Red Hat Enterprise Linux 7

開発者ガイド

RHEL 7 のアプリケーション開発ツールの概要

Last Updated: 2022-02-01

Red Hat Enterprise Linux 7 開発者ガイド

RHEL 7 のアプリケーション開発ツールの概要

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

法律上の通知

概要

本書は、アプリケーション開発に最適なエンタープライズプラットフォームとして、Red Hat Enterprise Linux 7 を活用するさまざまな機能とユーティリティーを説明します。

目次

前書き

本書は、アプリケーション開発に最適なエンタープライズプラットフォームとして、Red Hat Enterprise Linux 7 を活用するためのさまざまな機能とユーティリティーについて説明します。

Red Hat Enterprise Linux 7 は、RHEL 7.9 のリリース後には更新が停止されます。RHEL 7 製品の現在のステータスについては、「Red Hat Enterprise Linux 7 の Red Hat Software Collections の製品ライフサイクル」を確認してください。

パート I. 開発ワークステーションの設定

Red Hat Enterprise Linux 7 は、カスタムアプリケーションの開発をサポートします。開発者がカスタムアプリケーションを開発できるように、必要なツールやユーティリティーを使用して、システムを設定する必要があります。本章では、開発に関する最も一般的なユースケース、インストールする項目について紹介します。

第1章 オペレーティングシステムのインストール

特定の開発ニーズを設定する前に基盤のシステムを設定する必要があります。

1. **ワークステーション**などに Red Hat Enterprise Linux をインストールします。『Red Hat Enterprise Linux インストールガイド』の手順に従います。

2. インストール時には「ソフトウェアの選択」の内容に注意してください。**Development and Creative Workstation** のシステムプロファイルを選択して、開発のニーズに適したアドオンをインストールできるようにします。以下のセクションに適切なアドオンを記載します。 セクションごとに、さまざまな種類の開発にフォーカスしています。

3. ドライバーなど、 Linux カーネルに密に連携するアプリケーションを開発する場合は、インストール時に、**kdump** で自動クラッシュダンプを有効化してください。

4. システムをインストールしたら、システムを登録し、必要なサブスクリプションを割り当てます。Red Hat Enterprise Linux システム管理ガイドの「第 7 章システム登録およびサブスクリプション管理」を参照してください。
以下のセクションでは、該当する開発種別に合わせて、アタッチする必要がある特定のサブスクリプションを記載します。

5. 最新版の開発ツールやユーティリティーは、Red Hat Software Collections として入手できます。Red Hat Software Collections へのアクセス方法は、『Red Hat Software Collections リリースノート』の「第 2 章 インストール」を参照してください。

関連資料

- Red Hat Enterprise Linux インストールガイド - サブスクリプションマネージャー
- Red Hat Subscription Management
- Red Hat Enterprise Linux 7 パッケージマニフェスト

第2章 アプリケーションのバージョンを管理するための設定

複数の開発者が関わるプロジェクトではすべて、効果的な訂管理が必須になります。Red Hat Enterprise Linux は **Git** という分散型のバージョン管理システムが同梱され、配信されています。

1. インストール時に **開発ツール** アドオンを選択して、**Git** をインストールします。
2. または、システムのインストール後に、Red Hat Enterprise Linux のリポジトリーから **git** パッケージをインストールします。

```
# yum install git
```

3. Red Hat がサポートする **Git** の最新版を取得するには、Red Hat Software Collections から **rh-git227** のコンポーネントをインストールします。

```
# yum install rh-git227
```

4. **Git** コミットに関連付けるフルネームとメールアドレスを設定します。

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

full name と **email_address** は、実際の名前とメールアドレスに置き換えます。

5. **Git** で開始するデフォルトのテキストエディターを変更するには、**core.editor** の設定オプションの値を設定します。

```
$ git config --global core.editor command
```

command は、選択のテキストエディターの起動に使用するコマンドに置き換えます。

関連情報

- *11章Git の使用*

第3章 C および C++ を使用してアプリケーションを開発するための設定

Red Hat Enterprise Linux は、完全にコンパイルされた C および C++ のプログラミング言語を使用すると、最適に開発をサポートできます。

1. インストール時に **開発ツール** および **デバッグツール** のアドオンを選択して、**GNU コンパイラーコレクション (GCC)** と **GNU デバッガー (GDB)** などの開発ツールをインストールします。

2. **GCC**、**GDB** および関連のツールの最新版は、Red Hat Developer Toolset ツールチェーンのコンポーネントとして入手できます。

   ```
   # yum install devtoolset-9-toolchain
   ```

   注記:Red Hat Developer Toolset は、Software Collection として同梱されています。**scl** ユーティリティーでこのツールが使用できるようになり、Red Hat Enterprise Linux システムに含まれる同等のものより、 使用する Red Hat Developer Toolset バイナリーを優先して、コマンドを実行します。

3. Red Hat Enterprise Linux リポジトリーには、C および C++ アプリケーションの開発に幅広く使用されるライブラリーが多数含まれます。**yum** パッケージマネージャーを使用して、アプリケーションに必要なライブラリーの開発パッケージをインストールします。

4. グラフィカルインターフェースベースの開発の場合は、**Eclipse** 統合の開発環境をインストールします。C および C++ 言語は直接サポートされます。**Eclipse** は Red Hat Developer Tools の一部として入手できます。実際のインストール手順については、『Using Eclipse』 を参照してください。

関連資料

- Red Hat Developer Toolset ユーザーガイド: 「第 1 章 Red Hat Developer Toolset コンポーネント」

第4章 アプリケーションをデバッグするための設定

Red Hat Enterprise Linux では、内部のアプリケーションの動作を分析してトラブルシューティングを行うためのデバッグおよびインストルメンテーションツールが複数提供されています。

1. システムのインストール時に、**デバッグツール** および **Desktop Debugging and Performance Tools** アドオンを選択して、**GNU Debugger (GDB)**、**Valgrind**、**SystemTap**、**ltrace**、**strace** などのツールをインストールします。

2. **GDB**、**Valgrind**、**SystemTap**、**strace** および **ltrace** の最新版については、Red Hat Developer Toolset をインストールしてください。これをインストールすると、**memstomp** もインストールされます。

   ```
   # yum install devtoolset-9
   ```

   注記:Red Hat Developer Toolset は、Software Collection として同梱されています。**scl** ユーティリティーでこのツールが使用できるようになり、Red Hat Enterprise Linux システムに含まれる同等のものより、 使用する Red Hat Developer Toolset バイナリーを優先して、コマンドを実行します。

3. **memstomp** ユーティリティーは、Red Hat Developer Toolset の一部としてのみ入手可能です。Developer Toolset 全体のインストールは必要ないが、**memstomp** が必要な場合は、Red Hat Developer Toolset からこのコンポーネントだけをインストールしてください。

   ```
   # yum install devtoolset-9-memstomp
   ```

4. **debuginfo-install** ツールを使用するには、**yum-utils** パッケージをインストールします。

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

5. Red Hat Enterprise Linux の一部として利用可能なアプリケーションやライブラリーをデバッグするには、**debuginfo-install** ツールを使用して、適切な debuginfo およびソースパッケージを Red Hat Enterprise Linux リポジトリーからインストールします。コアのダンプファイル分析についても、これは該当します。

6. **SystemTap** アプリケーションで必要な kernel debuginfo およびソースパッケージをインストールします。『SystemTap ビギナーズガイド』の 2.1.1 章「SystemTap のインストール」 を参照してください。

7. カーネルダンプを取得するには、**kdump** をインストールして設定します。『カーネルクラッシュダンプガイド』の 7.2 章「kdump のインストールと設定」を参照してください。

8. **SELinux** ポリシーで、関連するアプリケーションを正常に実行できるだけでなく、デバッグ状況でも実行できるようになっていることを確認してください。『SELinux ユーザーおよび管理者のガイド』の「第 11.3 章問題の修正」を参照してください。

関連情報

- 「デバッグ情報を使用したデバッグの有効化」
- 『SystemTap ビギナーズガイド』

第5章 アプリケーションのパフォーマンスを測定するための設定

Red Hat Enterprise Linux には、開発者がアプリケーションのパフォーマンス低下の原因を特定できるように支援するアプリケーションが同梱されています。

1. インストール時に **デバッグツール**、**開発ツール** および **パフォーマンスツール** のアドオンを選択して、**OProfile**, **perf** および **pcp** のツールをインストールします。

2. 一部の種類のパフォーマンス分析を可能にする **SystemTap** ツールと、パフォーマンス測定のモジュールを含む **Valgrind** をインストールします。

```
# yum install valgrind systemtap systemtap-runtime
```

   注記:Red Hat Developer Toolset は、Software Collection として同梱されています。**scl** ユーティリティーでこのツールが使用できるようになり、Red Hat Enterprise Linux システムに含まれる同等のものより、 使用する Red Hat Developer Toolset バイナリーを優先して、コマンドを実行します。

3. SystemTap 環境設定用の **SystemTap** ヘルパースクリプトを実行します。

```
# stap-prep
```

注記

このスクリプトを実行すると、サイズが非常に大きいカーネル debuginfo パッケージがインストールされます。

4. より頻繁にバージョンが更新される **SystemTap**、**OProfile** および **Valgrind** については、Red Hat Developer Toolset package **perftools** をインストールしてください。

```
# yum install devtoolset-9-perftools
```

関連資料

- 『Red Hat Developer Toolset ユーザーガイド』: 「パート IV., パフォーマンス監視ツール」

第6章 JAVA を使用してアプリケーションを開発するための設定

Red Hat Enterprise Linux は Java でのアプリケーションの開発をサポートします。

1. システムのインストール時に、**Java Platform** アドオンを選択して、デフォルトの Java バージョンとして OpenJDK をインストールします。
   または、『Installation Guide for Red Hat CodeReady Studio』の「2.2 章 Installing OpenJDK 1.8.0 on RHEL」の説明に従い、OpenJDK を個別にインストールします。

2. 統合型のグラフィカル開発環境の場合は、Java 開発を幅広くサポートする Eclipse ベースの Red Hat CodeReady をインストールしてください。『Installation Guide for Red Hat CodeReady Studio』の手順に従います。

第7章 PYTHON を使用してアプリケーションを開発するための設定

Python 言語バージョン 2.7.5 は Red Hat Enterprise Linux の一部として提供されています。

- **Python** インタープリターおよびライブラリーの新規バージョン (Python 2.7 の新しいバージョンを含む) は、Red Hat Software Collections パッケージとして入手できます。以下の表を参照して必要なバージョンのパッケージをインストールしてください。

  ```
  # yum install package
  ```

  Red Hat Software Collections パッケージに対応する **Python** バージョン

バージョン	パッケージ
Python 2.7	**python27**
Python 3.6	**rh-python36**
Python 3.8	**rh-python38**

  python27 ソフトウェアコレクションは、RHEL 7 の **Python 2** パッケージの更新版です。

2. **Python** 言語での開発をサポートする **Eclipse** 統合型開発環境をインストールします。**Eclipse** は Red Hat Developer Tools の一部として入手できます。実際のインストール手順については、『Using Eclipse』を参照してください。

関連資料

- Red Hat Software Collections Hello-World: 「Python」
- Red Hat Software Collections

第8章 C# および .NET CORE を使用してアプリケーションを開発するための設定

Red Hat は、.NET Core を対象とするアプリケーションの開発をサポートします。

- ランタイム、コンパイラー、追加のツールを含む **.NET Core for Red Hat Enterprise Linux** をインストールします。『Getting Started Guide for .NET Core』の説明に従ってください。

関連資料

- 「.NET Core for Red Hat Enterprise Linux Overview」
- .NET Core for Red Hat Enterprise Linux ドキュメント

第9章 コンテナーアプリケーションの開発のための設定

Red Hat は Red Hat Enterprise Linux、Red Hat OpenShift などの Red Hat 製品をベースとするコンテナーアプリケーションの開発をサポートします。

- **Red Hat Container Development Kit**(CDK) には、単一ノードの Red Hat OpenShift 3 クラスターを実行する Red Hat Enterprise Linux 仮想マシンが同梱されます。OpenShift 4 はサポート対象ではありません。『Red Hat Container Development Kit Getting Started Guide』の「1.4 章 Installing CDK」を参照してください。

- **Red Hat CodeReady Containers** (CRC) は、ローカルコンピューターに最小限の OpenShift 4 クラスターを提供し、開発およびテスト目的として最小限必要な環境を提供します。CodeReady コンテナーは、主に開発者のデスクトップ上での実行を目的としています。

- **Red Hat Development Suite** は、Java、C、および C++ でのコンテナーアプリケーションの開発に使用する Red Hat ツールを提供します。このスイートには、**Red Hat JBoss Developer Studio**、**OpenJDK**、**Red Hat Container Development Kit**や他のマイナーなコンポーネントが含まれています。**DevSuite** をインストールするには、『Red Hat Development Suite Installation Guide』の説明に従ってください。

- **.NET Core 3.1**は、OpenShift Container Platform バージョン 3.3 以降で実行される高品質のアプリケーションを構築するための汎用開発プラットフォームです。インストールおよび使用方法については、『.NET Core Getting Started Guide』の 第 2 章「Using .NET Core 3.1 on Red Hat OpenShift Container Platform」を参照してください。

関連資料

- Red Hat CodeReady Studio:「Getting Started with Container and Cloud-based Development」

- Red Hat Container Development Kit の製品ドキュメント

- OpenShift Container Platform の製品ドキュメント

- Red Hat Enterprise Linux Atomic Host:「Overview of Containers in Red Hat Systems」

