimizations. For more information, see the following Additional Resources.
15.5. HARDENING CODE WITH GCC

When the compiler transforms source code to object code, it can add various checks to prevent commonly exploited situations and thus increase security. Choosing the right set of compiler options can help produce more secure programs and libraries, without changes to the source code.

Release Version Options
The following list of options is the recommended minimum for developers targeting Red Hat Enterprise Linux:

```
$ gcc ... -O2 -g -Wall -Wl,-z,now,-z,relro -fstack-protector-strong -D_FORTIFY_SOURCE=2 ...
```

- For programs, add the `-fPIE` and `-pie` Position Independent Executable options.
- For dynamically linked libraries, the mandatory `-fPIC` (Position Independent Code) option indirectly increases security.

Development Options
The following options are recommended to detect security flaws during development. Use these options in conjunction with the options for the release version:

```
$ gcc ... -Walloc-zero -Walloca-larger-than -Wextra -Wformat-security -Wvla-larger-than ...
```

Additional Resources
- Defensive Coding Guide
- Memory Error Detection Using GCC — Red Hat Developers Blog post

15.6. LINKING CODE TO CREATE EXECUTABLE FILES

Linking is the final step when building a C or C++ application. Linking combines all object files and libraries into an executable file.

Prerequisites
- One or more object file(s)
- GCC installed on the system

Steps
1. Change to the directory containing the object code file(s).
2. Run `gcc:`
An executable file named `executable-file` is created from the supplied object files and libraries.

To link additional libraries, add the required options before the list of object files. See *Chapter 16, Using Libraries with GCC*.

**NOTE**

With C++ source code, replace the `gcc` command with `g++` for convenient handling of C++ Standard Library dependencies.

**Additional Resources**

- Section 15.8, “Example: Building a C Program with GCC”
- Chapter 16, *Using Libraries with GCC*

**15.7. C++ COMPATIBILITY OF VARIOUS RED HAT PRODUCTS**

The Red Hat ecosystem includes several versions of the GCC compiler and linker, provided in Red Hat Enterprise Linux and Red Hat Developer Toolset. The C++ ABI compatibility between these is as follows:

- The *system compiler* based on GCC 4.8 and provided directly as part of Red Hat Enterprise Linux 7 supports only compiling and linking the C++98 standard (also known as C++03), and its variant with GNU extensions.

- Any C++98-compliant binaries or libraries built explicitly with options `-std=C++98` or `-std=gnu++98` can be freely mixed, regardless of the version of compiler used.

- Using and mixing the C++11 and C++14 language versions is supported only when using compilers from Red Hat Developer Toolset and only when all C++ objects compiled with the respective flag have been built using the same major version of GCC.

- When linking C++ files built with both Red Hat Developer Toolset and Red Hat Enterprise Linux toolchain, prefer the Red Hat Developer Toolset version of compiler and linker.

- The default setting for compilers in Red Hat Enterprise Linux 6 and 7 and Red Hat Developer Toolset up to 4.1 is `-std=gnu++98`. That is, C++98 with GNU extensions.

- The default setting for compilers in Red Hat Developer Toolset 6, 6.1, 7, and 7.1 is `-std=gnu++14`. That is, C++14 with GNU extensions.

**Additional Resources**

- Application Compatibility GUIDE
- What gcc versions are available in Red Hat Enterprise Linux? — Knowledge base solution
- Red Hat Developer Toolset User Guide — C++ Compatibility

**15.8. EXAMPLE: BUILDING A C PROGRAM WITH GCC**
This example shows the exact steps to build a sample minimal C program.

**Prerequisites**

- Understanding use of GCC

**Steps**

1. Create a directory `hello-c` and change to it:

   ```
   $ mkdir hello-c
   $ cd hello-c
   ```

2. Create file `hello.c` with the following contents:

   ```
   #include <stdio.h>
   
   int main() {
       printf("Hello, World!\n");
       return 0;
   }
   ```

3. Compile the code with GCC:

   ```
   $ gcc -c hello.c
   ```

   The object file `hello.o` is created.

4. Link an executable file `helloworld` from the object file:

   ```
   $ gcc hello.o -o helloworld
   ```

5. Run the resulting executable file:

   ```
   $ ./helloworld
   Hello, World!
   ```

**Additional Resources**

- Section 18.2, "Example: Building a C Program Using a Makefile"

**15.9. EXAMPLE: BUILDING A C++ PROGRAM WITH GCC**

This example shows the exact steps to build a sample minimal C++ program.

**Prerequisites**

- Understanding the use of GCC

- Understanding the difference between `gcc` and `g++`

**Steps**

1. Create a directory `hello-cpp` and change to it:
$ mkdir hello-cpp
$ cd hello-cpp

2. Create file `hello.cpp` with the following contents:

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, World!\n";
    return 0;
}
```

3. Compile the code with `g++`:

```
$ g++ -c hello.cpp
```

The object file `hello.o` is created.

4. Link an executable file `helloworld` from the object file:

```
$ g++ hello.o -o helloworld
```

5. Run the resulting executable file:

```
$ ./helloworld
Hello, World!
```
CHAPTER 16. USING LIBRARIES WITH GCC

This chapter describes using libraries in code.

16.1. LIBRARY NAMING CONVENTIONS

A special file name convention is used for libraries: A library known as foo is expected to exist as file
libfoo.so or libfoo.a. This convention is automatically understood by the linking input options of
GCC, but not by the output options:

- When linking against the library, the library can be specified only by its name foo with the -l
  option as -lfoo:

  $ gcc ... -lfoo ...

- When creating the library, the full file name libfoo.so or libfoo.a must be specified.

Additional Resources

- Section 17.2, "The soname Mechanism"

16.2. STATIC AND DYNAMIC LINKING

Developers have a choice of using static or dynamic linking when building applications with fully
compiled languages. This section lists the differences, particularly in the context using the C and C++
languages on Red Hat Enterprise Linux. To summarize, Red Hat discourages the use of static linking in
applications for Red Hat Enterprise Linux.

Comparison of static and dynamic linking

Static linking makes libraries part of the resulting executable file. Dynamic linking keeps these libraries as
separate files.

Dynamic and static linking can be compared in a number of ways:

Resource use

Static linking results in larger executable files which contain more code. This additional code coming
from libraries cannot be shared across multiple programs on the system, increasing file system usage
and memory usage at run time. Multiple processes running the same statically linked program will still
share the code.

On the other hand, static applications need fewer run-time relocations, leading to reduced startup
time, and require less private resident set size (RSS) memory. Generated code for static linking can
be more efficient than for dynamic linking due to the overhead introduce by position-independent
code (PIC).

Security

Dynamically linked libraries which provide ABI compatibility can be updated without changing the
executable files depending on these libraries. This is especially important for libraries provided by
Red Hat as part of Red Hat Enterprise Linux, where Red Hat provides security updates. Static linking
against any such libraries is strongly discouraged.

Additionally, security measures such as load address randomization cannot be used with a statically
linked executable file. This further reduces security of the resulting application.
Static linking appears to provide executable files independent of the versions of libraries provided by the operating system. However, most libraries depend on other libraries. With static linking, this dependency becomes inflexible and as a result, both forward and backward compatibility is lost. Static linking is guaranteed to work only on the system where the executable file was built.

**WARNING**

Applications linking statically libraries from the GNU C library (**glibc**) still require **glibc** to be present on the system as a dynamic library. Furthermore, the dynamic library variant of **glibc** available at the application’s run time must be a bitwise identical version to that present while linking the application. As a result, static linking is guaranteed to work only on the system where the executable file was built.

**Support coverage**

Most static libraries provided by Red Hat are in the **Optional** channel and not supported by Red Hat.

**Functionality**

Some libraries, notably the GNU C Library (**glibc**), offer reduced functionality when linked statically. For example, when statically linked, **glibc** does not support threads and any form of calls to the **dlopen()** function in the same program.

As a result of the listed disadvantages, static linking should be avoided at all costs, particularly for whole applications and the **glibc** and **libstdc++** libraries.

**NOTE**

The **compat-glibc** package is included with Red Hat Enterprise Linux 7, but it is not a run time package and therefore not required for running anything. It is solely a development package, containing header files and dummy libraries for linking. This allows compiling and linking packages to run in older Red Hat Enterprise Linux versions (using **compat-gcc-\*** against those headers and libraries). For more information on use of this package, run: `rpm -qpi compat-glibc-\*`.

**Reasons for static linking**

Static linking might be a reasonable choice in some cases, such as:

- A library which is not enabled for dynamic linking
- Fully static linking can be required for running code in an empty **chroot** environment or container. However, static linking using the **glibc-static** package is not supported by Red Hat.

**Additional Resources**

- Red Hat Enterprise Linux 7: Application Compatibility GUIDE

**16.3. USING A LIBRARY WITH GCC**
A library is a package of code which can be reused in your program. A C or C++ library consists of two parts:

- The library code
- Header files

**Compiling Code That Uses a Library**

The header files describe the interface of the library: The functions and variables available in the library. Information from the header files is needed for compiling the code.

Typically, header files of a library will be placed in a different directory than your application's code. To tell GCC where the header files are, use the `-I` option:

```
$ gcc ... -Iinclude_path ...
```

Replace `include_path` with the actual path to the header file directory.

The `-I` option can be used multiple times to add multiple directories with header files. When looking for a header file, these directories are searched in the order of their appearance in the `-I` options.

**Linking Code That Uses a Library**

When linking the executable file, both the object code of your application and the binary code of the library must be available. The code for static and dynamic libraries is present in different forms:

- Static libraries are available as archive files. They contain a group of object files. The archive file has an file name extension `.a`.
- Dynamic libraries are available as shared objects. They are a form of an executable file. A shared object has an file name extension `.so`.

To tell GCC where the archives or shared object files of a are, use the `-L` option:

```
$ gcc ... -Llibrary_path -lfoo ...
```

Replace `library_path` with the actual path to the library directory.

The `-L` option can be used multiple times to add multiple directories. When looking for a library, these directories are searched in the order of their `-L` options.

The order of options matters: GCC cannot link against a library `foo` unless it knows the directory with this library. Therefore, use the `-L` options to specify library directories before using the `-l` options for linking against libraries.

**Compiling and Linking Code Which Uses a Library in One Step**

When the situation allows the code to be compiled and linked in one `gcc` command, use the options for both situations mentioned above at once.

**Additional Resources**

- Using the GNU Compiler Collection (GCC) — 3.15 Options for Directory Search
- Using the GNU Compiler Collection (GCC) — 3.14 Options for Linking

**16.4. USING A STATIC LIBRARY WITH GCC**
Static libraries are available as archives containing object files. After linking, they become part of the resulting executable file.

**NOTE**

Red Hat discourages use of static linking for various reasons. See Section 16.2, “Static and dynamic linking”. Use static linking only when necessary, especially against libraries provided by Red Hat.

**Prerequisites**

- GCC installed on your system
- Understanding static and dynamic linking
- A set of source or object files forming a valid program, requiring some static library `foo` and no other libraries
- The `foo` library is available as a file `libfoo.a`, and no file `libfoo.so` is provided for dynamic linking.

**NOTE**

Most libraries which are part of Red Hat Enterprise Linux are supported for dynamic linking only. The steps below only work for libraries which are not enabled for dynamic linking. See Section 16.6, “Using Both Static and Dynamic Libraries with GCC”.

**Steps**

To link a program from source and object files, adding a statically linked library `foo`, which is to be found as a file `libfoo.a`:

1. Change to the directory containing your code.
2. Compile the program source files with headers of the `foo` library:

   ```
   $ gcc ... -I header_path -c ...
   
   Replace `header_path` with a path to a directory containing the header files for the `foo` library.
   ```
3. Link the program with the `foo` library:

   ```
   $ gcc ... -L library_path -l foo ...
   
   Replace `library_path` with a path to a directory containing the file `libfoo.a`.
   ```
4. To run the program later, simply:

   ```
   $ ./program
   ```
16.5. USING A DYNAMIC LIBRARY WITH GCC

Dynamic libraries are available as standalone executable files, required at both linking time and run time. They stay independent of your application’s executable file.

Prerequisites

- GCC installed on the system
- A set of source or object files forming a valid program, requiring some dynamic library `foo` and no other libraries
- The `foo` library available as a file `libfoo.so`

Linking a Program Against a Dynamic Library

To link a program against a dynamic library `foo`:

```bash
$ gcc ... -Llibrary_path -lfoo ...
```

When a program is linked against a dynamic library, the resulting program must always load the library at run time. There are two options for locating the library:

- Using a `rpath` value stored in the executable file itself
- Using the `LD_LIBRARY_PATH` variable at runtime

Using a `rpath` Value Stored in the Executable File

The `rpath` is a special value saved as a part of an executable file when it is being linked. Later, when the program is loaded from its executable file, the runtime linker will use the `rpath` value to locate the library files.

While linking with GCC, to store the path `library_path` as `rpath`:

```bash
$ gcc ... -Llibrary_path -lfoo -Wl,-rpath=library_path ...
```

The path `library_path` must point to a directory containing the file `libfoo.so`.

CAUTION

There is no space after the comma in the `-Wl,-rpath=` option!

To run the program later:

```bash
$ ./program
```

Using the `LD_LIBRARY_PATH` Environment Variable
If no `rpath` is found in the program’s executable file, the runtime linker will use the `LD_LIBRARY_PATH` environment variable. The value of this variable must be changed for each program according to the path where the shared library objects are to be found.

To run the program without `rpath` set, with libraries present in path `library_path`:

```bash
$ export LD_LIBRARY_PATH=library_path:$LD_LIBRARY_PATH
$ ./program
```

Leaving out the `rpath` value offers flexibility, but requires setting the `LD_LIBRARY_PATH` variable every time the program is to run.

### Placing the Library into the Default Directories

The runtime linker configuration specifies a number of directories as a default location of dynamic library files. To use this default behaviour, copy your library to the appropriate directory.

A full description of the dynamic linker behavior is out of scope of this document. For more information, see the following resources:

- Linux manual pages for the dynamic linker:
  ```bash
  $ man ld.so
  ```

- Contents of the `/etc/ld.so.conf` configuration file:
  ```bash
  $ cat /etc/ld.so.conf
  ```

- Report of the libraries recognized by the dynamic linker without additional configuration, which includes the directories:
  ```bash
  $ ldconfig -v
  ```

### 16.6. USING BOTH STATIC AND DYNAMIC LIBRARIES WITH GCC

Sometimes it is required to link some libraries statically and some dynamically. This situation brings some challenges.

#### Prerequisites

- Understanding static and dynamic linking

#### Introduction

gcc recognizes both dynamic and static libraries. When the `-lfoo` option is encountered, gcc will first attempt to locate a shared object (a `.so` file) containing a dynamically linked version of the `foo` library, and then look for the archive file (.a) containing a static version of the library. Thus, the following situations can result from this search:

- Only the shared object is found and gcc links against it dynamically
- Only the archive is found and gcc links against it statically
- Both the shared object and archive are found; gcc selects by default dynamic linking against the shared object
Neither shared object nor archive is found and linking fails

Because of these rules, the best way to select the static or dynamic version of library for linking is having only that version found by gcc. This can be controlled to some extent by using or leaving out directories containing the library versions, when specifying the -Lpath options.

Additionally, because dynamic linking is the default, the only situation where linking must be explicitly specified is when a library with both versions present should be linked statically. There are two possible resolutions:

- Specifying the static libraries by file path instead of the -l option
- Using the -Wl option to pass options to the linker

**Specifying the static libraries by file**

Usually, gcc is instructed to link against a library foo with the -lfoo option. However, it is possible to specify the full path to file libfoo.a containing the library instead:

```
$ gcc ... path/to/libfoo.a ...
```

From the file extension .a, gcc will understand that this is a library to link with the program. However, specifying the full path to the library file is a less flexible method.

**Using the -Wl option**

The gcc option -Wl is a special option for passing options to the underlying linker. Syntax of this option differs from the other gcc options: It is followed by a comma-separated list of linker options, so that the linker options do not get mixed up with space-separated gcc options.

The ld linker used by gcc offers the options -Bstatic and -Bdynamic to specify whether libraries following this option should be linked statically or dynamically, respectively. After passing -Bstatic and a library to the linker, the default dynamic linking behaviour must be restored manually for the following libraries to be linked dynamically with the -Bdynamic option.

To link a program, linking library first statically (libfirst.a) and second dynamically (libsecond.so):

```
$ gcc ... -Wl,-Bstatic -lfirst -Wl,-Bdynamic -lsecond ...
```

**NOTE**

gcc can be configured to use linkers other than the default ld. The -Wl option applies to the gold linker, too.

**Additional Resources**

- Using the GNU Compiler Collection (GCC) — 3.14 Options for Linking
- Documentation for binutils 2.27 — 2.1 Command Line Options
CHAPTER 17. CREATING LIBRARIES WITH GCC

This chapter describes steps for creating libraries and explains the necessary concepts used by the Linux operating system for libraries.

17.1. LIBRARY NAMING CONVENTIONS

A special file name convention is used for libraries: A library known as foo is expected to exist as file libfoo.so or libfoo.a. This convention is automatically understood by the linking input options of GCC, but not by the output options:

- When linking against the library, the library can be specified only by its name foo with the -l option as -lfoo:

  $ gcc ... -lfoo ...

- When creating the library, the full file name libfoo.so or libfoo.a must be specified.

Additional Resources

- Section 17.2, "The soname Mechanism"

17.2. THE SONAME MECHANISM

Dynamically loaded libraries (shared objects) use a mechanism called soname to manage multiple compatible versions of a library.

Prerequisites

- Understanding dynamic linking and libraries
- Understanding the concept of ABI compatibility
- Understanding library naming conventions
- Understanding symbolic links

Problem Introduction

A dynamically loaded library (shared object) exists as an independent executable file. This makes it possible to update the library without updating the applications that depend on it. However, the following problems arise with this concept:

- Identification of the actual version of the library
- Need for multiple versions of the same library present
- Signalling ABI compatibility of each of the multiple versions

The soname Mechanism

To resolve this, Linux uses a mechanism called soname.

A library foo version X,Y is ABI-compatible with other versions with the same value of X in version number. Minor changes preserving compatibility increase the number Y. Major changes that break compatibility increase the number X.
The actual library foo version X.Y exists as a file `libfoo.so.x.y`. Inside the library file, a soname is recorded with value `libfoo.so.x` to signal the compatibility.

When applications are built, the linker looks for the library by searching for the file `libfoo.so`. A symbolic link with this name must exist, pointing to the actual library file. The linker then reads the soname from the library file and records it into the application executable file. Finally, the linker creates the application such that it declares dependency on the library using the soname, not name or file name.

When the runtime dynamic linker links an application before running, it reads the soname from application's executable file. This soname is `libfoo.so.x`. A symbolic link with this name must exist, pointing to the actual library file. This allows loading the library, regardless of the Y component of version, because the soname does not change.

**NOTE**

The Y component of the version number is not limited to just a single number. Additionally, some libraries encode version in their name.

**Reading soname from a File**

To display the soname of a library file `somelibrary`:

```bash
$ objdump -p somelibrary | grep SONAME
```

Replace `somelibrary` with the actual file name of the library you wish to examine.

### 17.3. CREATING DYNAMIC LIBRARIES WITH GCC

Dynamically linked libraries (shared objects) allow resource conservation through code reuse and increased security by easier updates of the library code. This section describes the steps to build and install a dynamic library from source.

**Prerequisites**

- Understanding the soname mechanism
- GCC installed on the system
- Source code for a library

**Steps**

1. Change to the directory with library sources.

2. Compile each source file to an object file with the Position independent code option `-fPIC`:

   ```bash
   $ gcc ... -c -fPIC some_file.c ...
   ```

   The object files have the same file names as the original source code files, but their extension is `.o`.

3. Link the shared library from the object files:

   ```bash
   $ gcc -shared -o libfoo.so.x.y -Wl,-soname,libfoo.so.x some_file.o ...
   ```
The used major version number is X and minor version number Y.

4. Copy the `libfoo.so.x.y` file to an appropriate location, where the system’s dynamic linker can find it. On Red Hat Enterprise Linux, the directory for libraries is `/usr/lib64`:

```bash
# cp libfoo.so.x.y /usr/lib64
```

Note that you need root permissions to manipulate files in this directory.

5. Create the symlink structure for soname mechanism:

```bash
# ln -s libfoo.so.x.y libfoo.so.x
# ln -s libfoo.so.x libfoo.so
```

Additional Resources

- The Linux Documentation Project — Program Library HOWTO — 3. Shared Libraries

17.4. CREATING STATIC LIBRARIES WITH GCC AND AR

Creating libraries for static linking is possible through conversion of object files into a special type of archive file.

**NOTE**

Red Hat discourages use of static linking for security reasons. Use static linking only when necessary, especially against libraries provided by Red Hat. See Section 16.2, “Static and dynamic linking”.

**Prerequisites**

- GCC and binutils installed on the system
- Understanding static and dynamic linking
- Source file(s) with functions to be shared as a library

**Steps**

1. Create intermediate object files with GCC.