第10章 WEB アプリケーションの開発のための設定

Red Hat Enterprise Linux は、Web アプリケーションの開発をサポートするだけでなく、デプロイメントのプラットフォームとしての役割を果たします。

Web 開発のトピックは多岐にわたるため、単純な説明だけで全体を把握することはできません。本セクションでは、Red Hat Enterprise Linux での Web アプリケーション開発のサポートされているパスのみを対象とします。

- 従来の Web アプリケーションの開発環境を設定するには、**Apache** Web サーバー、**PHP** ランタイム、**MariaDB** データベースサーバーおよびツールをインストールします。

```
# yum install httpd mariadb-server php-mysql php
```

  または、これらのアプリケーションの最新版は、Red Hat Software Collections のコンポーネントとして入手できます。

```
# yum install httpd24 rh-mariadb102 rh-php73
```

関連資料

- Red Hat Software Collections
- Red Hat Developer portal Cheat Sheet: 「Advanced Linux Commands Cheat Sheet (setting up a LAMP stack)」

パート II. アプリケーションでの他の開発者との共同作業

このドキュメントでは、バージョン管理システム Git の簡単な概要を説明します。

第11章 GIT の使用

複数の開発者が関わるすべてのプロジェクトでは、効果的な改訂管理が必須になります。改訂管理を効果的に行うことで、チーム内の開発者全員が組織的かつ規則的な方法でコードの作成、見直し、改訂、記録を行えるようになります。Red Hat Enterprise Linux 7 は、オープンソースのバージョン管理システム **Git** が含まれた状態で配信されています。

Git およびその機能の詳しい説明は、本書の対象外となっています。このバージョン管理システムに関する情報は、以下に記載のリソースを参照してください。

インストールされているドキュメント

- **Git** とチュートリアルの Linux の man ページ:

  ```
  $ man git
  $ man gittutorial
  $ man gittutorial-2
  ```

  Git コマンドの多くには、独自の man ページがあるのでご留意ください。

- **Git ユーザーマニュアル**: **Git** の HTML ドキュメントは **/usr/share/doc/git-1.8.3/user-manual.html** にあります。

オンラインドキュメント

- **Pro Git** ブックのオンライン版では、**Git**、Git のコンセプトと用途が詳細にわたり説明されています (Pro Git Book)。

- **Git** のオンライン版 Linux man ページ (Pro Git Reference Sheet)

パート III. ユーザーへのアプリケーションの公開

ユーザーにアプリケーションを公開する方法は複数存在します。本ガイドでは、最も一般的な方法を説明します。

- アプリケーションを RPM にパッケージ化
- アプリケーションをソフトウェアコレクションにパッケージ化
- アプリケーションをコンテナーにパッケージ化

第12章 配信オプション

Red Hat Enterprise Linux は、サードパーティーのアプリケーションを 3 種類の方法で配信します。

RPM パッケージ

RPM パッケージは、ソフトウェアの配信/インストールを行うための従来の方式です。

- RPM パッケージは広く普及したナレッジや、複数のツールが含まれる成熟した技術です。
- アプリケーションはシステムの一部としてインストールされる
- インストールツールは依存関係の解決を大幅に支援する

注記

パッケージのバージョン 1 つしかインストールできないので、アプリケーションの複数バージョンのインストールが困難である

RPM パッケージを作成する方法は、『RPM Packaging Guide』の「Packaging Software」の説明に従うようにしてください。

Software Collections

Software Collection は、アプリケーションの別バージョン用に特別に用意される RPM パッケージです。

- Software Collection は、Red Hat が使用し、サポートするパッケージメソッド
- RPM パッケージメカニズム上に構築される
- アプリケーションの複数バージョンを一度にインストールできる

詳しい情報は、『Red Hat Software Collections Packaging Guide』の「What Are Software Collections?」を参照してください。

Software Collection パッケージを作成する方法は、『Red Hat Software Collections Packaging Guide』の「Packaging Software Collections」の説明に従ってください。

コンテナー

Docker 形式のコンテナーは、軽量な仮想化を実現する方法の 1 つです。

- アプリケーションは、複数の個別バージョンおよびインスタンスとして存在できる
- RPM パッケージおよび Software Collection から簡単に用意できる
- システムとの対話を正確に制御できる
- アプリケーションの分離によりセキュリティーが強化される
- コンテナーのアプリケーションまたはそのコンポーネントで複数インスタンスのオーケストレーションが可能になる

関連資料

- 『Red Hat Software Collections Packaging Guide』：「What Are Software Collections?」

第13章 アプリケーションでのコンテナーの作成

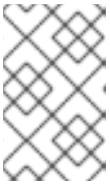
以下のセクションでは、Docker 形式のコンテナーイメージをローカルで構築したアプリケーションから作成する方法について説明します。デプロイメントにオーケストレーションを使用する場合、アプリケーションをコンテナーとして利用可能にすることにはいくつかの利点があります。または、効果的にコンテナー化により、依存関係の衝突が解決されます。

前提条件

- コンテナーを理解していること
- アプリケーションをソースからローカルに構築していること

手順

1. 使用するベースイメージを決定します。

   注記

   Red Hat は、基盤として Red Hat Enterprise Linux を使用するベースイメージの使用から開始することを推奨します。詳細情報は、「Base Image in the Red Hat Container Catalog」を参照してください。

2. ワークスペース用のディレクトリーを作成します。

3. アプリケーションの必要なファイルをすべて含むディレクトリーとして、アプリケーションを用意します。このディレクトリーをワークスペースディレクトリー内に配置します。

4. コンテナーの作成に必要な手順を記述する Dockerfile を作成します。
   コンテンツの追加、デフォルトの実行コマンドの設定、必要なポートの開放および他の機能の追加などの、Dockerfile の作成方法については、「Dockerfile Reference」を参照してください。

   この例では、**my-program/** ディレクトリーを含む最小限の Dockerfile を示しています。

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

   この Dockerfile をワークスペースディレクトリーに配置します。

5. Dockerfile からコンテナーイメージを構築します。

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

   この手順では、新規に作成されたコンテナーイメージの **container-id** をメモするようにしてください。

6. イメージにタグを追加して、コンテナーイメージの保存先のレジストリーを識別します。「Getting Started with Containers: Tagging Images」を参照してください。

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

container-id を、直前の手順の出力に表示された値に置き換えます。

registry を、イメージをプッシュするレジストリーのアドレスに、**port** をレジストリーのポートに (必要に応じて省略)、**name** をイメージの名前に置き換えます。

たとえば、イメージの名前が **myimage** のローカルシステムで **docker-distribution** サービスを使用してレジストリーを実行している場合に、タグ **localhost:5000/myimage** を使用することで、イメージのレジストリーへのプッシュが可能になります。

7. イメージをレジストリーにプッシュし、後でそのレジストリーからプルできるようにします。

```
# docker push registry:port/name
```

タグの部分を、直前の手順で使用した値と同じ値に置き換えます。

独自の Docker レジストリーを実行するには、「Getting Started with Containers – Working with Docker registries」を参照してください。

関連情報

- OpenShift Container Platform - 開発：images
- Red Hat Enterprise Linux Atomic Host: 「コンテナー開発の推奨プラクティス」
- 「Dockerfile Reference」
- Docker ドキュメント - Get Started, Part 2:コンテナー
- Red Hat Enterprise Linux Atomic Host: 『Getting Started with Containers』
- Red Hat Container Catalog listing: 「Base Images」

第14章 パッケージからのアプリケーションのコンテナー化

複数の理由により、RPM のパッケージ化されたアプリケーションをコンテナーとして配信することには利点があります。

前提条件

- コンテナーを理解していること
- アプリケーションが 1 つ以上の RPM パッケージとしてパッケージ化されている

手順

RPM パッケージのアプリケーションをコンテナー化するには、『Getting Started with Containers』の「Creating Docker images」を参照してください。

追加情報

- OpenShift Container Platform: 『イメージの作成』
- Red Hat Enterprise Linux Atomic Host: 『Getting Started with Containers』
- Red Hat Enterprise Linux Atomic Host の製品ドキュメント
- Docker ドキュメント - Get Started, Part 2:コンテナー
- Docker ドキュメント: 「Dockerfile reference」
- Red Hat Container Catalog listing: 「Base Images」

パート IV. C または C++ アプリケーションの作成

Red Hat は、C 言語および C++ 言語を使用してアプリケーションを作成するための各種のツールを提供しています。本書の以下の箇所に、最も一般的な開発タスクの一部を記載しています。

第15章 GCC でのビルドコード

本章では、ソースコードを実行可能なコードに変換する必要のある状況を説明します。

15.1. コード形式間の関係

前提条件

- コンパイルとリンクの概念を理解している

使用可能なコード形式

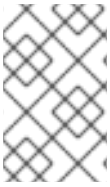
C 言語および C++ 言語を使用する場合は、以下の 3 つのコード形式を使用できます。

- C 言語または C++ 言語で記述された **ソースコード**。プレーンテキストファイルとして表示されます。
  このファイルは通常、**.c**、**.cc**、**.cpp**、**.h**、**.hpp**、**.i**、**.inc** などの拡張子を使用します。サポートされる拡張子およびその解釈の一覧は、gcc の man ページを参照してください。

  ```
  $ man gcc
  ```

- **コンパイラー** でソースコードを **コンパイル**して作成する **オブジェクトコード**。これは中間形式です。
  オブジェクトコードファイルは、拡張子 **.o** を使用します。

- **リンカー** でオブジェクトコードを**リンク** して作成する **実行可能なコード**。
  Linux アプリケーションの実行可能ファイルは、ファイル名の拡張子を使用しません。共有オブジェクト (ライブラリー) の実行可能ファイルは、**.so** のファイル名の拡張子を使用します。



注記

静的リンク用のライブラリーアーカイブファイルも存在します。これはオブジェクトコードのバリアントで、ファイル名拡張子 **.a** を使用します。静的リンクは推奨されません。「静的リンクおよび動的リンク」 を参照してください。

GCC でのコード形式の処理

ソースコードから実行可能なコードを生成するには、2 つの手順を実行してください。各手順では、異なるアプリケーションまたはツールが必要です。GCC は、コンパイラーとリンカーのどちらにも、インテリジェントドライバーとして使用できます。これにより、必要なアクションに **gcc** コマンド 1 つだけで対応できます。GCC は、必要なアクション (コンパイルおよびリンク) とそのシーケンスを自動的に選択します。

1. ソースファイルは、オブジェクトファイルにコンパイルされます。
2. オブジェクトファイルおよびライブラリーはリンクされます (以前にコンパイルしたソールも含む)。

ステップ 1 だけ、ステップ 2 だけ、ステップ 1 と 2 の両方というように、GCC を実行することができます。これは、入力タイプや要求する出力タイプにより決定されます。

大規模なプロジェクトには、アクションごとに個別に GCC を実行するビルドシステムが必要なため、GCC が両方同時に実行できる場合でも 2 つの異なるアクションとしてコンパイルとリンクを実行するように検討する方が有用です。

関連情報

- 「ソースファイルのオブジェクトコードへのコンパイル」
- 「実行可能ファイルを作成するためのコードのリンク」

15.2. ソースファイルのオブジェクトコードへのコンパイル

オブジェクトコードファイルを、実行可能ファイルから直接作成するのではなく、ソースファイルから作成するには、GCC で、オブジェクトコードファイルのみを出力として作成するように必要があります。このアクションは、大規模なプロジェクトのビルドプロセスの基本操作となります。

前提条件

- C または C++ のソースコードファイルがある
- GCC がシステムにインストールされている

手順

1. ソースコードファイルが含まれるディレクトリーに移動します。
2. **-c** オプションを指定して **gcc** を実行します。

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

オブジェクトファイルは、オリジナルのソースコードファイルを反映したファイル名を使用して作成されます。**source.c** は **source.o** になります。

注記

C++ ソースコードの場合は、標準 C++ ライブラリーの依存関係を処理しやすくするために、**gcc** コマンドを **g++** に置き換えます。

関連情報

- 「GCC でのコードのハード化」
- 「GCC でのコードの最適化」
- 「例:GCC での C プログラムの構築」

15.3. GCC を使用した C および C++ アプリケーションのデバッグの有効化

デバッグ情報のサイズが大きい場合、その情報はデフォルトで実行可能ファイルに含まれません。GCC を使用した C および C++ のアプリケーションのデバッグを有効にするには、コンパイラーに対して、デバッグ情報を作成するように、明示的に指示する必要があります。

GCC を使用したデバッグ情報の作成の有効化

コードのコンパイルおよびリンク時に **GCC** でデバッグ情報の作成を有効にするには、**-a** オプションを使用します。

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

- コンパイラーとリンカーで最適化を行うと、実行可能なコードを、元のソースコードと関連付けることが難しくなります。変数の最適化、ループのアンロール、周りの操作へのマージなど

が行われる可能性があります。これにより、デバッグに負の影響が及ぶ可能性があります。デバッグの体験を向上するには、**-Og** オプションを指定して、最適化を設定することを考慮してください。ただし、最適化レベルを変更すると、実行可能なコードが変更され、バグを取り除くための実際の動作が変更される可能性があります。

- **-fcompare-debug** GCC オプションでは、GCC でコンパイルしたコードを、デバッグ情報を使用して (または、デバッグ情報を使用せずに) テストします。このテストでは、出力されたバイナリーファイルの 2 つが同一であれば合格します。このテストを行うことで、実行可能なコードがデバッグオプションによる影響は受けないようにするだけでなく、デバッグコードにバグが含まれないようにします。**-fcompare-debug** オプションを使用するとコンパイルの時間が大幅に伸びることに留意してください。このオプションに関する詳細は、GCC の man ページを参照してください。

関連情報

- 「デバッグ情報を使用したデバッグの有効化」
- GNU コンパイラーコレクション (GCC) の使用: 「3.10 Options for Debugging Your Program」
- GDB でのデバッグ: 18.3 Debugging Information in Separate Files
- GCC の man ページ:

```
$ man gcc
```

15.4. GCC でのコードの最適化

1 つのプログラムは、複数の機械語命令シーケンスに変換できます。コンパイル時にコード分析用のリソースがより多く割り当てられると、最適な結果が得られます。

GCC でのコードの最適化
GCC では、**-Olevel** オプションを使用して最適化レベルを設定できます。このオプションでは、**level** の部分に値のセットを指定できます。

レベル	説明
0	コンピレーション速度の最適化: コードの最適化なし (デフォルト)
1、2、3	コード実行速度を高めるための最適化の作業量増加
s	作成されるファイルサイズの最適化
fast	レベル 3 以上は、厳密な標準への準拠を無視して追加の最適化を可能にします
g	デバッグ作業の最適化

リリースビルドの場合の最適化オプションとして、**-O2** を推奨します。

開発中は、場合によってはプログラムやライブラリーのデバッグに **-Og** オプションを使用する方が便利です。バグによっては、特定の最適化レベルでのみ出現するので、リリースの最適化レベルでプログラムまたはライブラリーをテストするようにしてください。

GCC では、個別の最適化を有効にするオプションが多数含まれています。詳細情報は、以下の追加リソースを参照してください。

関連資料

- GNU コンパイラーコレクションの使用: 3.11 Options That Control Optimization
- GCC の Linux man ページ:

```
$ man gcc
```

15.5. GCC でのコードのハード化

コンパイラーでソースコードをオブジェクトコードに変換する場合には、さまざまなチェックを追加して、一般的に悪用される状況などを回避し、セキュリティーを強化することができます。適切なコンパイラーオプションセットを選択して、ソースコードを変更せずに、よりセキュアなプログラムやライブラリーを生成することができます。

リリースバージョンのオプション
Red Hat Enterprise Linux を使用する開発者には、以下のオプション一覧が推奨される最小限のオプションとなります。

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

- プログラムには、**-fPIE** および **-pie** の位置独立実行形式オプションを追加します。
- 動的にリンクされたライブラリーには、必須の **-fPIC** (位置独立コード) オプションを使用すると間接的にセキュリティーが強化されます。

開発オプション
開発時にセキュリティーの欠陥を検出する場合には、以下のオプションを推奨します。これらのオプションは、リリースバージョンのオプションと併せて使用してください。

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

関連資料

- Defensive Coding Guide
- Red Hat 開発者のブログ投稿: Memory Error Detection Using GCC

15.6. 実行可能ファイルを作成するためのコードのリンク

C または C++ のアプリケーション構築の最後の手順は、リンクです。リンクにより、オブジェクトファイルやライブラリーがすべて実行可能ファイルに統合されます。

前提条件

- オブジェクトファイルが 1 つまたは複数ある。
- GCC がシステムにインストールされている

手順

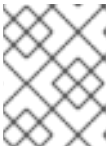
1. オブジェクトコードファイルを含むディレクトリーに移動します。

2. **gcc** を実行します。

```
$ gcc ... objectfile.o another_object.o ... -o executable-file
```

executable-file という名前の実行可能ファイルが、指定したオブジェクトファイルとライブラリーをベースに作成されます。

追加のライブラリーをリンクするには、オブジェクトファイルの一覧の前に必要なオプションを追加します。16章GCC *でのライブラリーの使用* を参照してください。



注記

C++ ソースコードの場合は、標準 C++ ライブラリーの依存関係を処理しやすくするために、**gcc** コマンドを **g++** に置き換えます。

関連情報

- 「例:GCC での C プログラムの構築」
- 16章GCC *でのライブラリーの使用*

15.7. 各種 RED HAT 製品との C++ の互換性

Red Hat エコシステムには、Red Hat Enterprise Linux および Red Hat Developer Toolset で提供される、GCC コンパイラーおよびリンカーのバージョンが複数含まれます。これらのバージョン間の C++ ABI の互換性は以下のとおりです。

- GCC 4.8 がベースになり、Red Hat Enterprise Linux 7 の一部として直接提供されている **システムコンパイラー** は、C++98 標準仕様 (C++03 としても知られています)、およびそのバリアント (GNU 拡張あり) へのコンパイルおよびリンクのみをサポートします。
- **-std=c++98** または **-std=gnu++98** オプションで明示的に構築された C++98 準拠のバイナリーまたはライブラリーは、使用するコンパイラーのバージョンにかかわらず、自由に組み合わせることができます。
- C++11 および C++14 言語のバージョンの使用および併用は、Red Hat Developer Toolset のコンパイラーを使用しており、それぞれのフラグを使用してコンパイルされたすべての C++ オブジェクトが同じメジャーバージョンの GCC を使用して構築されている場合にのみサポートされます。
- Red Hat Developer Toolset および Red Hat Enterprise Linux ツールチェーンで構築された C++ ファイルをリンクする場合には、Red Hat Developer Toolset バージョンのコンパイラーとリンカーが推奨されます。
- Red Hat Enterprise Linux 6 および 7、および Red Hat Developer Toolset (バージョン 4.1 まで) のコンパイラーにおけるデフォルト設定は **-std=gnu++98** です。つまり、C++98 と GNU 拡張です。
- Red Hat Developer Toolset 6、6.1、7、7.1、8.0、8.1、9.0、9.1、10 のコンパイラーのデフォルト設定は **-std=gnu++14** です。つまり、C++14 と GNU 拡張です。

関連資料

- 「Red Hat Enterprise Linux 7: アプリケーションの互換性ガイド」

- ナレッジベースソリューション:「Red Hat Enterprise Linux で利用できる gcc バージョン」
- 『Red Hat Developer Toolset User Guide』:「C++ Compatibility」

15.8. 例:GCC での C プログラムの構築

以下の例では、最小限の C プログラムのサンプルを構築する手順を説明します。

前提条件

- GCC の使用方法を理解していること

手順

1. **hello-c** ディレクトリーを作成して、その場所に移動します。

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

2. 以下の内容で **hello.c** ファイルを作成します。

```
#include <stdio.h>

int main() {
  printf("Hello, World!\n");
  return 0;
}
```

3. GCC でコードをコンパイルします。

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

オブジェクトファイル **hello.o** が作成されます。

4. オブジェクトファイルから作成した実行可能ファイル **helloworld** をリンクします。

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

5. 作成された実行可能ファイルを実行します。

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

関連情報

- 「例:例: Makefile を使用した C プログラムの構築」

15.9. 例:GCC での C++ プログラムの構築

以下の例では、最小限の C++ プログラムのサンプルを構築する手順を説明します。

前提条件

- GCC の使用方法を理解していること

- **gcc** と **g++** の相違点を理解していること

手順

1. **hello-cpp** ディレクトリーを作成して、その場所に移動します。

```
$ mkdir hello-cpp
$ cd hello-cpp
```

2. 以下の内容を含む **hello.cpp** ファイルを作成します。

```
#include <iostream>

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

3. **g++** でコードをコンパイルします。

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

オブジェクトファイル **hello.o** が作成されます。

4. オブジェクトファイルから作成した実行可能ファイル **helloworld** をリンクします。

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

5. 作成された実行可能ファイルを実行します。

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

第16章 GCC でのライブラリーの使用

本章では、コードでライブラリーを使用する方法を説明します。

16.1. ライブラリーの命名規則

特別なファイルの命名規則をライブラリーに使用します。**foo** と呼ばれるライブラリーは、**lib foo .so** または lib**foo** . **a** ファイルとして存在する必要があります。この規則は、リンクする **gcc** の入力オプションでは自動的に理解されますが、出力オプションでは理解されません。

- ライブラリーにリンクする場合は、**-lfoo** のように、**-l** オプションと **foo** の名前でしか、ライブラリーを指定することができません。

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

- ライブラリーの作成時には、**libfoo.so**、**libfoo.a** など、完全なファイル名を指定する必要があります。

関連情報

- 「soname のメカニズム」

16.2. 静的リンクおよび動的リンク

開発者は、完全にコンパイルされた言語でアプリケーションを構築する際に、静的リンクまたは動的リンクを使用できます。本セクションでは、Red Hat Enterprise Linux で C 言語および C++ 言語した場合の違い (特にコンテキストについて) を説明します。Red Hat は、Red Hat Enterprise Linux のアプリケーションで静的リンクを使用することは推奨していません。

静的リンクおよび動的リンクの比較
静的リンクは、作成される実行可能ファイルのライブラリーの一部になります。動的リンクは、これらのライブラリーを別々のファイルとして保持します。

動的リンクおよび静的リンクは、いくつかの点で異なります。

リソースの使用

静的リンクにより、より多くのコードが含まれるより大きな実行可能ファイルが生成されます。ライブラリーからのこの追加コードはシステムのプログラム間で共有できないため、ランタイム時にファイルシステムの使用量とメモリーの使用量が増加します。静的にリンクされた同じプログラムを実行している複数のプロセスは依然としてコードを共有します。
一方、静的アプリケーションは、必要なランタイムの再配置も少なくなるため、起動時間が短縮します。また、必要なプライベートの RSS (homeal Set Size) メモリーも少なくなります。静的リンク用に生成されたコードは、PIC (位置独立コード) により発生するオーバーヘッドにより、動的リンクよりも効率が良くなります。

セキュリティー

ABI 互換性を提供する動的にリンクされたライブラリーは、それらのライブラリーに依存する実行可能ファイルを変更せずに更新できます。これは、特に、Red Hat Enterprise Linux の一部として提供され、Red Hat がセキュリティー更新を提供するライブラリーで重要になります。このようなライブラリーには、静的リンクを使用しないことが強く推奨されます。
さらに、ロードアドレスのランダム化などのセキュリティー対策は、静的にリンクされた実行可能ファイルで使用することはできません。これにより、アプリケーションのセキュリティーが低下します。

互換性

静的リンクは、オペレーティングシステムが提供するライブラリーのバージョンに依存しない実行可能ファイルを提供しているように見えます。ただし、ほとんどのライブラリーは他のライブラリーに依存しています。静的リンクを使用すると、依存関係に柔軟性がなくなり、前方互換性と後方互換性が失われます。静的リンクは、実行可能ファイルが構築されたシステムでのみ機能します。

警告

GNU C ライブラリー (**glibc**) からライブラリーを静的にリンクするアプリケーションでは、引き続き **glibc** が動的ライブラリーとしてシステムに存在する必要があります。さらに、アプリケーションのランタイム時に利用できる **glibc** の動的ライブラリーのバリアントは、アプリケーションのリンク時に表示されるものとビット単位で同じバージョンである必要があります。したがって、静的リンクは、実行可能ファイルが構築されたシステムでのみ機能することが保証されます。

サポート範囲

Red Hat が提供するほとんどの静的ライブラリーは **Optional** チャンネルにあり、Red Hat ではサポートされていません。

機能

いくつかのライブラリー (特に GNU C ライブラリー (**glibc**)) は、静的にリンクすると提供する機能が少なくなります。
たとえば、静的にリンクすると、**glibc** はスレッドや、同じプログラム内の **dlopen()** 関数に対する呼び出しの形式をサポートしません。

上述の不利な点により、静的リンクは、特にアプリケーション全体、**glibc** ライブラリー、および **libstdc++** ライブラリーに対しては、使用しないようにする必要があります。

注記

compat-glibc パッケージは Red Hat Enterprise Linux 7 に含まれますが、ランタイムパッケージではないため、何も実行する必要がありません。これは、リンク用のヘッダーファイルとダミーのライブラリーが含まれる開発パッケージでしかありません。これにより、以前の Red Hat Enterprise Linux バージョンで実行するパッケージのコンパイルおよびリンクが可能になります (ヘッダーやライブラリーに **compat-gcc-*** を使用)。このパッケージの使用の詳細については、**rpm -qpi compat-glibc-*** を実行します。

静的リンクを使用する理由

以下のようなケースでは、静的リンクは妥当なオプションとして選択できます。

- 動的リンクが使用できないライブラリー
- 空の **chroot** 環境またはコンテナーでコードを実行するには、完全に静的なリンクが必要です。ただし、**glibc-static** パッケージを使用した静的リンクは、Red Hat ではサポートされません。

関連情報

- Red Hat Enterprise Linux 7:アプリケーションの互換性ガイド

16.3. GCC でのライブラリーの使用

ライブラリーは、プログラムで再利用可能なコードのパッケージです。C または C++ のライブラリーは、以下の 2 つの部分で構成されます。

- ライブラリーコード
- ヘッダーファイル

ライブラリーを使用するコードのコンパイル

ヘッダーファイルは、ライブラリーのインターフェースを記述します。ライブラリーで利用可能な関数および変数。コードをコンパイルする場合に、ヘッダーファイルの情報が必要です。

通常、ライブラリーのヘッダーファイルは、アプリケーションのコードとは別のディレクトリーに配置されます。ヘッダーファイルの場所を GCC に指示するには、**-I** オプションを使用します。

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

include_path は、ヘッダーファイルのディレクトリーのパスに置き換えます。

-I オプションは、複数回使用して、ヘッダーファイルを含むディレクトリーを複数追加できます。ヘッダーファイルを検索する場合は、**-I** オプションで表示順に、これらのディレクトリーが検索されます。

ライブラリーを使用するコードのリンク

実行可能ファイルをリンクする場合には、アプリケーションのオブジェクトコードとライブラリーのバイナリーコードの両方が利用できる状態でなければなりません。静的ライブラリーおよび動的ライブラリーのコードは、形式が異なります。

- 静的なライブラリーは、アーカイブファイルとして利用できます。静的なライブラリーには、一連のオブジェクトファイルが含まれます。アーカイブファイルのファイル名の拡張子は **.a** になります。
- 動的なライブラリーは共有オブジェクトとして利用できます。実行可能ファイルの形式です。共有オブジェクトのファイル名の拡張子は **.so** になります。

ライブラリーのアーカイブファイルまたは共有オブジェクトファイルの場所を GCC に渡すには、**-L** オプションを使用します。

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

library_path は、ライブラリーのディレクトリーのパスに置き換えます。

-l オプションは複数回使用して、ディレクトリーを複数追加できます。ライブラリーを検索する場合は、**-L** オプションで表示順に、このディレクトリーが検索されます。

オプションの順序は重要です。GCC は、このライブラリーのディレクトリーを認識しない限り、ライブラリー **foo** とリンクできません。そのため、**-L** オプションを使用して先にライブラリーディレクトリーを指定してから、**-l** オプションでライブラリーをリンクするようにしてください。

ライブラリーを使用するコードを 1 つの手順でコンパイルし、リンクする方法

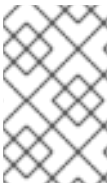
1 つの **gcc** コマンドでコードをコンパイルおよびリンクできる場合は、上記の場合にこれらのオプションを一度に使用します。

関連資料

- GNU コンパイラーコレクション (GCC) の使用: 3.16 Options for Directory Search
- GNU コンパイラーコレクション (GCC) の使用: 「3.15 Options for Linking」

16.4. GCC での静的ライブラリーの使用

静的なライブラリーは、オブジェクトファイルを含むアーカイブとして利用できます。リンク後、それらは作成された実行可能ファイルの一部となります。



注記

Red Hat は、さまざまな理由から静的リンクを使用することは推奨していません。「静的リンクおよび動的リンク」 を参照してください。静的リンクは、特に Red Hat が提供するライブラリーに対して、必要な場合に限り使用してください。

前提条件

- GCC がシステムにインストールされていること
- 静的リンクおよび動的リンクについて理解していること
- 有効なプログラムを構成するソースまたはオブジェクトファイルセット。 静的ライブラリー **foo** のみが必要です。
- **foo** ライブラリーは **libfoo.a**ファイルとして利用でき、動的リンクに **libfoo.so** ファイルは指定されていない。

注記

Red Hat Enterprise Linux に含まれるライブラリーのほとんどは、動的リンク用としてのみサポートされています。次の手順は、動的リンクに **無効の** ライブラリーに対してのみ有効です。「GCC での静的および動的ライブラリーの両方の使用」 を参照してください。

手順

ソースとオブジェクトファイルからプログラムをリンクするには、静的にリンクされたライブラリー **foo** (**libfoo.a**)を追加します。

1. コードが含まれるディレクトリーに移動します。
2. **foo** ライブラリーのヘッダーで、プログラムソースファイルをコンパイルします。