```bash
$ gcc -c source_file.c ...
```

Append more source files as required. The resulting object files share the file name but use the `.o` file name extension.

2. Turn the object files into a static library (archive) using the `ar` tool from the `binutils` package.

```bash
$ ar rcs libfoo.a source_file.o ...
```

File `libfoo.a` is created.

3. Use the `nm` command to inspect the resulting archive:
$ nm libfoo.a

4. Copy the static library file to the appropriate directory.

5. When linking against the library, GCC will automatically recognize from the .a file name extension that the library is an archive for static linking.

$ gcc ... -lfoo ...

Additional Resources

- Linux manual page for the ar tool:

  $ man ar
CHAPTER 18. MANAGING MORE CODE WITH MAKE

The GNU make utility, commonly abbreviated `make`, is a tool for controlling the generation of executables from source files. `make` automatically determines which parts of a complex program have changed and need to be recompiled. `make` uses configuration files called Makefiles to control the way programs are built.

18.1. GNU MAKE AND MAKEFILE OVERVIEW

To create a usable form (usually executable files) from the source files of a particular project, perform several necessary steps. Record the actions and their sequence to be able to repeat them later.

Red Hat Enterprise Linux contains GNU `make`, a build system designed for this purpose.

Prerequisites

- Understanding the concepts of compiling and linking

GNU make

GNU `make` reads Makefiles which contain the instructions describing the build process. A Makefile contains multiple rules that describe a way to satisfy a certain condition (target) with a specific action (recipe). Rules can hierarchically depend on another rule.

Running `make` without any options makes it look for a Makefile in the current directory and attempt to reach the default target. The actual Makefile file name can be one of `Makefile`, `makefile`, and `GNUmakefile`. The default target is determined from the Makefile contents.

Makefile Details

Makefiles use a relatively simple syntax for defining variables and rules, which consists of a target and a recipe. The target specifies what is the output if a rule is executed. The lines with recipes must start with the TAB character.

Typically, a Makefile contains rules for compiling source files, a rule for linking the resulting object files, and a target that serves as the entry point at the top of the hierarchy.

Consider the following Makefile for building a C program which consists of a single file, `hello.c`.

```
all: hello

hello: hello.o
    gcc hello.o -o hello

hello.o: hello.c
    gcc -c hello.c -o hello.o
```

This specifies that to reach the target `all`, file `hello` is required. To get `hello`, one needs `hello.o` (linked by `gcc`), which in turn is created from `hello.c` (compiled by `gcc`).

The target `all` is the default target because it is the first target that does not start with a period (`.`). Running `make` without any arguments is then identical to running `make all`, when the current directory contains this Makefile.

Typical Makefile
A more typical Makefile uses variables for generalization of the steps and adds a target "clean" - remove everything but the source files.

```bash
CC=gcc
CFLAGS=-c -Wall
SOURCE=hello.c
OBJ=$(SOURCE:.c=.o)
EXE=hello

all: $(SOURCE) $(EXE)

$(EXE): $(OBJ)
  $(CC) $(OBJ) -o $@

%.o: %.c
  $(CC) $(CFLAGS) $< -o $@

clean:
  rm -rf $(OBJ) $(EXE)
```

Adding more source files to such Makefile requires only adding them to the line where the SOURCE variable is defined.

Additional resources

- GNU make: Introduction — 2 An Introduction to Makefiles
- Chapter 15, Building Code with GCC

**18.2. EXAMPLE: BUILDING A C PROGRAM USING A MAKEFILE**

Build a sample C program using a Makefile by following the steps in the below example.

**Prerequisites**

- Understanding Makefiles and `make`

**Steps**

1. Create a directory `hellomake` and change to this directory:

   ```bash
   $ mkdir hellomake
   $ cd hellomake
   ```

2. Create a file `hello.c` with the following contents:

   ```c
   #include <stdio.h>

   int main(int argc, char *argv[]) {
     printf("Hello, World!\n");
     return 0;
   }
   ```

3. Create a file `Makefile` with the following contents:
CC=gcc
CFLAGS=-c -Wall
SOURCE=hello.c
OBJ=$(SOURCE:.c=.o)
EXE=hello

all: $(SOURCE) $(EXE)

$(EXE): $(OBJ)
    $(CC) $(OBJ) -o @

%.o: %.c
    $(CC) $(CFLAGS) $< -o @$

clean:
    rm -rf $(OBJ) $(EXE)

CAUTION

The Makefile recipe lines must start with the tab character! When copying the text above from
the browser, you may paste spaces instead. Correct this change manually.

4. Run make:

   $ make
   gcc -c -Wall hello.c -o hello.o
   gcc hello.o -o hello

   This creates an executable file hello.

5. Run the executable file hello:

   $ ./hello
   Hello, World!

6. Run the Makefile target clean to remove the created files:

   $ make clean
   rm -rf hello.o hello

Additional Resources

- Section 15.8, “Example: Building a C Program with GCC”
- Section 15.9, “Example: Building a C++ Program with GCC”

18.3. DOCUMENTATION RESOURCES FOR make

For more information about make, see the resources listed below.

Installed Documentation
● Use the **man** and **info** tools to view manual pages and information pages installed on your system:

```
$ man make
$ info make
```

**Online Documentation**

● The [GNU Make Manual](https://www.gnu.org/software/make/manual/make.html) hosted by the Free Software Foundation

CHAPTER 19. USING THE ECLIPSE IDE FOR C AND C++ APPLICATION DEVELOPMENT

Some developers prefer using an IDE instead of an array of command line tools. Red Hat makes available the Eclipse IDE with support for development of C and C++ applications.

Using Eclipse to Develop C and C++ Applications
A detailed description of the Eclipse IDE and its use for developing C and C++ applications is out of scope of this document. Please refer to the resources linked below.

Additional Resources

- Using Eclipse
- Eclipse documentation — C/C++ Development User Guide
PART V. DEBUGGING APPLICATIONS

Debugging applications is a very wide topic. This part provides a developer with the most common techniques for debugging in multiple situations.
CHAPTER 20. DEBUGGING A RUNNING APPLICATION

This chapter will introduce the techniques for debugging an application which can be started as many times as needed, on a machine directly accessible to the developer.

20.1. ENABLING DEBUGGING WITH DEBUGGING INFORMATION

To debug applications and libraries, debugging information is required. The following sections describe how to obtain this information.

20.1.1. Debugging Information

While debugging any executable code, two kinds of information allow the tools and by extension the programmer to comprehend the binary code:

- the source code text
- a description of how the source code text relates to the binary code

This is referred to as debugging information.

Red Hat Enterprise Linux uses the ELF format for executable binaries, shared libraries, or debuginfo files. Within these ELF files, the DWARF format is used to hold the debug information.

DWARF symbols are read by the `readelf -w file` command.

CAUTION

STABS is occasionally used with UNIX. STABS is an older, less capable format. Its use is discouraged by Red Hat. GCC and GDB support STABS production and consumption on a best effort basis only. Some other tools such as Valgrind and elfutils do not support STABS at all.

Additional Resources

- The DWARF Debugging Standard

20.1.2. Enabling Debugging of C and C++ Applications with GCC

Because debugging information is large, it is not included in executable files by default. To enable debugging of your C and C++ applications with it, you must explicitly instruct the compiler to create it.

Enabling Creation of Debugging Information with GCC

To enable creation of debugging information with GCC when compiling and linking code, use the `-g` option:

```
$ gcc ... -g ...
```

- Optimizations performed by the compiler and linker can result in executable code which is hard to relate to the original source code: Variables may be optimized out, loops unrolled, operations merged into the surrounding ones etc. This affects debugging negatively. For improved debugging experience, consider setting the optimization with the `-Og` option. However, changing the optimization level changes the executable code and may change the actual behaviour so as to remove some bugs.
The `-fcompare-debug` GCC option tests code compiled by GCC with debug information and without debug information. The test passes if the resulting two binary files are identical. This test ensures that executable code is not affected by any debugging options, which further ensures that there are no hidden bugs in the debug code. Note that using the `-fcompare-debug` option significantly increases compilation time. See the GCC manual page for details about this option.

Additional Resources

- Section 20.1, "Enabling Debugging with Debugging Information"
- Using the GNU Compiler Collection (GCC) — Options for Debugging Your Program
- Debugging with GDB — Debugging Information in Separate Files
- The GCC manual page:
  
  ```
  $ man gcc
  ```

20.1.3. Debuginfo Packages

Debuginfo packages contain debugging information and debug source code for programs and libraries.

Prerequisites

- Understanding debugging information

Debuginfo Packages

For applications and libraries installed in packages from the Red Hat Enterprise Linux repositories, you can obtain the debugging information and debug source code as separate debuginfo packages available through another channel. The debuginfo packages contain `.debug` files, which contain DWARF debuginfo and the source files used for compiling the binary packages. Debuginfo package contents are installed to the `/usr/lib/debug` directory.

A debuginfo package provides debugging information valid only for a binary package with the same name, version, release and architecture:

- Binary package: `packagename-version-release.architecture.rpm`
- Debuginfo package: `packagename-debuginfo-version-release.architecture.rpm`

20.1.4. Getting debuginfo Packages for an Application or Library using GDB

The GNU Debugger (GDB) automatically recognizes missing debug information and resolves the package name.

Prerequisites

- The application or library you want to debug installed on the system
- `GDB` installed on the system
- The `debuginfo-install` tool installed on the system

Procedure
1. Start GDB attached to the application or library you want to debug. GDB automatically recognizes missing debugging information and suggests a command to run.

   $ gdb -q /bin/ls
   Reading symbols from /usr/bin/ls...Reading symbols from /usr/bin/ls...(no debugging symbols found)...done.
   (no debugging symbols found)...done.
   Missing separate debuginfos, use: debuginfo-install coreutils-8.22-21.el7.x86_64
   (gdb)

2. Exit GDB without proceeding further: type q and Enter.

   (gdb) q

3. Run the command suggested by GDB to install the needed debuginfo packages:

   # debuginfo-install coreutils-8.22-21.el7.x86_64

   Installing a debuginfo package for an application or library installs debuginfo packages for all dependencies, too.

4. In case GDB is not able to suggest the debuginfo package, follow the procedure in Section 20.1.5, “Getting debuginfo Packages for an Application or Library Manually”.

Additional Resources

- Red Hat Developer Toolset User Guide, section Installing Debugging Information

- How can I download or install debuginfo packages for RHEL systems? — Red Hat Knowledgebase solution

20.1.5. Getting debuginfo Packages for an Application or Library Manually

You can determine manually debuginfo packages for installation by locating the executable file and finding the package which installs it.

**NOTE**

Prefer use of GDB to determine the packages for installation. Use this manual procedure only if GDB is not able to suggest the package to install.

Prerequisites

- The application or library must be installed on the system.

- The debuginfo-install tool must be available on the system.

Procedure

1. Find the executable file of the application or library.

   a. Use the **which** command to find the application file.
$ which nautilus
/usr/bin/nautilus

b. Use the **locate** command to find the library file.

```
$ locate libz | grep so
/usr/lib64/libz.so
/usr/lib64/libz.so.1
/usr/lib64/libz.so.1.2.7
```

If the original reasons for debugging included error messages, pick the result where the library has the same additional numbers in its file name. If in doubt, try following the rest of the procedure with the result where the library file name includes no additional numbers.

**NOTE**

The **locate** command is provided by the **mlocate** package. To install it and enable its use:

```
# yum install mlocate
# updatedb
```

2. Using the file path, search for a package which provides that file.

```
# yum provides /usr/lib64/libz.so.1.2.7
Loaded plugins: product-id, search-disabled-repos, subscription-manager
zlib-1.2.7-17.el7.x86_64 : The compression and decompression library
Repo : @anaconda/7.4
Matched from:
Filename : /usr/lib64/libz.so.1.2.7
```

The output provides a list of packages in the format

```
name-version.distribution.platform
```

In this step, only the package **name** is important, because the version shown in **yum** output may not be the actual installed version.

**IMPORTANT**

If this step does not produce any results, it is not possible to determine which package provided the binary file and this procedure fails.

3. Use the **rpm** low-level package management tool to find what package version is installed on the system. Use the package name as an argument:

```
$ rpm -q zlib
zlib-1.2.7-17.el7.x86_64
```

The output provides details for the installed package in the format

```
name-version.distribution.platform
```

4. Install the debuginfo packages using the **debuginfo-install** utility. In the command, use the package name and other details you determined during the previous step:
# debuginfo-install

zlib-1.2.7-17.el7.x86_64

Installing a debuginfo package for an application or library installs debuginfo packages for all dependencies, too.

Additional Resources

- Red Hat Developer Toolset User Guide — 1.5.4. Installing Debugging Information
- How can I download or install debuginfo packages for RHEL systems? — Knowledgebase article

## 20.2. INSPECTING APPLICATION INTERNAL STATE WITH GDB

To find why an application does not work properly, control its execution and examine its internal state with a debugger. This section describes how to use the GNU Debugger (GDB) for this task.

### 20.2.1. GNU Debugger (GDB)

A debugger is a tool that enables control of code execution and inspection of the state of the code. This capability is used to investigate what is happening in a program and why.

Red Hat Enterprise Linux contains the GNU debugger (GDB) which offers this functionality through a command line user interface.

For a graphical frontend to GDB, install the Eclipse integrated development environment. See Using Eclipse.

**GDB Capabilities**

A single GDB session can debug:

- multithreaded and forking programs
- multiple programs at once
- programs on remote machines or in containers with the `gdbserver` utility connected over a TCP/IP network connection

**Debugging Requirements**

To debug any executable code, GDB requires the respective debugging information:

- For programs developed by you, you can create the debugging information while building the code.
- For system programs installed from packages, their respective debuginfo packages must be installed.

### 20.2.2. Attaching GDB to a Process

In order to examine a process, GDB must be attached to the process.

**Prerequisites**

- GDB must be installed on the system

**Starting a Program with GDB**

When the program is not running as a process, start it with GDB:
$ gdb program

Replace *program* with file name or path to the program.

GDB sets up to start execution of the program. You can set up breakpoints and the *gdb* environment before beginning execution of the process with the *run* command.

**Attaching GDB to an Already Running Process**

To attach GDB to a program already running as a process:

1. Find the process id (*pid*) with the *ps* command:

   ```
   $ ps -C program -o pid h
   pid
   
   Replace *program* with file name or path to the program.
   ```

2. Attach GDB to this process:

   ```
   $ gdb -p pid
   
   Replace *pid* with an actual process id number from the *ps* output.
   ```

**Attaching an Already Running GDB to an Already Running Process**

To attach an already running GDB to an already running program:

1. Use the *shell* GDB command to run the *ps* command and find the program’s process id (*pid)*:

   ```
   (gdb) shell ps -C program -o pid h
   pid
   
   Replace *program* with file name or path to the program.
   ```

2. Use the *attach* command to attach GDB to the program:

   ```
   (gdb) attach pid
   
   Replace *pid* by an actual process id number from the *ps* output.
   ```

**NOTE**

In some cases, GDB might not be able to find the respective executable file. Use the *file* command to specify the path:

```
(gdb) file path/to/program
```
Once the **GDB** debugger is attached to a program, you can use a number of commands to control the execution of the program.

**Prerequisites**

- **GDB must be installed on the system**
- You must have the required debugging information available:
  - The program is compiled and built with debugging information, or
  - The relevant debuginfo packages are installed
- **GDB is attached to the program to be debugged**

**GDB Commands to Step Through the Code**

**r (run)**
Start the execution of the program. If **run** is executed with any arguments, those arguments are passed on to the executable as if the program has been started normally. Users normally issue this command after setting breakpoints.

**start**
Start the execution of the program, and stop at the beginning of the program's main function. If **start** is executed with any arguments, those arguments are passed on to the executable as if the program has been started normally.

**c (continue)**
Continue the execution of the program from the current state. The execution of the program will continue until one of the following becomes true:

- A breakpoint is reached
- A specified condition is satisfied
- A signal is received by the program
- An error occurs
- The program terminates

**n (next)**
Continue the execution of the program from the current state, until next line of code in the current source file is reached. The execution of the program will continue until one of the following becomes true:

- A breakpoint is reached
- A specified condition is satisfied
- A signal is received by the program
- An error occurs
- The program terminates
s (step)
The `step` command also halts execution at each sequential line of code in the current source file. However, if the execution is currently stopped at a source line containing a function call, GDB stops the execution after entering the function call (rather than executing it).

until location
Continue the execution until the code location specified by the `location` option is reached.

fini (finish)
Resume the execution of the program and halt when execution returns from a function. The execution of the program will continue until one of the following becomes true:

- A breakpoint is reached
- A specified condition is satisfied
- A signal is received by the program
- An error occurs
- The program terminates

q (quit)
Terminate the execution and exit GDB.

Additional Resources
- Section 20.2.5, “Using GDB Breakpoints to Stop Execution at Defined Code Locations”
- Debugging with GDB — 4.2 Starting your Program
- Debugging with GDB — 5.2 Continuing and Stepping

20.2.4. Showing Program Internal Values with GDB
Displaying the values of a program’s internal variables is important for understanding of what the program is doing. GDB offers multiple commands that you can use to inspect the internal variables. This section describes the most useful of these commands.

Prerequisites
- Understanding the GDB debugger

GDB Commands to Display the Internal State of a Program

p (print)
Display the value of the argument given. Usually, the argument is the name of a variable of any complexity, from a simple single value to a structure. An argument can also be an expression valid in the current language, including the use of program variables and library functions, or functions defined in the program being tested.

It is possible to extend GDB with `pretty-printer` Python or Guile scripts for customized display of data structures (such as classes, structs) using the `print` command.

bt (backtrace)
Display the chain of function calls used to reach the current execution point, or the chain of functions used up until execution was terminated. This is useful for investigating serious bugs (such as segmentation faults) with elusive causes. Adding the `full` option to the `backtrace` command displays local variables, too.

It is possible to extend GDB with `frame filter` Python scripts for customized display of data displayed using the `bt` and `info frame` commands. The term `frame` refers to the data associated with a single function call.