```
$ gcc ... -Iheader_path -c ...
```

header_path は、**foo** ライブラリーのヘッダーファイルを含むディレクトリーのパスに置き換えます。

3. プログラムを **foo** ライブラリーにリンクします。

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

library_path は、**libfoo.a** ファイルを含むディレクトリーへのパスに置き換えます。

4. プログラムを実行するには、以下を行います。

```
$ ./program
```

注意

静的リンクに関連付けられる GCC オプション **-static** は、すべての動的リンクを禁止します。代わりに **-Wl,-Bstatic** オプションおよび **-Wl,-Bdynamic** オプションを使用して、リンカーの動作をより正確に制御します。「GCC での静的および動的ライブラリーの両方の使用」 を参照してください。

16.5. GCC での動的ライブラリーの使用

動的ライブラリーは、スタンドアロンの実行可能ファイルとして利用できます。このファイルは、リンク時およびランタイム時に必要です。このファイルは、アプリケーションの実行可能ファイルからは独立しています。

前提条件

- GCC がシステムにインストールされている
- 有効なプログラムを構成するソースまたはオブジェクトファイルのセット。 動的ライブラリー **foo** のみが必要です。
- **foo** ライブラリーは **libfoo.so** として利用できること

プログラムの動的ライブラリーへのリンク
動的ライブラリー **foo** にプログラムをリンクするには、以下のコマンドを実行します。

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

プログラムを動的ライブラリーにリンクすると、作成されるプログラムは常にランタイム時にライブラリーを読み込む必要があります。ライブラリーの場所を特定するオプションは 2 つあります。

- 実行ファイルに保存された **rpath** の値を使用する方法
- ランタイム時に **LD_LIBRARY_PATH** 変数を使用する方法

実行ファイルに保存された rpath の値を使用する方法
rpath は、リンク時に実行可能ファイルの一部として保存される特殊な値です。後に実行可能ファイルからプログラムを読み込む時に、ランタイムリンカーが **rpath** の値を使用してライブラリーファイルの場所を特定します。

GCC とリンクし、library_path のパスを **rpath** として保存します。

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

library_path のパスは、**libfoo.so** ファイルを含むディレクトリーを参照する必要があります。

注意

-Wl,-rpath= オプションのコンマの後にスペースがあるので注意してください。

後でプログラムを実行するには、以下のコマンドを実行します。

-

```
$ ./program
```

LD_LIBRARY_PATH 環境変数を使用する方法
プログラムの実行可能ファイルに **rpath** がない場合、ランタイムリンカーは **LD_LIBRARY_PATH** 環境変数を使用します。この変数の値は、共有ライブラリーのオブジェクトがあるパスに合わせて、プログラムごとに変更する必要があります。

rpath セットがなく、ライブラリーが **library_path** パスにある状態で、プログラムを実行します。

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

rpath の値を空白にすると柔軟性が出ますが、プログラムを実行するたびに **LD_LIBRARY_PATH** 変数を設定する必要があります。

ライブラリーのデフォルトディレクトリーへの配置
ランタイムのリンカー設定では、複数のディレクトリーを動的ライブラリーファイルのデフォルトの場所として指定します。このデフォルトの動作を使用するには、ライブラリーを適切なディレクトリーにコピーします。

動的リンカーの動作に関する詳細については、本書では扱いません。詳しい情報は、以下の資料を参照してください。

- 動的リンカーの Linux man ページ:

  ```
  $ man ld.so
  ```

- **/etc/ld.so.conf** 設定ファイルの内容:

  ```
  $ cat /etc/ld.so.conf
  ```

- 追加設定なしに動的リンカーにより認識されるライブラリーのレポート (ディレクトリーを含む):

  ```
  $ ldconfig -v
  ```

16.6. GCC での静的および動的ライブラリーの両方の使用

場合によっては、ライブラリーを静的にリンクする場合と動的にリンクする場合とを分ける必要があります。

前提条件

- 静的リンクおよび動的リンクについて理解していること

はじめに
GCC は、動的ライブラリーと静的ライブラリーの両方を認識します。**-lfoo** オプションがあると、gcc はまず、動的にリンクされたバージョンの foo ライブラリーを含む共有オブジェクト (**.so** ファイル) を検索し、静的ライブラリーを含むアーカイブファイル (**.a**) を検索します。これにより、この検索の後に以下の状況が発生する可能性があります。

- 共有オブジェクトのみが見つかり、gcc がそのオブジェクトに動的にリンクする
- アーカイブファイルのみが見つかり、gcc がそのファイルに静的にリンクする

- 共有オブジェクトとアーカイブファイルの両方が見つかり、gcc はデフォルト設定の通りに共有オブジェクトに動的にリンクする
- 共有オブジェクトもアーカイブファイルも見つからず、リンクに失敗する

このようなルールがあるため、リンクするために、静的ライブラリーまたは動的ライブラリーを選択する場合は、gcc が検索可能なバージョンのみを指定するようにします。これにより、**-Lpath** オプションで指定する場合に、静的ライブラリーまたは動的ライブラリーを含むディレクトリーを追加するか、追加しないかで、ある程度制御が可能になります。

また、動的リンクがデフォルトの設定であるため、明示的にリンクを指定する必要があるのは、どちらのバージョンも含むライブラリーを静的にリンクする必要がある場合のみです。考えられる解決方法は以下の 2 つです。

- **-l** オプションではなく、ファイルパスで静的ライブラリーを指定する
- **-Wl** オプションを使用して、オプションをリンカーに渡す

ファイルで静的ライブラリーを指定する方法

通常、gcc は、**-lfoo** オプションを指定して foo ライブラリーにリンクするように指示されます。ただし、代わりに、ライブラリーを含む **libfoo.a** ファイルの完全パスは指定できます。

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

ファイルの拡張子 **.a** から、gcc は、このファイルがプログラムとリンクするためのライブラリーであることを理解します。ただし、ライブラリーファイルの完全パスを指定するのは柔軟な方法ではありません。

-Wl オプションの使用

gcc オプションの **-Wl** は、基盤のリンカーにオプションを渡す特別なオプションです。このオプションの構文は、他の gcc オプションとは異なります。-Wl オプションの後に、リンカーのオプションをコンマ区切りのリストにして入力します。ただし、他の gcc オプションには、スペース区切りのリストにしてオプションを指定する必要があります。gcc が使用する ld リンカーには、**-Bstatic** と **-Bdynamic** のオプションがあり、このオプションの後に来るライブラリーを静的または動的にリンクすべきかどうかを指定します。**-Bstatic** とライブラリーをリンカーに渡した後、後続のライブラリーを **-Bdynamic** オプションで動的にリンクするには、デフォルトの動的リンクの動作を手動で復元する必要があります。

プログラムをリンクするには、**まず** 静的に (**libfirst.a**) ライブラリーをリンクして、**次に** 動的に (**libsecond.so**) リンクして、以下を実行します。

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

注記

gcc は、デフォルトの ld 以外のリンカーを使用するように設定できます。**-Wl** オプションは、gold リンカーにも適用されます。

関連資料

- GNU コンパイラーコレクション (GCC) の使用: 「3.15 Options for Linking」
- binutils 2.32 のドキュメント: 「2.1 Command Line Options」

第17章 GCC でのライブラリーの作成

本章では、ライブラリーの作成手順と、Linux オペレーティングシステムで使用するために必要なライブラリーの概念を説明します。

17.1. ライブラリーの命名規則

特別なファイルの命名規則をライブラリーに使用します。**foo** と呼ばれるライブラリーは、**lib foo .so** または lib**foo** . **a** ファイルとして存在する必要があります。この規則は、リンクする **gcc** の入力オプションでは自動的に理解されますが、出力オプションでは理解されません。

- ライブラリーにリンクする場合は、**-lfoo** のように、**-l** オプションと **foo** の名前でしか、ライブラリーを指定することができません。

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

- ライブラリーの作成時には、**libfoo.so**、**libfoo.a** など、完全なファイル名を指定する必要があります。

関連情報

- 「soname のメカニズム」

17.2. SONAME のメカニズム

動的に読み込まれたライブラリー (共有オブジェクト) は、**soname** というメカニズムを使用して、複数の互換性のあるバージョンのライブラリーを管理します。

前提条件

- 動的リンクとライブラリーについて理解している
- ABI の互換性の概念を理解している
- ライブラリーの命名規則を理解している
- シンボリックリンクを理解している

問題の概要
動的に読み込んだライブラリー (共有オブジェクト) は、独立した実行可能ファイルとして存在します。そのため、依存するアプリケーションを更新せずに、ライブラリーを更新できます。ただし、この概念に関連して以下の問題が生じます。

- ライブラリーの実際のバージョンを特定すること
- 同じライブラリーの複数のバージョンを存在させる必要があること
- 複数のバージョンでそれぞれ ABI の互換性を示す

soname のメカニズム
この問題を解決するには、Linux では soname というメカニズムを使用します。

ライブラリー **foo** の **X.Y** バージョンは、バージョン番号に同じ値 **X** を持つ他のバージョンと ABI の互換性があります。互換性を確保してマイナーな変更を加えると、**Y** の数が増えます。互換性がなくなるようなメジャーな変更を加えると、**X** の数が増えます。

実際のライブラリー **foo** のバージョン X.Y は、**libfoo.so.x.y** ファイルとして存在します。ライブラリーファイルの中に、soname が **libfoo.so.x** の値として記録され、互換性を指定します。

アプリケーションが構築されると、リンカーが **libfoo.so** ファイルを検索してライブラリーを特定します。この名前のシンボリックリンクが存在し、実際のライブラリーファイルを参照している必要があります。次にリンカーは、ライブラリーファイルから soname を読み込み、これをアプリケーションの実行可能ファイルに記録します。最後に、リンカーにより、名前またはファイル名でなく、soname を使用してライブラリーで依存関係を宣言するようにアプリケーションが作成されます。

ランタイムの動的リンカーが実行前にアプリケーションをリンクすると、soname がアプリケーションの実行可能ファイルから読み込まれます。この soname は **libfoo.so.x** です。この名前のシンボリックリンクが存在し、実際のライブラリーファイルを参照している必要があります。soname が変更されないため、これにより、バージョンの Y コンポーネントに関係なく、ライブラリーを読み込むことができます。

注記

バージョン番号の Y の部分は、単一の数字に制限される訳ではありません。また、ライブラリーによっては、バージョンを名前にエンコードするものもあります。

ファイルからの soname の読み込み
somelibrary ライブラリーファイルの soname を表示するには、以下を実行します。

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

somelibrary は、検査するライブラリーの実際のファイル名に置き換えます。

17.3. GCC での動的ライブラリーの作成

動的にリンクされたライブラリー (共有オブジェクト) を使用すると、コードを再利用してリソースを確保でき、またライブラリーコードの更新が簡単になることからセキュリティーの強化を図ることができます。このセクションでは、ソースから動的ライブラリーを構築し、インストールする手順を説明します。

前提条件

- soname メカニズムを理解している
- GCC がシステムにインストールされている
- ライブラリーのソースコード

手順

1. ライブラリーソースのディレクトリーに移動します。
2. 位置独立コードオプション **-fPIC** でオブジェクトファイルに各ソースファイルをコンパイルします。

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

   オブジェクトファイルは元のソースコードファイルと同じファイル名を持ちますが、拡張子は **.o** となります。
3. オブジェクトファイルから共有ライブラリーをリンクします。

```
$ gcc -shared -o libfoo.so.x.y -Wl,-soname,libfoo.so.x some_file.o ...
```

使用するメジャーバージョン番号は X で、マイナーバージョン番号は Y です。

4. **libfoo.so.x.y** ファイルを、システムの動的リンカーが検索できる適切な場所にコピーします。Red Hat Enterprise Linux では、ライブラリーのディレクトリーは **/usr/lib64** となります。

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

このディレクトリー内のファイルを操作するには、root パーミッションが必要な点に注意してください。

5. soname メカニズムのシンボリックリンク構造を作成します。

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

関連資料

- Linux ドキュメントプロジェクト: Program Library HOWTO の 3.Shared Libraries

17.4. GCC および AR での静的ライブラリーの作成

オブジェクトファイルを特別なアーカイブファイルに変換して、静的にリンクするライブラリーを作成できます。

注記

Red Hat は、セキュリティー上の理由から、静的リンクの使用は推奨していません。静的リンクは、特に Red Hat が提供するライブラリーに対して、必要な場合にのみ使用してください。「静的リンクおよび動的リンク」 を参照してください。

前提条件

- GCC と binutils がシステムにインストールされていること
- 静的リンクおよび動的リンクについて理解していること
- 関数を含むソースファイルをライブラリーとして共有していること

手順

1. GCC で仲介となるオブジェクトファイルを作成します。

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

必要に応じて、さらにソースファイルを追加します。作成されるオブジェクトファイルはファイル名を共有しますが、拡張子は **.o** を使用します。

2. **binutils** パッケージの **ar** ツールを使用して、オブジェクトファイルを静的ライブラリー (アーカイブ) に変換します。

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

libfoo.a ファイルが作成されます。

3. **nm** コマンドを使用して、作成されたアーカイブを検証します。

```
$ nm libfoo.a
```

4. 静的ライブラリーファイルを適切なディレクトリーにコピーします。

5. ライブラリーにリンクする場合、GCC は自動的に **.a** のファイル名の拡張子 (ライブラリーが静的リンクのアーカイブであること) を認識します。

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

関連資料

- **ar** ツールの Linux man ページ:

```
$ man ar
```

第18章 MAKE での追加コードの管理

GNU Make ユーティリティー (略称 **Make**) は、ソースファイルからの実行可能ファイルの生成を管理するツールです。**Make** は自動的に、複雑なプログラムのどの部分が変更されたかを判断し、再コンパイルする必要があります。**Make** は、Makefiles と呼ばれる絵設定ファイルを使用して、プログラムを構築する方法を管理します。

18.1. GNU MAKE および MAKEFILE の概要

特定のプロジェクトのソースファイルから使用可能な形式 (通常は実行可能ファイル) を作成するには、いくつかの必要な手順を実行します。後で繰り返し実行できるように、アクションとそのシーケンスを記録します。

Red Hat Enterprise Linux には、この目的に合わせて設計されたビルドシステムである、GNU **make** が含まれています。

前提条件

- コンパイルとリンクの概念を理解している。

GNU make

GNU **make** はビルドプロセスの命令が含まれる Makefile を読み込みます。Makefile には、特定のアクション (**レシピ**) で特定の条件 (**ターゲット**) を満たす方法を記述する複数の **ルール** が含まれています。ルールは、別のルールに階層的に依存できます。

オプションを指定せずに **make** を実行すると、現在のディレクトリーで Makefile を検索し、デフォルトのターゲットに到達しようと試みます。実際の Makefile ファイル名は **Makefile**、**makefile**、および **GNUmakefile** です。デフォルトのターゲットは、Makefile の内容で決まります。

Makefile の詳細

Makefile は比較的単純な構文を使用して **変数** と **ルール** を定義します。Makefile は **ターゲット** と **レシピ** で構成されます。ターゲットでは、ルールが実行された場合の出力を指定します。レシピの行は、TAB 文字で開始する必要があります。

通常 Makefile には、ソースファイルをコンパイルするルール、作成されるオブジェクトファイルをリンクするルール、および階層の上部にあるエントリーポイントとして機能するターゲットが含まれます。

1 つのファイル (**hello.c**) で構成される C プログラムを構築する場合は、以下の **Makefile** を参照してください。

```
all: hello

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

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

上記の例では、ターゲット **all** に到達するには、ファイル **hello** が必要です。**hello** を取得するには、**hello.o** (**gcc** でリンクされる) が必要で、これは **hello.c** (**gcc** でコンパイルされる) に基づいて作成されます。

ターゲットの **all** は、ピリオドで開始されない最初のターゲットであるため、デフォルトのターゲットとなっています。この **Makefile** が現在のディレクトリーに含まれている場合に、引数なしで **make** を実行するのは、**make all** を実行するのと同じです。

一般的な Makefile
より一般的な Makefile は、この手順を一般化するために変数を使用し、ターゲット「clean」を追加して、ソースファイル以外をすべて削除します。

```
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)
```

このような Makefile にソースファイルを追加する場合には、SOURCE 変数が定義されている行に追加する必要があります。

関連情報

- GNU make:2 An Introduction to Makefiles
- 15章GCC でのビルドコード

18.2. 例:例: MAKEFILE を使用した C プログラムの構築

以下の例の手順に従い、Makefile を使用してサンプル C プログラムを構築します。

前提条件

- Makefiles および **make** を理解していること

手順

1. **hellomake** ディレクトリーを作成して、そのディレクトリーに移動します。

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

2. 以下の内容で **hello.c** ファイルを作成します。

```
#include <stdio.h>

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

3. 以下の内容で **Makefile** ファイルを作成します。

```
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)
```

注意

Makefile レシピの行は、Tab 文字で開始する必要があります。ブラウザーから上記のテキストをコピーする場合、スペースを代わりに貼り付けてしまう可能性があります。この場合、手動で修正します。

4. **make** を実行します。

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

このコマンドで、実行可能ファイル **hello** が作成されます。

5. この実行可能ファイル **hello** を実行します。

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

6. Makefile ターゲットの **clean** を実行して、作成されたファイルを削除します。

```
$ make clean
rm -rf hello.o hello
```

関連情報

- 「例:GCC での C プログラムの構築」
- 「例:GCC での C++ プログラムの構築」

18.3. MAKE のドキュメントリソース

make の詳細は、以下に記載のドキュメントを参照してください。

インストールされているドキュメント

- **man** ツールおよび **info** ツールを使用して、お使いのシステムにインストールされている man

ページと情報ページを表示します。

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

オンラインドキュメント

- Free Software Foundation 提供の 『GNU Make Manual』
- Red Hat Developer Toolset User Guide: GNU make

第19章 C および C++ アプリケーション開発での ECLIPSE IDE の使用

開発者によっては、複数のコマンドラインツールではなく、IDE の使用を好む場合があります。
Red Hat は、C および C++ アプリケーションの開発のサポート付きで Eclipse IDE を提供します。

C および C++ アプリケーション開発での Eclipse の使用

Eclipse IDE と、C および C++ アプリケーションの開発のためにこれを使用する方法についての詳細情報は、本書の対象外です。以下のリンクのリソースを参照してください。

関連資料

- 「Using Eclipse」
- Eclipse ドキュメント:『C/C++ Development User Guide』

パート V. アプリケーションのデバッグ

アプリケーションのデバッグに関するトピックは非常に広範囲にわたります。ここでは、開発者向けに複数の状況でデバッグを行うための最も一般的な手法を説明します。

第20章 実行中のアプリケーションのデバッグ

本章では、開発者が直接アクセスできるマシンで、必要に応じて何度でも実行できるアプリケーションのデバッグ方法を紹介します。

20.1. デバッグ情報を使用したデバッグの有効化

アプリケーションおよびライブラリーをデバッグするには、デバッグ情報が必要です。以下のセクションでは、この情報を取得する方法を説明します。

20.1.1. デバッグの情報

実行可能なコードをデバッグする場合に、ツールやプログラマーは 2 種類の情報を使用して、バイナリーコードを理解することができます。

- ソースコードテキスト
- ソースコードテキストがバイナリーコードにどのように関連しているのかの説明

上記はデバッグ情報と呼ばれます。

Red Hat Enterprise Linux は、実行可能なバイナリー、共有ライブラリー、または debuginfo ファイルに ELF 形式を使用します。これらの ELF ファイル内では、DWARF 形式を使用してデバッグ情報が保持されます。

DWARF シンボルは、**readelf -w file** コマンドを使用して読み込みます。

注意

STABS は UNIX で使用される場合もあります。STABS は、機能が少ない旧式の形式です。Red Hat はこの使用を推奨していません。GCC および GDB では、STABS の実稼働および使用はベストエフォートでのみサポートされます。Valgrind および elfutils などの他のツールでは、STABS のサポートはありません。

関連資料

- DWARF デバッグ仕様

20.1.2. GCC を使用した C および C++ アプリケーションのデバッグの有効化

デバッグ情報のサイズが大きい場合、その情報はデフォルトで実行可能ファイルに含まれません。GCC を使用した C および C++ のアプリケーションのデバッグを有効にするには、コンパイラーに対して、デバッグ情報を作成するように、明示的に指示する必要があります。

GCC を使用したデバッグ情報の作成の有効化
コードのコンパイルおよびリンク時に **GCC** でデバッグ情報の作成を有効にするには、**-a** オプションを使用します。

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

- コンパイラーとリンカーで最適化を行うと、実行可能なコードを、元のソースコードと関連付けることが難しくなります。変数の最適化、ループのアンロール、周りの操作へのマージなどが行われる可能性があります。これにより、デバッグに負の影響が及ぶ可能性があります。デ

バッグの体験を向上するには、**-Og** オプションを指定して、最適化を設定することを考慮してください。ただし、最適化レベルを変更すると、実行可能なコードが変更され、バグを取り除くための実際の動作が変更される可能性があります。

- **-fcompare-debug** GCC オプションでは、GCC でコンパイルしたコードを、デバッグ情報を使用して (または、デバッグ情報を使用せずに) テストします。このテストでは、出力されたバイナリーファイルの 2 つが同一であれば合格します。このテストを行うことで、実行可能なコードがデバッグオプションによる影響は受けないようにするだけでなく、デバッグコードにバグが含まれないようにします。**-fcompare-debug** オプションを使用するとコンパイルの時間が大幅に伸びることに留意してください。このオプションに関する詳細は、GCC の man ページを参照してください。

関連情報

- 「デバッグ情報を使用したデバッグの有効化」
- GNU コンパイラーコレクション (GCC) の使用: 「3.10 Options for Debugging Your Program」
- GDB でのデバッグ: 18.3 Debugging Information in Separate Files
- GCC の man ページ:

```
$ man gcc
```

20.1.3. Debuginfo パッケージ

Debuginfo パッケージには、プログラムとライブラリーのデバッグ情報と、デバッグソースコードが含まれます。

前提条件

- デバッグ情報を理解していること

Debuginfo パッケージ
Red Hat Enterprise Linux リポジトリーのパッケージにインストールされているアプリケーションやライブラリーの場合は、別のチャンネルで提供されている別の **debuginfo** パッケージとしてデバッグ情報とデバッグソースコードを取得することができます。debuginfo パッケージには **.debug** ファイルが含まれており、その中には、バイナリーパッケージのコンパイルに使用する DWARF debuginfo とソースファイルがあります。Debuginfo パッケージのコンテンツは、**/usr/lib/debug** ディレクトリーにインストールされます。

debuginfo パッケージでは、名前、バージョン、リリース、アーキテクチャーが同じバイナリーパッケージでのみ有効なデバッグ情報が提供されます。

- バイナリーパッケージ: **packagename-version-release.architecture.rpm**
- debuginfo パッケージ: **packagename-debuginfo-version-release.architecture.rpm**

20.1.4. GDB を使用したアプリケーションまたはライブラリー向けの debuginfo パッケージの取得

GNU デバッガー (GDB) は、自動的に足りないデバッグ情報を認識して、パッケージ名を解決します。

前提条件

- デバッグするアプリケーションまたはライブラリーがシステムにインストールされていること

- システムに GDB がインストールされていること
- システムに **debuginfo-install** ツールがインストールされていること

手順

1. デバッグするアプリケーションまたはライブラリーに割り当てられた GDB を起動します。GDB は不足しているデバッグ情報を自動的に認識し、実行するコマンドを提案します。

```
$ 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. これ以上先に進まずに GDB を終了します。 **q** を入力して、**Enter** を押します。

```
(gdb) q
```

3. GDB が提案するコマンドを実行して、必要な debuginfo パッケージをインストールします。

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

アプリケーションまたはライブラリーの debuginfo パッケージをインストールすると、すべての依存関係の debuginfo パッケージもインストールされます。