```
info
```

The `info` command is a generic command to provide information about various items. It takes an option specifying the item to describe.

- The `info args` command displays options of the function call that is the currently selected frame.
- The `info locals` command displays local variables in the currently selected frame.

For a list of the possible items, run the command `help info` in a GDB session:

```
(gdb) help info
```

```
l (list)
```

Show the line in the source code where the program stopped. This command is available only when the program execution is stopped. While not strictly a command to show internal state, `list` helps the user understand what changes to the internal state will happen in the next step of the program’s execution.

**Additional Resources**

- The GDB Python API — Red Hat Developers Blog entry
- Debugging with GDB — 10.9 Pretty Printing

**20.2.5. Using GDB Breakpoints to Stop Execution at Defined Code Locations**

In many cases, it is advantageous to let the program execute until a certain line of code is reached.

**Prerequisites**

- Understanding GDB

**Using Breakpoints in GDB**

Breakpoints are markers that tell GDB to stop the execution of a program. Breakpoints are most commonly associated with source code lines: Placing a breakpoint requires specifying the source file and line number.

- To place a breakpoint:
  - Specify the name of the source code `file` and the `line` in that file:
    ```
    (gdb) br file:line
    ```
  - When `file` is not present, name of the source file at the current point of execution is used:
Alternatively, use a function name to put the breakpoint on its start:

```
(gdb) br function_name
```

A program might encounter an error after a certain number of iterations of a task. To specify an additional condition to halt execution:

```
(gdb) br file:line if condition
```

Replace condition with a condition in the C or C++ language. The meaning of file and line is the same as above.

To inspect the status of all breakpoints and watchpoints:

```
(gdb) info br
```

To remove a breakpoint by using its number as displayed in the output of info br:

```
(gdb) delete number
```

To remove a breakpoint at a given location:

```
(gdb) clear file:line
```

### Additional Resources

- Debugging with GDB — 5.1 Breakpoints, Watchpoints, and Catchpoints

### 20.2.6. Using GDB Watchpoints to Stop Execution on Data Access and Changes

In many cases, it is advantageous to let the program execute until certain data changes or is accessed. This section lists the most common

#### Prerequisites

- Understanding GDB

#### Using Watchpoints in GDB

Watchpoints are markers which tell GDB to stop the execution of program. Watchpoints are associated with data: Placing a watchpoint requires specifying an expression describing a variable, multiple variables, or a memory address.

To place a watchpoint for data change (write):

```
(gdb) watch expression
```

Replace expression with an expression that describes what you want to watch. For variables, expression is equal to the name of the variable.

To place a watchpoint for data access (read):

```
(gdb) rwatch expression
```
To place a watchpoint for any data access (both read and write):

```
(gdb) awatch expression
```

To inspect the status of all watchpoints and breakpoints:

```
(gdb) info br
```

To remove a watchpoint:

```
(gdb) delete num
```

Replace the `num` option with the number reported by the `info br` command.

**Additional Resources**

- Debugging with GDB — 5.1.2 Setting Watchpoints

### 20.2.7. Debugging Forking or Threaded Programs with GDB

Some programs use forking or threads to achieve parallel code execution. Debugging multiple simultaneous execution paths requires special considerations.

**Prerequisites**

- Understanding the GDB debugger
- Understanding the concepts of process forking and threads

**Debugging Forked Programs with GDB**

Forking is a situation when a program (parent) creates an independent copy of itself (child). Use the following settings and commands to affect what GDB does when a fork occurs:

- The `follow-fork-mode` setting controls whether GDB follows the parent or the child after the fork.

  ```
  set follow-fork-mode parent
  
  After a fork, debug the parent process. This is the default.
  ```

  ```
  set follow-fork-mode child
  
  After a fork, debug the child process.
  ```

  ```
  show follow-fork-mode
  
  Display the current setting of `follow-fork-mode`.
  ```

- The `set detach-on-fork` setting controls whether the GDB keeps control of the other (not followed) process or leaves it to run.

  ```
  set detach-on-fork on
  
  The process which is not followed (depending on the value of `follow-fork-mode`) is detached and runs independently. This is the default.
  ```

  ```
  set detach-on-fork off
  ```
GDB keeps control of both processes. The process which is followed (depending on the value of `follow-fork-mode`) is debugged as usual, while the other is suspended.

```bash
show detach-on-fork
```

Display the current setting of `detach-on-fork`.

**Debugging Threaded Programs with GDB**

GDB has the ability to debug individual threads, and to manipulate and examine them independently. To make GDB stop only the thread that is examined, use the commands `set non-stop on` and `set target-async on`. You can add these commands to the `.gdbinit` file. After that functionality is turned on, GDB is ready to conduct thread debugging.

GDB uses a concept of *current thread*. By default, commands apply to the current thread only.

```bash
info threads
```

Display a list of threads with their `id` and `gid` numbers, indicating the current thread.

```bash
thread id
```

Set the thread with the specified `id` as the current thread.

```bash
thread apply ids command
```

Apply the command `command` to all threads listed by `ids`. The `ids` option is a space-separated list of thread ids. A special value `all` applies the command to all threads.

```bash
break location thread id if condition
```

Set a breakpoint at a certain `location` with a certain `condition` only for the thread number `id`.

```bash
watch expression thread id
```

Set a watchpoint defined by `expression` only for the thread number `id`.

```bash
command&
```

Execute command `command` and return immediately to the gdb prompt (`gdb`), continuing any code execution in the background.

```bash
interrupt
```

Halt execution in the background.

**Additional Resources**

- Debugging with GDB — 4.10 Debugging Programs with Multiple Threads
- Debugging with GDB — 4.11 Debugging Forks

## 20.3. RECORDING APPLICATION INTERACTIONS

The executable code of applications interacts with the code of the operating system and shared libraries. Recording an activity log of these interactions can provide enough insight into the application’s behavior without debugging the actual application code. Alternatively, analyzing an application’s interactions can help pinpoint the conditions in which a bug manifests.

### 20.3.1. Tools Useful for Recording Application Interactions

Red Hat Enterprise Linux offers multiple tools for analyzing an application’s interactions should be analyzed.

```bash
strace
```
The **strace** tool primarily enables logging of system calls (kernel functions) used by an application.

- The **strace** output is detailed and explains the calls well, because **strace** interprets parameters and results with knowledge of the underlying kernel code. Numbers are turned into the respective constant names, bitwise combined flags expanded to flag list, pointers to character arrays dereferenced to provide the actual string, and more. Support for more recent kernel features may be lacking.

- You can filter the traced calls to reduce the amount of captured data.

- The use of [command]`strace` does not require any particular setup except for setting up the log filter.

- Tracing the application code with **strace** results in significant slowdown of the application’s execution. As a result, **strace** is not suitable for many production deployments. As an alternative, consider using **ltrace** or SystemTap.

- The version of **strace** available in Red Hat Developer Toolset can also perform system call tampering. This capability is useful for debugging.

**ltrace**

The **ltrace** tool enables logging of an application’s user space calls into shared objects (dynamic libraries).

- **ltrace** enables tracing calls to any library.

- You can filter the traced calls to reduce the amount of captured data.

- The use of **ltrace** does not require any particular setup except for setting up the log filter.

- **ltrace** is lightweight and fast, offering an alternative to **strace**: it is possible to trace the respective interfaces in libraries such as **glibc** with **ltrace** instead of tracing kernel functions with **strace**.

- Because **ltrace** does not handle a known set of calls like **strace**, it does not attempt to explain the values passed to library functions. The **ltrace** output contains only raw numbers and pointers. The interpretation of **ltrace** output requires consulting the actual interface declarations of the libraries present in the output.

**SystemTap**

SystemTap is an instrumentation platform for probing running processes and kernel activity on the Linux system. SystemTap uses its own scripting language for programming custom event handlers.

- Compared to using **strace** and **ltrace**, scripting the logging means more work in the initial setup phase. However, the scripting capabilities extend SystemTap’s usefulness beyond just producing logs.

- SystemTap works by creating and inserting a kernel module. The use of SystemTap is efficient and does not create a significant slowdown of the system or application execution on its own.

- SystemTap comes with a set of usage examples.

**GDB**
The GNU Debugger is primarily meant for debugging, not logging. However, some of its features make it useful even in the scenario where an application’s interaction is the primary activity of interest.

- With GDB, it is possible to conveniently combine the capture of an interaction event with immediate debugging of the subsequent execution path.
- GDB is best suited for analyzing response to infrequent or singular events, after the initial identification of problematic situation by other tools. Using GDB in any scenario with frequent events becomes inefficient or even impossible.

Additional Resources

- Red Hat Enterprise Linux SystemTap Beginners Guide
- Red Hat Developer Toolset User Guide

20.3.2. Monitoring an Application’s System Calls with strace

The `strace` tool enables monitoring the system (kernel) calls performed by an application.

Prerequisites

- `strace` installed on the system

Steps

1. Identify the system calls you wish to monitor.
2. If the program you want to monitor is not running, start `strace` and specify the `program`:

   ```
   $ strace -fvttTy -s 256 -e trace=call program
   ```

   Replace `call` with the system calls to be displayed. You can use the `-e trace=call` option multiple times. If left out, `strace` will display all system call types. See the `strace(1)` manual page for more information.

   If the program is already running, find its process id (`pid`) and attach `strace` to it:

   ```
   $ ps -C program (....)
   $ strace -fvttTy -s 256 -e trace=call -ppid
   ```

   If you do not wish to trace any forked processes or threads, leave out the `-f` option.

3. `strace` displays the system calls made by the application and their details.
   In most cases, an application and its libraries make a large number of calls and `strace` output appears immediately, if no filter for system calls is set.

4. `strace` exits when the program exits.
   To terminate the monitoring before the traced program exits, press `ctrl+C`.

   - If `strace` started the program, the program terminates together with `strace`. 
If you attached **strace** to an already running program, the program terminates together with **strace**.

5. Analyze the list of system calls done by the application.

- Problems with resource access or availability are present in the log as calls returning errors.
- Values passed to the system calls and patterns of call sequences provide insight into the causes of the application’s behaviour.
- If the application crashes, the important information is probably at the end of log.
- The output contains a lot of unnecessary information. However, you can construct a more precise filter and repeat the procedure.

**NOTE**

It is advantageous to both see the output and save it to a file. Use the **tee** command to achieve this:

```
$ strace ... |& tee your_log_file.log
```

**Additional Resources**

- The **strace(1)** manual page.
- **How do I use strace to trace system calls made by a command?** — Knowledgebase article
- Red Hat Developer Toolset User Guide — Chapter strace

### 20.3.3. Monitoring Application’s Library Function Calls with ltrace

The **ltrace** tool enables monitoring of the calls done by an application to functions available in libraries (shared objects).

**Prerequisites**

- **ltrace** installed on the system

**Steps**

1. Identify the libraries and functions of interest, if possible.

2. If the program you want to monitor is not running, start **ltrace** and specify **program**:

   ```
   $ ltrace -f -l library -e function program
   ```

   Use the options `-e` and `-l` to filter the output:

   - Supply the function names to be displayed as **function**. The **-e function** option can be used multiple times. If left out, **ltrace** will display calls to all functions.

   - Instead of specifying functions, you can specify whole libraries with the **-l library** option. This option behaves similarly to the **-e function** option.
See the `ltrace(1)` manual page for more information.

If the program is already running, find its process id (`pid`) and attach `ltrace` to it:

```bash
$ ps -C program
(...)
$ ltrace ... -p pid
```

If you do not wish to trace any forked processes or threads, leave out the `-f` option.

3. `ltrace` displays the library calls made by the application.
   In most cases, an application will make a large number of calls and `ltrace` output appears immediately, if no filter is set.

4. `ltrace` exits when the program exits.
   To terminate the monitoring before the traced program exits, press `ctrl+C`.
   - If `ltrace` started the program, the program terminates together with `ltrace`.
   - If you attached `ltrace` to an already running program, the program terminates together with `ltrace`.

5. Analyze the list of library calls done by the application.
   - If the application crashes, the important information is probably at the end of log.
   - The output contains a lot of unnecessary information. However, you can construct a more precise filter and repeat the procedure.

**NOTE**

It is advantageous to both see the output and save it to a file. Use the `tee` command to achieve this:

```bash
$ ltrace ... |& tee your_log_file.log
```

Additional Resources
- The `strace(1)` manual page.
- Red Hat Developer Toolset User Guide — Chapter ltrace

### 20.3.4. Monitoring Application’s System Calls with SystemTap

The SystemTap tool enables registering custom event handlers for kernel events. In comparison with `strace`, it is harder to use but more efficient and enables more complicated processing logic.

**Prerequisites**
- SystemTap installed on the system

**Steps**

1. Create a file `my_script.stp` with the contents:
probe begin
{
    printf("waiting for syscalls of process %d \n", target())
}

probe syscall.*
{
    if (pid() == target())
        printf("%s(%s)\n", name, argstr)
}

probe process.end
{
    if (pid() == target())
        exit()
}

2. Find the process ID (pid) of the process you wish to monitor:

   $ ps -aux

3. Run SystemTap with the script:

   # stap my_script.stp -x pid

   The value of pid is the process id.

   The script is compiled to a kernel module which is then loaded. This introduces a slight delay
   between entering the command and getting the output.

4. When the process performs a system call, the call name and its parameters are printed to the
   terminal.

5. The script exits when the process terminates, or when you press Ctrl+C.

Additional Resources

- SystemTap Beginners Guide
- SystemTap Tapset Reference
- The SystemTap script approximating strace functionality, available as
  /usr/share/systemtap/examples/process/strace.stp

20.3.5. Using GDB to Intercept Application System Calls

GDB enables stopping the execution in various kinds of situations arising during the execution of a
program. To stop the execution when the program performs a system call, use a GDB catchpoint.

Prerequisites

- Understanding GDB breakpoints
- GDB attached to the program

Stopping Program Execution on System Calls with GDB
1. Set the catchpoint:

```
(gdb) catch syscall syscall-name
```

The command `catch syscall` sets a special type of breakpoint that halts execution when a system call is performed by the program.

The `syscall-name` option specifies the name of the call. You can specify multiple catchpoints for various system calls. Leaving out the `syscall-name` option causes GDB to stop on any system call.

2. If the program has not started execution, start it:

```
(gdb) r
```

If the program execution is only halted, resume it:

```
(gdb) c
```

3. GDB halts execution after any specified system call is performed by the program.

Additional Resources

- Section 20.2.4, “Showing Program Internal Values with GDB”
- Section 20.2.3, “Stepping through Program Code with GDB”
- Debugging with GDB — Setting Watchpoints

### 20.3.6. Using GDB to Intercept Handling of Signals by Applications

GDB enables stopping the execution in various kinds of situations arising during the execution of a program. To stop the execution when the program receives a signal from the operating system, use a GDB `catchpoint`.

Prerequisites

- Understanding GDB breakpoints
- GDB attached to the program

Stopping Program Execution on Receiving a Signal with GDB

1. Set the catchpoint:

```
(gdb) catch signal signal-type
```

The command `catch signal` sets a special type of a breakpoint that halts execution when a signal is received by the program. The `signal-type` option specifies the type of the signal. Use the special value `all` to catch all signals.

2. If the program has not started execution, start it:

```
(gdb) r
```
If the program execution is only halted, resume it:

```bash
(gdb) c
```

3. GDB halts execution after the program receives any specified signal.

**Additional Resources**

- Section 20.2.4, “Showing Program Internal Values with GDB”
- Section 20.2.3, “Stepping through Program Code with GDB”
- Debugging With GDB — 5.1.3 Setting Catchpoints
CHAPTER 21. DEBUGGING A CRASHED APPLICATION

Sometimes, it is not possible to debug an application directly. In these situations, you can collect information about the application at the moment of its termination and analyze it afterwards.

21.1. CORE DUMPS

This section describes what a core dump is and how to use it.

Prerequisites

- Understanding debugging information

Description

A core dump is a copy of a part of the application's memory at the moment the application stopped working, stored in the ELF format. It contains all the application's internal variables and stack, which enables inspection of the application's final state. When augmented with the respective executable file and debugging information, it is possible to analyze a core dump file with a debugger in a way similar to analyzing a running program.

The Linux operating system kernel can record core dumps automatically, if this functionality is enabled. Alternatively, you can send a signal to any running application to generate a core dump regardless of its actual state.

WARNING

Some limits might affect the ability to generate a core dump.

21.2. RECORDING APPLICATION CRASHES WITH CORE DUMPS

To record application crashes, set up core dump saving and add information about the system.

Steps

1. Enable core dumps. Edit the file /etc/systemd/systemd.conf and change the line containing DefaultLimitCORE to the following:

   DefaultLimitCORE=Infinity

2. Reboot the system:

   # shutdown -r now

3. Remove the limits for core dump sizes:

   # ulimit -c unlimited

   To reverse this change, run the command with value 0 instead of unlimited.
When an application crashes, a core dump is generated. The default location for core dumps is the application’s working directory at the time of crash.

Create an SOS report to provide additional information about the system:

```
# sosreport
```

This creates a tar archive containing information about your system, such as copies of configuration files.

Transfer the core dump and the SOS report to the computer where the debugging will take place. Transfer the executable file, too, if it is known.

**IMPORTANT**

When the executable file is not known, subsequent analysis of the core file identifies it.

Optional: Remove the core dump and SOS report after transferring them, to free up disk space.

Additional Resources

- [How to enable core file dumps when an application crashes or segmentation faults](#)
- [What is a sosreport and how to create one in Red Hat Enterprise Linux 4.6 and later?](#)

### 21.3. INSPECTING APPLICATION CRASH STATES WITH CORE DUMPS

**Prerequisites**

- Core dump file and sosreport
- GDB and elfutils installed on the system

**Steps**

1. To identify the executable file where the crash occurred, run the `eu-unstrip` command with the core dump file:

   ```
   $ eu-unstrip -n --core=./core.9814
   0x400000+0x207000 281b2209547f780a5639c904c1ed443e564973e@0x400284
   /usr/bin/sleep /usr/lib/debug/bin/sleep.debug [exe]
   0x7fff26fff000+0x1000
   1e2a683b7d87757697e4275d41a6aaec280795e@0x7fff26fff340 . - linux-
   vdo.so.1
   0x35e7e00000+0x3b6000
   3ed9e61c2b7e707ce24481635776a2ad0307d@0x35e7a001d8 /usr/lib64/ld-
   2.14.90.so.6
   0x35e7a00000+0x224000
   3ed9e61c2b7e707ce24481635776a2ad0307d@0x35e7a001d8 /usr/lib64/ld-
   2.14.90.so.6
   ```
The output contains details for each module on a line, separated by spaces. The information is listed in this order:

1. The memory address where the module was mapped
2. The build-id of the module and where in the memory it was found.
3. The module’s executable file name, displayed as - when unknown, or as . when the module has not been loaded from a file.
4. The source of debugging information, displayed as a file name when available, as . when contained in the executable file itself, or as - when not present at all.
5. The shared library name (soname), or [exe] for the main module.

In this example, the important details are the file name /usr/bin/sleep and the build-id 2818b209547f780a5639c904c4ed443e564973e on the line containing the text [exe]. With this information, you can identify the executable file required for analyzing the core dump.

2. Get the executable file that crashed.
   - If possible, copy it from the system where the crash occurred. Use the file name extracted from the core file.
   - Alternatively, use an identical executable file on your system. Each executable file built on Red Hat Enterprise Linux contains a note with an unique build-id value. Determine the build-id of the relevant locally available executable files:

   ```
   $ eu-readelf -n executable_file
   ```

   Use this information to match the executable file on the remote system with your local copy. The build-id of the local file and build-id listed in the core dump must match.

   - Finally, if the application is installed from a RPM package, you can get the executable file from the package. Use the sosreport output to find the exact version of the package required.

3. Get the shared libraries used by the executable file. Use the same steps as for the executable file.

4. If the application is distributed as a package, load the executable file in GDB, to display hints for missing debuginfo packages. For more details, see Section 20.1.4, “Getting debuginfo Packages for an Application or Library using GDB”.

5. To examine the core file in detail, load the executable file and core dump file with GDB:

   ```
   $ gdb -e executable_file -c core_file
   ```

   Further messages about missing files and debugging information help you identify what is missing for the debugging session. Return to the previous step if needed.

   If the application’s debugging information is available as a file instead of as a package, load this file in GDB with the symbol-file command:

   ```
   (gdb) symbol-file program.debug
   ```
Replace \textit{program.debug} with the actual file name.

\textbf{NOTE}

It might not be necessary to install the debugging information for all executable files contained in the core dump. Most of these executable files are libraries used by the application code. These libraries might not directly contribute to the problem you are analyzing, and you do not need to include debugging information for them.

6. Use the GDB commands to inspect the state of the application at the moment it crashed. See Section 20.2, “Inspecting Application Internal State with GDB”.

\textbf{NOTE}

When analyzing a core file, GDB is not attached to a running process. Commands for controlling execution have no effect.

\section*{Additional Resources}

\begin{itemize}
  \item Debugging with GDB — 2.1.1 Choosing Files
  \item Debugging with GDB — 18.1 Commands to Specify Files
  \item Debugging with GDB — 18.3 Debugging Information in Separate Files
\end{itemize}

\section*{21.4. DUMPING PROCESS MEMORY WITH \texttt{Gcore}}

The workflow of core dump debugging enables the analysis of the program’s state offline. In some cases, it is advantageous to use this workflow with a program that is still running, such as when it is hard to access the environment with the process. You can use the \texttt{gcore} command to dump memory of any process while it is still running.

\section*{Prerequisites}

\begin{itemize}
  \item Understanding core dumps
  \item GDB installed on the system
\end{itemize}

\section*{Steps}

To dump a process memory using \texttt{gcore}:

\begin{enumerate}
  \item Find out the process id (\textit{pid}). Use tools such as \texttt{ps}, \texttt{pgrep}, and \texttt{top}:

    \begin{verbatim}
    $ ps -C some-program
    \end{verbatim}

  \item Dump the memory of this process:

    \begin{verbatim}
    $ gcore -o filename pid
    \end{verbatim}

    This creates a file \textit{filename} is created and dumps the process memory in it. While the memory is being dumped, the execution of the process is halted.
\end{enumerate}
3. After the core dump is finished, the process resumes normal execution.

4. Create an SOS report to provide additional information about the system:

```bash
# sosreport
```

This creates a tar archive containing information about your system, such as copies of configuration files.

5. Transfer the program’s executable file, core dump, and the SOS report to the computer where the debugging will take place.

6. Optional: Remove the core dump and SOS report after transferring them, to free up disk space.

**Additional resources**

- [How to obtain a core file without restarting an application?](#) — Knowledgebase article

### 21.5. DUMPING PROTECTED PROCESS MEMORY WITH GDB

You can mark the memory of processes as not to be dumped. This can save resources and ensure additional security when the process memory contains sensitive data: in banking or accounting applications or on whole virtual machines. Both kernel core dumps (kdump) and manual core dumps (gcore, GDB) do not dump memory marked this way.

In some cases, it is necessary to dump the whole contents of the process memory regardless of these protections. This procedure shows how to do this using the GDB debugger.

**Prerequisites**

- Understanding core dumps
- GDB installed on the system
- GDB attached to the process with protected memory

**Steps**

1. Set GDB to ignore the settings in the `/proc/PID/coredump_filter` file:

   ```
   (gdb) set use-coredump-filter off
   ```

2. Set GDB to ignore the memory page flag `VM_DONTDUMP`:

   ```
   (gdb) set dump-excluded-mappings on
   ```

3. Dump the memory:

   ```
   (gdb) gcore core-file
   ```

   Replace `core-file` with name of file where you want to dump the memory.

**Additional Resources**

- [Debugging with GDB - How to Produce a Core File from Your Program](#)
Developers profile programs to focus attention on the areas of the program that have the largest impact on performance. The types of data collected include what section of the program consumes the most processor time, and where memory is allocated. Profiling collects data from the actual program execution. Thus, the quality of the data collected is influenced by the actual tasks being performed by the program. The tasks performed during profiling should be representative of actual use; this ensures that problems arising from realistic use of the program are addressed during development.

Red Hat Enterprise Linux includes a number of different tools (Valgrind, OProfile, perf, and SystemTap) to collect profiling data. Each tool is suitable for performing specific types of profile runs, as described in the following sections.
CHAPTER 22. VALGRIND

Valgrind is an instrumentation framework for building dynamic analysis tools that can be used to profile applications in detail. The default installation already provides five standard tools. Valgrind tools are generally used to investigate memory management and threading problems. The Valgrind suite also includes tools that allow the building of new profiling tools as required.

Valgrind provides instrumentation for user-space binaries to check for errors, such as the use of uninitialized memory, improper allocation/freeing of memory, and improper arguments for system calls. Its profiling tools can be used by normal users on most binaries; however, compared to other profilers, Valgrind profile runs are significantly slower. To profile a binary, Valgrind runs it inside a special virtual machine, which allows Valgrind to intercept all of the binary instructions. Valgrind’s tools are most useful for looking for memory-related issues in user-space programs; it is not suitable for debugging time-specific issues or kernel-space instrumentation and debugging.

Valgrind reports are most useful and accurate when debuginfo packages are installed for the programs or libraries under investigation. See Section 20.1, “Enabling Debugging with Debugging Information”.

22.1. VALGRIND TOOLS

The Valgrind suite is composed of the following tools:

memcheck

This tool detects memory management problems in programs by:

- checking all reads from and writes to memory
- intercepting memory manipulations like calls to malloc, free, new or delete

memcheck is perhaps the most used Valgrind tool, as memory management problems can be difficult to detect using other means. Such problems often remain undetected for long periods, eventually causing crashes that are difficult to diagnose.

cachegrind

cachegrind is a cache profiler that accurately pinpoints sources of cache misses in code by performing a detailed simulation of the L1, D1 and L2 caches in the CPU. It shows the number of cache misses, memory references, and instructions accruing to each line of source code; cachegrind also provides per-function, per-module, and whole-program summaries, and can even show counts for each individual machine instructions.

callgrind

Like cachegrind, callgrind can model cache behavior. However, the main purpose of callgrind is to record callgraphs data for the executed code.

massif

massif is a heap profiler; it measures how much heap memory a program uses, providing information on heap blocks, heap administration overheads, and stack sizes. Heap profilers are useful in finding ways to reduce heap memory usage. On systems that use virtual memory, programs with optimized heap memory usage are less likely to run out of memory, and may be faster as they require less paging.

helgrind

In programs that use the POSIX pthreads threading primitives, helgrind detects synchronization errors. Such errors are:

- Misuses of the POSIX pthreads API
Potential deadlocks arising from lock ordering problems

Data races (that is, accessing memory without adequate locking)

Valgrind also allows you to develop your own profiling tools. In line with this, Valgrind includes the lackey tool, which is a template for generating your own tools.

22.2. USING VALGRIND

The valgrind package and its dependencies install all the necessary tools for performing a Valgrind profile run. To profile a program with Valgrind, use:

```
$ valgrind --tool=toolname program
```

See Section 22.1, “Valgrind Tools” for a list of arguments for `toolname`. In addition to the suite of Valgrind tools, none is also a valid argument for `toolname`; this argument allows you to run a program under Valgrind without performing any profiling. This is useful for debugging or benchmarking Valgrind itself.

You can also instruct Valgrind to send all of its information to a specific file. To do so, use the option `--log-file=filename`. For example, to check the memory usage of the executable file hello and send profile information to output, use:

```
$ valgrind --tool=memcheck --log-file=output hello
```

See Section 22.3, “Additional information” for more information on Valgrind, along with other available documentation on the Valgrind suite of tools.

22.3. ADDITIONAL INFORMATION

For more extensive information on Valgrind, see man valgrind. Red Hat Enterprise Linux also provides a comprehensive Valgrind Documentation book available as PDF and HTML in:

CHAPTER 23. OPROFILE

OProfile is a low overhead, system-wide performance monitoring tool provided by the oprofile package. It uses performance monitoring hardware on the system’s processor to retrieve information about the kernel and executables on the system, such as when memory is referenced, the number of second-level cache requests, and the number of hardware interrupts received. OProfile is also able to profile applications that run in a Java Virtual Machine (JVM).

The following is a selection of the tools provided by OProfile:

ophelp
Displays available events for the system’s processor along with a brief description of each.

operf
The main profiling tool. The operf tool uses the Linux Performance Events subsystem, which allows precise targeting of profiling, and allows OProfile to operate alongside other tools using performance monitoring hardware of your system.
Unlike the previously used opcontrol tool, no initial setup is required, and it can be used without root privileges unless the --system-wide option is used.

ocount
A tool for counting the absolute number of event occurrences. It can count events on the whole system, per process, per CPU, or per thread.

opimport
Converts sample database files from a foreign binary format to the native format for the system. Only use this option when analyzing a sample database from a different architecture.

opannotate
Creates an annotated source for an executable if the application was compiled with debugging symbols.

opreport
Retrieves profile data.

23.1. USING OPROFILE

operf is the recommended tool for collecting profiling data. The tool does not require any initial configuration, and all options are passed to it on the command line. Unlike the legacy opcontrol tool, operf can run without root privileges. See the Using operf chapter in the System Administrator’s Guide for detailed instructions on how to use the operf tool.

Example 23.1. Using ocount

The following example shows counting the amount of events with ocount during execution of the sleep utility:

$ ocount -e INST_RETIRED -- sleep 1
Events were actively counted for 1.0 seconds.
Event counts (actual) for /bin/sleep:
Event Count % time counted
INST_RETIRED 683,011 100.00

Example 23.2. Basic operf usage
In the following example, the operf tool is used to collect profiling data from the ls -l ~ command.

1. Install debugging information for the ls command:
   
   ```bash
   # debuginfo-install -y coreutils
   ```

2. Run the profiling:
   
   ```bash
   $ operf ls -l ~
   Profiling done.
   ```

3. Analyze the collected data:
   
   ```bash
   $ opreport --symbols
   CPU: Intel Skylake microarchitecture, speed 3.4e+06 MHz (estimated)
   Counted cpu_clk_unhalted events (0) with a unit mask of 0x00 (Core cycles when at least one thread on the physical core is not in halt state) count 100000
   samples % image name symbol name
   161 81.3131 no-vmlinux /no-vmlinux
   3 1.5152 libc-2.17.so get_next_seq
   3 1.5152 libc-2.17.so strcoll_l
   2 1.0101 ld-2.17.so _dl_fixup
   2 1.0101 ld-2.17.so _dl_lookup_symbol_x
   [...]
   ```

Example 23.3. Using operf to Profile a Java Program
In the following example, the operf tool is used to collect profiling data from a Java (JIT) program, and the opreport tool is then used to output per-symbol data.

1. Install the demonstration Java program used in this example. It is a part of the java-1.8.0-openjdk-demo package, which is included in the Optional channel. When the Optional channel is enabled, install the package:
   
   ```bash
   # yum install java-1.8.0-openjdk-demo
   ```

2. Install the oprofile-jit package for OProfile to be able to collect profiling data from Java programs:
   
   ```bash
   # yum install oprofile-jit
   ```

3. Create a directory for OProfile data:
4. Change into the directory with the demonstration program:

   $ cd /usr/lib/jvm/java-1.8.0-openjdk/demo/applets/MoleculeViewer/

5. Start the profiling:

   $ operf -d ~/oprofile_data appletviewer \
   -J'-agentpath:/usr/lib64/oprofile/libjvmti_oprofile.so'
   example2.html

6. Change into the home directory and analyze the collected data:

   $ cd

   $ opreport --symbols --threshold 0.5

   A sample output may look like the following:

   $ opreport --symbols --threshold 0.5
   Using /home/rkratky/oprofile_data/samples/ for samples directory.
   CPU: Intel Ivy Bridge microarchitecture, speed 3600 MHz
   (estimated)
   Counted CPU_CLK_UNHALTED events (Clock cycles when not halted)
   with a unit mask of 0x00 (No unit mask) count 100000
   samples %          image name               symbol name
   14270    57.1257  libjvm.so                /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.51-1.b16.el7_1.x86_64/jre/lib/amd64/server/libjvm.so
   3537     14.1593  23719.jo                 Interpreter
   690       2.7622  libc-2.17.so             fgetc
   581       2.3259  libX11.so.6.3.0           /usr/lib64/libX11.so.6.3.0
   364       1.4572  libpthread-2.17.so       pthread_getspecific
   130       0.5204  libfreetype.so.6.10.0    __memset_sse2

23.2. OPROFILE DOCUMENTATION

For more extensive information on OPProfile, see the oprofile(1) manual page. Red Hat Enterprise Linux also provides two comprehensive guides to OPProfile in

file:///usr/share/doc/oprofile-version/:

OPProfile Manual
   A comprehensive manual with detailed instructions on the setup and use of OPProfile is found at file:///usr/share/doc/oprofile-version/oprofile.html

OPProfile Internals
Documentation on the internal workings of OProfile, useful for programmers interested in contributing to the OProfile upstream, can be found at file:///usr/share/doc/oprofile-version/internals.html
CHAPTER 24. SYSTEMTAP

SystemTap is a useful instrumentation platform for probing running processes and kernel activity on the Linux system. To execute a probe:

1. Write SystemTap scripts that specify which system events (for example, virtual file system reads, packet transmissions) should trigger specified actions (for example, print, parse, or otherwise manipulate data).

2. SystemTap translates the script into a C program, which it compiles into a kernel module.

3. SystemTap loads the kernel module to perform the actual probe.

SystemTap scripts are useful for monitoring system operation and diagnosing system issues with minimal intrusion into the normal operation of the system. You can quickly instrument running system test hypotheses without having to recompile and re-install instrumented code. To compile a SystemTap script that probes kernel-space, SystemTap uses information from three different kernel information packages:

- kernel-variant-devel-version
- kernel-variant-debuginfo-version
- kernel-debuginfo-common-arch-version

These kernel information packages must match the kernel to be probed. In addition, to compile SystemTap scripts for multiple kernels, the kernel information packages of each kernel must also be installed.

24.1. ADDITIONAL INFORMATION

For more detailed information about SystemTap, see the following Red Hat documentation:

- SystemTap Beginner’s Guide
- SystemTap Tapset Reference
CHAPTER 25. PERFORMANCE COUNTERS FOR LINUX (PCL) TOOLS AND PERF

Performance Counters for Linux (PCL) is a kernel-based subsystem that provides a framework for collecting and analyzing performance data. Red Hat Enterprise Linux 7 includes this kernel subsystem to collect data and the user-space tool `perf` to analyze the collected performance data. The PCL subsystem can be used to measure hardware events, including retired instructions and processor clock cycles. It can also measure software events, including major page faults and context switches. For example, PCL counters can compute Instructions Per Clock (IPC) from a process's counts of instructions retired and processor clock cycles. A low IPC ratio indicates the code makes poor use of CPU. Other hardware events can also be used to diagnose poor CPU performance.

Performance counters can also be configured to record samples. The relative amount of samples can be used to identify which regions of code have the greatest impact on performance.

25.1. PERF TOOL COMMANDS

Useful `perf` commands include the following:

```
perf stat
```

This `perf` command provides overall statistics for common performance events, including instructions executed and clock cycles consumed. Options allow selection of events other than the default measurement events.

```
perf record
```

This `perf` command records performance data into a file which can be later analyzed using `perf report`.

```
perf report
```

This `perf` command reads the performance data from a file and analyzes the recorded data.

```
perf list
```

This `perf` command lists the events available on a particular machine. These events will vary based on performance monitoring hardware and software configuration of the system.

Use `perf help` to obtain a complete list of `perf` commands. To retrieve `man` page information on each `perf` command, use `perf help command`.

25.2. USING PERF

Using the basic PCL infrastructure for collecting statistics or samples of program execution is relatively straightforward. This section provides simple examples of overall statistics and sampling.

To collect statistics on `make` and its children, use the following command:

```
# perf stat -- make all
```

The `perf` command collects a number of different hardware and software counters. It then prints the following information:

```
Performance counter stats for 'make all':

   244011.782059  task-clock-msecs  #    0.925 CPUs
    53328  context-switches  #    0.000 M/sec
```
The perf tool can also record samples. For example, to record data on the make command and its children, use:

```
# perf record -- make all
```

This prints out the file in which the samples are stored, along with the number of samples collected:

```
[ perf record: Woken up 42 times to write data ]
[ perf record: Captured and wrote 9.753 MB perf.data (~426109 samples) ]
```

### Performance Counters for Linux (PCL) Tools conflict with OProfile

Both OProfile and Performance Counters for Linux (PCL) use the same hardware Performance Monitoring Unit (PMU). If OProfile is running while attempting to use the PCL perf command, an error message like the following occurs when starting OProfile:

```
Error: open_counter returned with 16 (Device or resource busy).
/usr/bin/dmesg may provide additional information.

Fatal: Not all events could be opened.
```

To use the perf command, first shut down OProfile:

```
# opcontrol --deinit
```

You can then analyze perf.data to determine the relative frequency of samples. The report output includes the command, object, and function for the samples. Use perf report to output an analysis of perf.data. For example, the following command produces a report of the executable that consumes the most time:

```
# perf report --sort=comm
```

The resulting output:

```
# Samples: 1083783860000
#
# Overhead        Command
# ........        ...............
# 48.19%          xsltproc
# 44.48%          pdfxmltex
# 6.01%           make
```
0.95% perl
0.17% kernel-doc
0.05% xmllint
0.05% cc1
0.03% cp
0.01% xmlto
0.01% sh
0.01% docproc
0.01% ld
0.01% gcc
0.00% rm
0.00% sed
0.00% git-diff-files
0.00% bash
0.00% git-diff-index

The column on the left shows the relative amount of samples. This output shows that make spends most of this time in xsltproc and pdfxmltex. To reduce the time for make to complete, focus on xsltproc and pdfxmltex. To list functions executed by xsltproc, run:

```bash
# perf report -n --comm=xsltproc
```

This generates:

```
comm: xsltproc
# Samples: 472520675377
#
# Overhead  Samples            Shared Object                  Symbol
# .......... ..........  .............................  ......
#
45.54%215179861044  libxml2.so.2.7.6               [.] xmlXPathCmpNodesExt
11.63%54959620202  libxml2.so.2.7.6               [.] xmlXPathNodeSetAdd__internal_alias
8.60%40634845107  libxml2.so.2.7.6               [.] xmlXPathCompOpEval
4.63%21864091080  libxml2.so.2.7.6               [.] xmlXPathReleaseObject
2.73%12919672281  libxml2.so.2.7.6               [.] xmlXPathSetSort__internal_alias
2.60%12271959697  libxml2.so.2.7.6               [.] valuePop
2.41%11379910918  libxml2.so.2.7.6               [.] valuePop
2.19%10340901937  libxml2.so.2.7.6               [.] valuePush__internal_alias
```
APPENDIX A. REVISION HISTORY

Revision 7-6.1, Tue Oct 30 2018, Vladimír Slávik
   Build for 7.6 GA release.

Revision 7-6, Tue Aug 21 2018, Vladimír Slávik
   Build for 7.6 Beta release.

Revision 7-5.1, Tue Apr 10 2018, Vladimír Slávik
   Build for the 7.5 GA release.

Revision 7-5, Tue Jan 9 2018, Vladimír Slávik
   Published preview of new book version for 7.5 Beta.

Revision 7-4.1, Tue Aug 22 2017, Vladimír Slávik
   Update for new releases of linked products.

Revision 7-4, Wed Jul 26 2017, Vladimír Slávik
   Build for 7.4 GA release. New chapter about setting up a workstation for development.

Revision 1-12, Fri May 26 2017, Vladimír Slávik
   Update to remove outdated information.

Revision 7-3.9, Mon May 15 2017, Robert Krátký
   Build for 7.4 Beta release.