4. GDB が debuginfo パッケージを提案できない場合は、「手動でのアプリケーションまたはライブラリー向けの **debuginfo** パッケージの取得」 の手順に従います。

関連情報

- 『Red Hat Developer Toolset User Guide』: 「Chapter 1.5, Installing Debugging Information」
- Red Hat ナレッジベースソリューション: 「RHEL システムで debuginfo パッケージをダウンロードまたはインストールする」

20.1.5. 手動でのアプリケーションまたはライブラリー向けの **debuginfo** パッケージの取得

インストール用にどの **debuginfo** パッケージをインストールするのかを手作業で選択するには、実行ファイルの場所を特定して、インストールするパッケージを検索します。

注記

GDB を使用してインストール用のパッケージを決定すること を推奨します。この手動の手順は、GDB がインストールするパッケージを提案できない場合にのみ使用してください。

前提条件

- アプリケーションまたはライブラリーをシステムにインストールしている。
- **debuginfo-install** ツールは、システムで利用できるようにする必要がある。

手順

1. アプリケーションまたはライブラリーの実行可能ファイルを検索します。

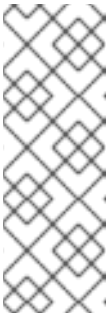
   a. **which** コマンドを使用して、アプリケーションファイルを検索します。

   ```
   $ which nautilus
   /usr/bin/nautilus
   ```

   b. **locate** コマンドを使用して、ライブラリーファイルを検索します。

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

   デバッグの元の理由にエラーメッセージが含まれている場合、ライブラリーのファイル名に同じ追加の数字が含まれる結果を選択します。不明な点がある場合は、ライブラリーファイルの名前に追加の番号が含まれていないものを使用して、残りの手順を試してください。

   注記

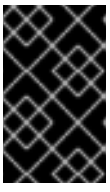
   locate コマンドは **mlocate** パッケージで提供されます。このパッケージをインストールして、その使用を有効にするには、以下のコマンドを実行します。

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

2. ファイルパスを使用して、対象のファイルを提供するパッケージを検索します。

   ```
   # 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
   ```

   この出力では、**name-version.distribution.architecture** の形式でパッケージの一覧が表示されます。**yum** の出力で表示されるバージョンは実際にインストールされているバージョンでない場合があるので、この手順では、パッケージの **名前** のみが重要です。

   重要

   この手順では結果が返されないので、どのパッケージがバイナリーファイル用のパッケージであるか、またこの手順が失敗したかどうかを判別することができません。

3. **rpm** の低レベルのパッケージ管理ツールを使用して、どのパッケージバージョンがシステムにインストールされているかを確認します。パッケージ名を引数として使用します。

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

この出力では、**name-version.distribution.architecture** の形式でインストールされたパッケージの詳細が表示されます。

4. **debuginfo-install** ユーティリティーを使用して **debuginfo** パッケージをインストールします。このコマンドで、直前の手順で確認したパッケージ名およびその他の詳細情報を使用します。

```
# debuginfo-install zlib-1.2.7-17.el7.x86_64
```

アプリケーションまたはライブラリーの **debuginfo** パッケージをインストールすると、すべての依存関係の **debuginfo** パッケージもインストールされます。

関連資料

- 『Red Hat Developer Toolset User Guide』:「Installing Debugging Information」
- ナレッジベースアーティクル:「RHEL システムで debuginfo パッケージをダウンロードまたはインストールする」

20.2. GDB を使用したアプリケーションの内部状態の検証

アプリケーションが適切に機能しない理由を特定するには、その実行を制御し、デバッガーで内部状態を検査します。本セクションでは、このタスクに GNU デバッガー (GDB) を使用する方法を説明します。

20.2.1. GNU デバッガー (GDB)

デバッガーは、コード実行の制御や、コードの状態の検査を有効にするツールです。この機能は、プログラム内で何が発生しているのか、またその発生理由についての調査に使用します。

Red Hat Enterprise Linux には、コマンドラインユーザーインターフェースでこの機能を提供する GNU デバッガー (GDB) が含まれます。

GDB へのグラフィカルフロントエンドについては、Eclipse 統合開発環境をインストールします。「Using Eclipse」を参照してください。

GDB 機能
単一の GDB セッションで、以下をデバッグできます。

- マルチスレッドやフォーク用のプログラムをデバッグ
- 一度に複数のプログラムをデバッグ
- TCP/IP ネットワーク接続経由で接続された **gdbserver** ユーティリティーを使用するコンテナー内またはリモートマシンのプログラムをデバッグ

デバッグの要件
実行可能なコードをデバッグするには、GDB では適切なデバッグ情報が必要です。

- 独自に開発したプログラムの場合は、コードの構築時にデバッグ情報を作成できます。
- パッケージからインストールしたシステムプログラムの場合は、それぞれの debuginfo パッケージをインストールする必要があります。

20.2.2. プロセスへの GDB の割り当て

プロセスを検査するには、GDB がプロセスに **割り当てられている** 必要があります。

前提条件

- GDB がシステムにインストールされている

GDB でのプログラムの起動
プログラムがプロセスとして実行されていない場合は、GDB でプログラムを起動します。

```
$ gdb program
```

program は、ファイル名またはプログラムへのパスに置き換えます。

GDB はプログラムの実行を開始します。**run** コマンドでプロセスの実行を開始する前に、ブレークポイントと **gdb** 環境を設定できます。

実行中のプロセスへの GDB の割り当て
プロセスとしてすでに実行中のプログラムに GDB を割り当てるには、以下を実行します。

1. **ps** コマンドで、プロセス id (**pid**) を検索します。

   ```
   $ ps -C program -o pid h
    pid
   ```

   program は、ファイル名またはプログラムへのパスに置き換えます。

2. このプロセスに GDB を割り当てます。

   ```
   $ gdb program -p pid
   ```

   program は、ファイル名またはプログラムへのパスに置き換えます。**pid** は、**ps** の出力にある実際のプロセス ID 番号に置き換えます。

実行中のプロセスに実行中の GDB を割り当てる手順
実行中のプロセスに実行中の GDB を割り当てるには、以下を実行します。

1. GDB コマンド **shell** を使用して **ps** コマンドを実行し、プログラムのプロセス ID (**pid**) を検索します。

   ```
   (gdb) shell ps -C program -o pid h
    pid
   ```

   program は、ファイル名またはプログラムへのパスに置き換えます。

2. **attach** コマンドを使用して、GDB をプログラムに割り当てます。

   ```
   (gdb) attach pid
   ```

   pid は、**ps** の出力にある実際のプロセス ID の番号に置き換えます。

注記

場合によっては、GDB がそれぞれの実行可能ファイルを検索できない可能性があります。**file** コマンドを使用して、パスを指定します。

```
(gdb) file path/to/program
```

関連資料

- GDB を使用したデバッグ: 2.1 Invoking GDB
- GDB を使用したデバッグ: 4.7 Debugging an Already-running Process

20.2.3. GDB でのプログラムコードの活用

GDB デバッガーがプログラムに割り当てられたら、複数のコマンドを使用して、プログラムの実行を制御できます。

前提条件

- GDB がシステムにインストールされている
- 必要なデバッグ情報を利用できる状態にしている。
  - プログラムはコンパイルされ、デバッグ情報で構築されている。
  - 関連する debuginfo パッケージがインストールされている。
- GDB がデバッグするプログラムに割り当てられている

コードを活用するための GDB コマンド

r (run)
プログラムの実行を開始します。引数を指定して **run** を実行すると、プログラムが通常起動しているかのように、指定した引数が実行可能ファイルに渡されます。通常、ユーザーはブレークポイントの設定後にこのコマンドを実行します。

start
プログラムの実行を開始してメイン機能の開始時に停止します。引数を指定して **start** を実行すると、プログラムが通常起動しているかのように、指定した引数が実行可能ファイルに渡されます。

c (continue)
現在の状態からプログラムの実行を継続します。プログラムの実行は、以下のいずれかが True になるまで継続します。

- ブレークポイントに到達した場合
- 指定の条件を満たした場合
- プログラムがシグナルを受信する場合
- エラーが発生した場合
- プログラムが終了する場合

n (next)

このコマンドのもう 1 つの既知の名前は、**step over** です。現在のソースファイルでコードが次の行に到達するまで、現在の状態からプログラムの実行を続行します。プログラムの実行は、以下のいずれかが True になるまで継続します。

- ブレークポイントに到達した場合
- 指定の条件を満たした場合
- プログラムがシグナルを受信する場合
- エラーが発生した場合
- プログラムが終了する場合

s (step)

このコマンドのもう 1 つの既知の名前は、**step into** です。また、**step** コマンドは、現在のソースファイル内のコードの連続行ごとに実行を停止することも行います。ただし、実行が **関数呼び出し** を含むソース行で停止中の場合には、GDB は、関数呼び出しを入力した後 (実行後ではなく)、実行を停止します。

until location

location オプションで指定したコードの場所に到達するまで、実行が継続されます。

fini (finish)

プログラムの実行を再開し、実行が関数から戻り値として返された時点で停止します。プログラムの実行は、以下のいずれかが True になるまで継続します。

- ブレークポイントに到達した場合
- 指定の条件を満たした場合
- プログラムがシグナルを受信する場合
- エラーが発生した場合
- プログラムが終了する場合

q (quit)

実行を中断して、GDB を終了します。

関連情報

- 「定義したコードの場所で実行を停止するための GDB ブレークポイントの使用」
- GDB を使用したデバッグ - 4.2 Starting your Program
- GDB を使用したデバッグ - 5.2 Continuing and Stepping

20.2.4. GDB でのプログラム内部値の表示

プログラムの内部変数の値を表示することは、プログラムの実行内容を理解する際に重要です。GDB は、内部変数の検査に使用できる複数のコマンドを提供します。このセクションでは、これらのコマンドの中で最も有用なものを説明します。

前提条件

- GDB デバッガーを理解していること

プログラムの内部の状態を表示するための GDB コマンド

p (print)

指定された引数の値を表示します。引数には通常、単純な値 1 つや構造など、複雑度の異なる変数名を指定できます。引数には、プログラム変数やライブラリー関数の使用、テストするプログラムに定義する関数など、現在の言語で有効な式も指定できます。
pretty-printer Python スクリプトまたは Guile スクリプトを使用して GDB を拡張し、 **print** コマンドを使用して、(クラス、構造などの) データ構造をカスタマイズ表示することができます。

bt (backtrace)

現在の実行ポイントに到達するために使用される関数呼び出しのチェーン、または実行が伝えられるまで使用される関数のチェーンを表示します。これは、原因を特定しにくい、深刻なバグ (セグメント障害など) を調査する場合に便利です。
backtrace コマンドに **full** オプションを追加すると、ローカル変数も表示されます。

bt コマンドおよび **info frame** コマンドを使用して表示されるデータをカスタマイズして表示するために、**frame filter** Pythonスクリプトで GDB を拡張できます。**フレーム** という用語は、1 つの関数呼び出しに関連付けられたデータを指します。

info

info コマンドは、さまざまな項目に関する情報を提供する汎用コマンドです。このコマンドでは、項目をオプションとして指定できます。

- **info args** コマンドは、現在選択されているフレームの関数呼び出しの引数を表示します。
- **info locals** コマンドは、現在選択されているフレームにローカル変数を表示します。

使用できる項目を一覧表示するには、GDB セッションで **help info** コマンドを実行します。

```
(gdb) help info
```

l (list)

プログラムが停止するソースコードの行を表示します。このコマンドは、プログラムの実行が停止した場合のみ利用できます。**list** は、厳密には内部状態を表示するコマンドではありませんが、ユーザーがプログラムの実行の次の手順で内部状態にどのような変更が発生するかを理解するのに役立ちます。

関連資料

- Red Hat Developer ブログエントリー: The GDB Python API
- GDB でのデバッグ: 10.9 Pretty Printing

20.2.5. 定義したコードの場所で実行を停止するための GDB ブレークポイントの使用

多くの場合、特定のコードの行に到達するまでプログラムを実行させることには利点があります。

前提条件

- GDB を理解していること

GDB でのブレークポイントの使用

ブレークポイントは、プログラムの実行を停止するように GDB に指示を出すマーカーです。ブレークポイントは、ソースコードの行に関連付けられている最も一般的なものです。ブレークポイントを配置するには、ソースファイルと行番号を指定する必要があります。

- **ブレークポイントを配置する** には、以下を行います。
  - ソースコード **ファイル** の名前と、そのファイルの **行** を指定します。

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

  - **ファイル** が存在しない場合は、現在の実行ポイントにソースファイルの名前が使用されます。

    ```
    (gdb) br line
    ```

  - または、関数名を使用して、起動時にブレークポイントを配置します。

    ```
    (gdb) br function_name
    ```

- タスクを特定の回数反復すると、プログラムでエラーが発生する可能性があります。実行を停止するために追加の **条件** を指定するには、以下を実行します。

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

  condition を、C または C++ 言語の条件に置き換えます。 file と line は、上記と同様に、ファイル名および行数に置き換えます。

- 全ブレークポイントおよびウォッチポイントの状態を **検査** する場合は、以下のコマンドを実行します。

  ```
  (gdb) info br
  ```

- **info br** の出力で表示された **番号** を使用してブレークポイントを **削除** するには、以下のコマンドを実行します。

  ```
  (gdb) delete number
  ```

- 指定の場所のブレークポイントを **削除** するには、以下を実行します。

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

関連資料

- GDB でのデバッグ: 5.1 Breakpoints, Watchpoints, and Catchpoints

20.2.6. データへのアクセスや変更時に実行を停止するための GDB ウォッチポイントの使用

多くの場合、特定のデータが変更されたり、アクセスされるまでプログラムを実行させることには利点があります。このセクションでは、最も一般的ウォッチポイントを取り上げます。

前提条件

- GDB を理解していること

GDB でのウォッチポイントの使用
ウォッチポイントは、プログラムの実行を停止するように GDB に指示を出すマーカーです。ウォッチポイントはデータに関連付けられます。ウォッチポイントを配置するには、変数、複数の変数、またはメモリーアドレスを記述する式を指定する必要があります。

- データの **変更** (書き込み) を行うために、ウォッチポイントを **配置** するには、以下を実行します。

  ```
  (gdb) watch expression
  ```

  expression を、監視する内容を記述する式に置き換えます。変数の場合、**式** は、変数の名前と同じです。

- データ **アクセス** (読み込み) のためのウォッチポイントを **配置** するには、以下を実行します。

  ```
  (gdb) rwatch expression
  ```

- **任意の** データへのアクセス (読み取りおよび書き込みの両方) のためにウォッチポイントを **配置** するには、以下を実行します。

  ```
  (gdb) awatch expression
  ```

- 全ウォッチポイントおよびブレークポイントの状態を **検査** するには以下を実行します。

  ```
  (gdb) info br
  ```

- ウォッチポイントを **削除** するには、以下を実行します。

  ```
  (gdb) delete num
  ```

  num オプションを、**info br** コマンドで報告される番号に置き換えます。

関連資料

- GDB でのデバッグ: 5.1.2 Setting Watchpoints

20.2.7. GDB でのフォーク用またはスレッド化されたプログラムのデバッグ

プログラムによっては、フォークまたはスレッドを使用して、コードの並行実行を実行します。複数の同時実行パスをデバッグするには、特別な留意点があります。

前提条件

- GDB デバッガーを理解していること
- フォークおよびスレッドプロセスのコンセプトを理解していること

GDB でのフォークされたプログラムのデバッグ
フォークとは、プログラム (**親**) により、独立したコピー (**子**) を作成する状況のことを指します。以下の設定およびコマンドを使用して、実行するフォークに対する GDB の対応を変更します。

- **follow-fork-mode** 設定で、フォークの後に GDB が親または子に従うかどうかを制御します。

  set follow-fork-mode parent
  フォークの後に、親プロセスのデバッグを行います。これはデフォルトです。

set follow-fork-mode child

フォークの後に、子プロセスのデバッグを行います。

show follow-fork-mode

follow-fork-mode の現在の設定を表示します。

- **set detach-on-fork** 設定では、GDB が (フォローしていない) 他のプロセスを制御するか、そのまま実行させるかを制御します。

set detach-on-fork on

続いていないプロセス (**follow-fork-mode** の値により異なる) は切り離され、独立して実行されます。これはデフォルトです。

set detach-on-fork off

GDB は両方のプロセスの制御を維持します。フォローしているプロセス (**follow-fork-mode** の値による) は通常通りにデバッグされ、他は一時停止されます。

show detach-on-fork

detach-on-fork の現在の設定を表示します。

GDB でのスレッド化されたプログラムのデバッグ

GDB には、個別のスレッドをデバッグして、独立して操作し、検査する機能があります。GDB が検査したスレッドのみを停止させるには、**set non-stop on** コマンドおよび **set target-async on** コマンドを使用します。これらのコマンドは、**.gdbinit** ファイルに追加できます。その機能が有効になると、GDB がスレッドのデバッグを実行する準備が整います。

GDB は **current thread** の概念を使用します。デフォルトでは、コマンドは現在のスレッドのみに適用されます。

info threads

現在のスレッドを示す **id** 番号および **gid** 番号を使用してスレッドの一覧を表示します。

thread id

指定した **id** を現在のスレッドとして設定します。

thread apply ids command

command コマンドを、**ids** で一覧表示されたすべてのスレッドに適用します。**ids** オプションは、スペースで区切られたスレッド ID の一覧です。特殊な値 **all** は、すべてのスレッドにコマンドを適用します。

break location thread id if condition

スレッド番号 **id** に対してのみ特定の **condition** を持つ特定の **location** にブレークポイントを設定します。

watch expression thread id

スレッド番号 **id** に対してのみ **expression** で定義されるウォッチポイントを設定します。

command&

command コマンドを実行すると、GDB プロンプト **(gdb)** に即座に戻り、バックグラウンドでコードの実行が続行されます。

interrupt

バックグラウンドでの実行が停止されます。

関連資料

- GDB を使用したデバッグ: 4.10 Debugging Programs with Multiple Threads

- GDB を使用したデバッグ: 4.11 Debugging Forks

20.3. アプリケーションの対話の記録

アプリケーションの実行可能コードは、オペレーティングシステムや共有ライブラリーのコードと対話します。これらの対話に関するアクティビティーログを記録すると、実際のアプリケーションコードをデバッグせずに、アプリケーションの動作を詳細にわたり知ることができます。また、アプリケーションの対話を分析すると、バグが発生した状況をピンポイントで特定しやすくなります。

20.3.1. アプリケーションの対話の記録に役立つツール

Red Hat Enterprise Linux は、アプリケーションの相互作用を分析するための複数のツールを提供しています。

strace

strace ツールは、システムコール、シグナル配信、プロセス状態の変更など、アプリケーションと Linux カーネル間の対話 (および改ざん) のトレースを有効にします。

- **strace** は、パラメーターを解釈し、基礎となるカーネルコードに関する知識が得られるため、**strace** の出力と呼び出しの詳しい情報を提供します。数値は、定数名、フラグリストに展開されたビット単位の結合フラグ、実際の文字列を提供するために逆参照された文字配列へのポインターなどにそれぞれ変換されます。ただし、より新しいカーネル機能のサポートがない場合があります。ただし、
- **strace** を使用するために、ログフィルターの設定以外に、特別な設定は必要ありません。
- strace でアプリケーションコードを追跡すると、アプリケーションの実行速度が大幅に遅くなるため、strace は、多くの実稼働環境のデプロイメントには適しません。代わりに、このような場合に SystemTap の使用を検討してください。
- 追跡されたシステムコールとシグナルの一覧をフィルタリングして、キャプチャーしたデータ量を制限できます。
- strace は kernel-userspace の対話のみをキャプチャーし、ライブラリーコールなどはを追跡しません。ライブラリー呼び出しの追跡には ltrace の使用を検討してください。

ltrace

ltrace ツールを使用すると、アプリケーションのユーザー空間呼び出しのログを共有オブジェクト (動的ライブラリー) に記録できます。

- **ltrace** は、ライブラリーへの呼び出しを追跡できるようにします。
- トレースされた呼び出しをフィルタリングして、取得するデータ量を減らすことができます。
- **ltrace** を使用するために、ログフィルターの設定以外に、特別な設定は必要ありません。
- **ltrace** は、軽量、高速で、 **strace** の代わりとして使用できます。**strace** でカーネル関数を追跡する代わりに、**ltrace** で **glibc** などのライブラリー内の各インターフェースを追跡できます。ただし、システムコールのトレースで **ltrace** は strace より正確性に欠ける点に注意してください。
- **ltrace** は、制限されたライブラリー呼び出しセットのみにパラメーターをデコードできます。プロトタイプが、関連する設定ファイルで定義されている呼び出しです。**ltrace** パッケージの一部として、一部の **libacl**、**libc**、および **libm** 呼び出しおよびシステムコールのプ

ロトタイプが提供されます。**ltrace** の出力には多くの場合、生の数値およびポインターのみが含まれます。**ltrace** の出力の解釈には通常、出力にあるライブラリーの実際のインターフェース宣言を確認する必要があります。

SystemTap

SystemTap は、Linux システム上で実行中のプロセスおよびカーネルアクティビティーを調査するための有用なインストルメンテーションプラットフォームです。SystemTap は、独自のスクリプト言語を使用してカスタムイベントハンドラーをプログラミングします。

- **strace** と **ltrace** の使用と比較した場合、ロギングのスクリプトを作成すると、初期の設定フェーズでより多くの作業が必要になります。ただし、スクリプト機能は単にログを生成するだけでなく、SystemTap の有用性を高めます。
- SystemTap は、カーネルモジュールを作成し、挿入すると機能します。SystemTap は効率的に使用でき、システムまたはアプリケーションの実行速度が大幅に低下することはありません。
- SystemTap には一連の使用例が提供されます。

GDB

GNU デバッガーは主に、ロギングではなく、デバッグを目的としています。ただし、その機能の一部は、アプリケーションの対話が主要なアクティビティーとなるシナリオでも役に立ちます。

- GDB では、対話イベントを取得して、後に続く実行パスを即時にデバッグするように、都合よく組み合わせることができます。
- GDB は、他のツールで問題のある状況を最初に特定した後、頻度の低いイベントや特異なイベントへの応答を分析するのに最適です。イベントが頻繁に発生するシナリオで GDB を使用すると、効率が悪くなったり、不可能になったりします。

関連資料

- Red Hat Enterprise Linux SystemTap ビギナーズガイド
- Red Hat Developer Toolset User Guide

20.3.2. strace でのアプリケーションのシステム呼び出しの監視

strace ツールは、システムコール、シグナル配信、プロセス状態の変更など、アプリケーションと Linux カーネル間の対話 (および改ざん) のトレースを有効にします。

前提条件

- **strace** がシステムにインストールされている
  - **strace** をインストールするには、root で以下のコマンドを実行します。

```
# yum install strace
```

手順

strace の追跡仕様構文は、システムコールの特定に役立つ正規表現と syscall クラスを提供することに注意してください。

1. 監視するプロセスを実行またはアタッチします。

- 監視するプログラムが実行していない場合は、**strace** を起動して、**プログラム** を指定します。

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

上記の例で使用されるオプションは必須ではありません。必要な場合に使用します。

- **-f** オプションは、「follow forks」の頭字語です。このオプションは、fork、vfork、および clone システムコールが作成した子を追跡します。
- **-v** または **-e abbrev=none** オプションは、出力の省略を無効し、さまざまな構造フィールドを省略します。
- **-tt** オプションは、各行を絶対タイムスタンプでプレフィックスする **-t** オプションのバリアントです。**-tt** オプションを指定すると、出力される時間にはマイクロ秒が含まれます。
- **-T** オプションは、行末に各システムコールに費やした時間を出力します。
- **-yy** オプションは、ファイル記述子番号に関連付けられたパスの出力を有効にする **-y** オプションのバリアントです。**-yy** オプションはパスだけでなく、ソケットファイル記述子に関連するプロトコル固有の情報、デバイスのファイル記述子に関連するブロックまたはキャラクターのデバイス番号も出力します。
- **-s** オプションは、出力される文字列の最大サイズを制御します。ファイル名は文字列と見なされず、常にフル出力されることに注意してください。
- **-e trace** は、トレースするシステムコールのセットを制御します。**call** を、表示するシステムコールのコンマ区切りリストに置き換えます。**call** を残すと、**strace** はすべてのシステムコールを表示します。一部のシステム呼び出しのグループについては、**strace(1)** man ページで提供されています。

- プログラムがすでに実行中の場合は、プロセス ID (**pid**) を検索して、その id に **strace** を割り当てます。

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

- フォークしたプロセスまたはスレッドを追跡しない場合には、**-f** オプションを使用しないでください。

2. **strace** は、アプリケーションで作成したシステム呼び出しとその詳細を表示します。ほとんどの場合、システム呼び出しのフィルターが設定されていないと、アプリケーションとそのライブラリーは多数の呼び出しを行い、**strace** 出力がすぐに表示されます。

3. **strace** は、トレースしたプロセスがすべて終了する際に終了します。追跡しているプログラムの終了前に監視を中断するには、**Ctrl+C** を押します。

- **strace** がプログラムを起動すると、終了するシグナル (この場合は SIGINT) を起動しているプログラムに送信します。ただし、そのプログラムは、そのシグナルを無視する可能性があることに注意してください。
- 実行中のプログラムに **strace** を割り当てると、そのプログラムは **strace** と共に中断します。

4. アプリケーションが実行したシステム呼び出しの一覧を分析します。

   - リソースへのアクセスや可用性の問題は、エラーを返す呼び出しとしてログに表示されます。
   - システム呼び出しや呼び出しシーケンスのパターンに渡す値により、アプリケーションの動作の原因が分かります。
   - アプリケーションがクラッシュした場合に、重要な情報はおそらく、ログの最後に表示されます。
   - 出力には多くの追加情報が含まれています。ただし、より正確なフィルターを作成して、手順を繰り返すことができます。

備考

- 出力を確認することにも、ファイルに保存することにも利点があります。これを行うには **tee** コマンドを実行します。

```
$ strace ...-o |tee your_log_file.log>&2
```

- 異なるプロセスに対応する個別の出力を表示するには、以下を実行します。

```
$ strace ... -ff -o your_log_file
```

  プロセス ID (**pid**) のプロセスの出力は、**_log_file.pid** に保存されます。

関連資料

- **strace(1)** man ページ
- ナレッジベースアーティクル: 「strace を使用して、コマンドが実行したシステムコールを追跡する」
- 『Red Hat Developer Toolset User Guide』: strace

20.3.3. ltrace でのアプリケーションのライブラリー関数呼び出しの監視

ltrace ツールは、ライブラリー (共有オブジェクト) で利用可能な関数に対するアプリケーションの呼び出しを監視できます。

前提条件

- **ltrace** がシステムにインストールされている

手順

1. 可能であれば、対象のライブラリーおよび関数を特定します。
2. 監視するプログラムが実行されていない場合には、**ltrace** を起動して、**プログラム** を指定します。

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

   -e および **-l** のオプションを使用して、出力をフィルタリングします。

- **function** として表示される関数の名前を指定します。**-e function** オプションは複数回使用できます。何も指定しない場合は、**ltrace** はすべての関数への呼び出しを表示します。
- 関数を指定するのではなく、**-l library** オプションでライブラリー全体を指定することができます。このオプションは、**-e function** オプションと同じように動作します。

詳細情報は、man ページの ltrace(1) を参照してください。

プログラムがすでに実行中の場合は、プロセス id (**pid**) を検索して、その id に **ltrace** を割り当てます。

```
$ ps -C program
(...)
$ ltrace ... -ppid
```

フォークしたプロセスまたはスレッドを追跡しない場合には、**-f** オプションは指定しないでください。

3. **ltrace** はアプリケーションのライブラリー呼び出しを表示します。
多くの場合、アプリケーションは大量の呼び出しを作成し、フィルターが設定されていない場合には、**ltrace** の出力がすぐに表示されます。

4. **ltrace** は、プログラムが終了すると終了します。
追跡しているプログラムの終了前に監視を中断するには、**ctrl+C** を押します。
   - **ltrace** でプログラムを起動した場合には、プログラムは **ltrace** と共に中断します。
   - 実行中のプログラムに **ltrace** を割り当てると、プログラムは **ltrace** と共に終了します。

5. アプリケーションが実行したライブラリー呼び出しの一覧を分析します。
   - アプリケーションがクラッシュした場合に、重要な情報はおそらく、ログの最後に表示されます。
   - 出力には不要な情報が多く含まれています。ただし、より正確なフィルターを作成して、手順を繰り返すことができます。

注記

出力を確認することにも、ファイルに保存することにも利点があります。これを実行するには、**tee** コマンドを使用します。

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

関連資料

- **strace(1)** man ページ
- 『Red Hat Developer Toolset User Guide』: ltrace

20.3.4. SystemTap でのアプリケーションのシステム呼び出しの監視

SystemTap ツールでは、カーネルイベントにカスタムイベントハンドラーを登録できます。**strace** と比較すると、これは使いにくくなりますが、**SystemTap** はより効率的で、より複雑な処理ロジックを使用することができます。

前提条件

- システムに SystemTap がインストールされていること

手順

1. 以下の内容を含む **my_script.stp** ファイルを作成します。

```
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. 監視するプロセスのプロセス ID (**pid**) を検索します。

```
$ ps -aux
```

3. スクリプトで **SystemTap** を実行します。

```
# stap my_script.stp -x pid
```

pid の値は、プロセス ID です。

スクリプトはカーネルモジュールにコンパイルされ、それが読み込まれます。これにより、コマンドの入力から出力の取得までにわずかな遅延が生じます。

4. プロセスでシステム呼び出しが実行されると、呼び出し名とパラメーターがターミナルに出力されます。

5. プロセスが終了した場合、または **Ctrl+C** を押すと、スクリプトは終了します。

関連資料

- 『SystemTap ビギナーズガイド』
- SystemTap タップセットリファレンス
- **strace** 機能に近い SystemTap スクリプトは、**/usr/share/systemtap/examples/process/strace.stp** で利用できます。スクリプトを実行するには、以下を実行します。
  # stap --example strace.stp -x pid

  または

```
# stap --example strace.stp -c "cmd args ..."
```

20.3.5. GDB を使用したアプリケーションシステム呼び出しの遮断

GDB は、プログラムの実行中に発生するさまざまな状況において実行を停止することができます。プログラムがシステム呼び出しを実行する際に実行を停止するには、GDB **catchpoint** を使用します。

前提条件

- GDB ブレークポイントを理解していること
- GDB がプログラムに割り当てられていること

GDB の使用によるシステム呼び出しでのプログラム実行の停止

1. キャッチポイントを設定します。

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

   catch syscall コマンドは、プログラムがシステム呼び出しを実行する際に実行を停止する特殊なタイプのブレークポイントを設定します。

   syscall-name オプショは、呼び出し名を指定します。様々なシステム呼び出しに対して複数のキャッチポイントを指定することができます。**syscall-name** オプションに何も指定しない場合には、システム呼び出しがあると GDB が停止してしまいます。

2. プログラムが実行を開始していない場合は、これを開始します。

   ```
   (gdb) r
   ```

   プログラムの実行が一時停止しているだけの場合は、これを再開します。

   ```
   (gdb) c
   ```

3. GDB は、指定のシステム呼び出しがプログラムによって実行された後に実行を一時停止します。

関連情報

- 「GDB でのプログラム内部値の表示」
- 「GDB でのプログラムコードの活用」
- GDB でのデバッグ: 5.1.3 Setting Catchpoints

20.3.6. アプリケーションによるシグナル処理を遮断するための GDB の使用

GDB は、プログラムの実行中に発生するさまざまな状況において実行を停止することができます。プログラムがオペレーティングシステムからシグナルを受信した時に実行を停止するには、GDB の **キャッチポイント** を使用します。

前提条件

- GDB ブレークポイントを理解していること
- GDB がプログラムに割り当てられていること

GDB でのシグナル受信時のプログラム実行の停止

1. キャッチポイントを設定します。

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

catch signal コマンドは、プログラムがシステムコールを受けたときに実行を停止する特別なブレークポイントを設定します。**signal-type** オプションは、シグナルのタイプを指定します。すべてのシグナルを取得するには、特別な値 **'all'** を使用します。

2. プログラムが実行を開始していない場合は、これを開始します。

```
(gdb) r
```

プログラムの実行が一時停止しているだけの場合は、これを再開します。

```
(gdb) c
```

3. GDB は、プログラムが指定のシグナルを受けると実行を停止します。

関連情報

- 「GDB でのプログラム内部値の表示」
- 「GDB でのプログラムコードの活用」
- GDB でのデバッグ: 5.3.1 Setting Catchpoints

第21章 クラッシュしたアプリケーションのデバッグ

アプリケーションを直接デバッグできない場合があります。このような状況では、アプリケーションの終了時にアプリケーションに関する情報を収集し、後で分析できます。

21.1. コアダンプ

以下のセクションでは、**コアダンプ** の概要、その用途について説明します。

前提条件

- デバッグ情報を理解していること

詳細

コアダンプは、アプリケーションの動作が停止した時点のアプリケーションのメモリーの一部のコピーで、ELF 形式で保存されます。コアダンプには、アプリケーションの内部変数、スタックすべてが含まれ、アプリケーションの最終的な状態を検査することができます。それぞれの実行可能ファイルおよびデバッグ情報を追加すると、実行中のプログラムを分析するのと同様に、デバッガーでコアダンプファイルを分析できます。

Linux オペレーティングシステムカーネルは、この機能が有効な場合に、コアダンプを自動的に記録できます。または、実行中のアプリケーションにシグナルを送信すると、実際の状態に関係なくコアダンプを生成できます。

警告

一部の制限は、コアダンプを生成する機能に影響する場合があります。

21.2. コアダンプでのアプリケーションのクラッシュの記録

アプリケーションのクラッシュを記録するには、コアダンプの保存内容を設定し、システムに関する情報を追加します。

手順

1. コアダンプを有効にします。**/etc/systemd/system.conf** ファイルを編集し、**DefaultLimitCORE** を含む行を以下のように変更します。

```
DefaultLimitCORE=infinity
```

2. システムを再起動します。

```
# shutdown -r now
```

3. コアダンプサイズの制限を削除します。

```
# ulimit -c unlimited
```

この変更を元に戻すには、**unlimited** の代わりに 0 を指定してコマンドを実行します。

4. アプリケーションがクラッシュすると、コアダンプが生成されます。コアダンプのデフォルトの場所は、クラッシュ発生時のアプリケーションの作業ディレクトリーです。

5. SOS レポートを作成して、システムに関する追加情報を提供します。

```
# sosreport
```

これにより、設定ファイルのコピーなど、システムに関する情報が含まれる .tar アーカイブが作成されます。

6. デバッグが行われるコンピューターに、コアダンプと SOS レポートを移動します。既知の場合は、実行可能ファイルも転送します。

重要

実行可能ファイルが不明な場合は、コアファイルの後続の分析で特定します。

7. オプション:コアダンプと SOS レポートを転送後に削除し、ディスク領域を解放します。

関連情報

- ナレッジベースアーティクル:「アプリケーションがクラッシュまたはセグメンテーション違反が発生した時にコアファイルのダンプを有効にする」
- ナレッジベースアーティクル:「Red Hat Enterprise Linux 4.6 以降での sosreport の役割と取得方法」

21.3. コアダンプを使用したアプリケーションのクラッシュの状態の検査

前提条件

- コアダンプファイルおよび SOS レポートがある。
- GDB および elfutils がシステムにインストールされていること

手順

1. クラッシュが発生した実行可能ファイルを特定するには、コアダンプファイルを指定して **eu-unstrip** コマンドを実行します。

```
$ eu-unstrip -n --core=./core.9814
0x400000+0x207000 2818b2009547f780a5639c904cded443e564973e@0x400284
/usr/bin/sleep /usr/lib/debug/bin/sleep.debug [exe]
0x7fff26fff000+0x1000 1e2a683b7d877576970e4275d41a6aaec280795e@0x7fff26fff340 . -
linux-vdso.so.1
0x35e7e00000+0x3b6000
374add1ead31ccb449779bc7ee7877de3377e5ad@0x35e7e00280 /usr/lib64/libc-2.14.90.so
/usr/lib/debug/lib64/libc-2.14.90.so.debug libc.so.6
0x35e7a00000+0x224000
3ed9e61c2b7e707ce244816335776afa2ad0307d@0x35e7a001d8 /usr/lib64/ld-2.14.90.so
/usr/lib/debug/lib64/ld-2.14.90.so.debug ld-linux-x86-64.so.2
```

出力には、行ごとに各モジュールの詳細が、スペースで区切られます。情報は以下の順序で一覧表示されます。

1. モジュールがマッピングされているメモリーアドレス
2. モジュールのビルド ID、およびメモリー内の場所
3. モジュールの実行ファイル名 - 不明の場合は **-**、モジュールがファイルから読み込まれていない場合は **.** と表示されます。
4. デバッグ情報のソース - 使用可能な場合はファイル名が表示されます。実行可能ファイル自体に含まれている場合は **.**、存在しない場合は **-** と表示されます。
5. 主要なモジュールの共有ライブラリー名 (**soname**) または **[exe]**

この例では、重要な詳細は、テキスト **[exe]** を含む行のファイル名 **/usr/bin/sleep** と、ビルド ID **2818b2009547f780a5639c904cded443e564973e** です。この情報を使用して、コアダンプの分析に必要な実行可能ファイルを特定できます。

2. クラッシュした実行可能ファイルを取得します。

   - 可能であれば、クラッシュが発生したシステムからコピーします。コアファイルから抽出したファイル名を使用します。
   - または、お使いのシステムで同じ実行可能ファイルを使用します。Red Hat Enterprise Linux にビルドされた実行ファイルはそれぞれ、固有の build-id 値を持つメモが含まれています。関連する、ローカルで利用可能な実行ファイルの build-id を特定します。

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

     この情報を使用して、リモートシステムの実行可能ファイルをローカルコピーと一致させます。ローカルファイルの build-id とコアダンプに記載されている build-id は一致する必要があります。
   - 最後に、アプリケーションが RPM パッケージからインストールされている場合は、パッケージから実行ファイルを取得できます。**sosreport** 出力を使用して、必要なパッケージの正確なバージョンを確認します。

3. 実行可能ファイルで使用する共有ライブラリーを取得します。実行可能ファイルと同じ手順を使用します。

4. アプリケーションがパッケージとして配布されている場合は、GDB で実行可能ファイルを読み込み、足りない debuginfo パッケージに関するヒントを表示します。詳細は 「GDB を使用したアプリケーションまたはライブラリー向けの debuginfo パッケージの取得」 を参照してください。

5. コアファイルの詳細を調べるには、GDB で実行可能ファイルとコアダンプファイルを読み込みます。

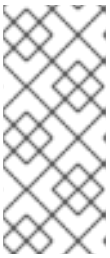
   ```
   $ gdb -e executable_file -c core_file
   ```

   不足しているファイルとデバッグ情報に関する追加のメッセージは、デバッグセッションで不足しているものを特定するのに役に立ちます。必要に応じて直前の手順に戻ります。

   デバッグ情報がパッケージではなくファイルとして利用できる場合は、**symbol-file** コマンドを使用してこのファイルを GDB に読み込みます。

   ```
   (gdb) symbol-file program.debug
   ```

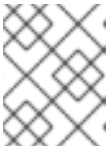
program.debug は、実際のファイル名に置き換えます。



注記

コアダンプに含まれるすべての実行可能ファイルのデバッグ情報をインストールする必要はない場合があります。これらの実行可能ファイルのほとんどは、アプリケーションコードで使用されるライブラリーです。これらのライブラリーが分析中の問題の直接原因でない可能性があるので、ライブラリーのデバッグ情報を含める必要はありません。

6. GDB コマンドを使用して、クラッシュした時点のアプリケーションの状態を検証します。「GDB を使用したアプリケーションの内部状態の検証」 を参照してください。



注記

コアファイルを分析する場合に、GDB が実行中のプロセスに割り当てられるわけではありません。実行を制御するコマンドは影響を受けません。

関連資料

- GDB でのデバッグ: 2.1.1 Choosing Files
- GDB でのデバッグ: 18.1 Commands to Specify Files
- GDB でのデバッグ: 18.3 Debugging Information in Separate Files

21.4. GCORE を使用したプロセスメモリーのダンプ

コアダンプのデバッグのワークフローでは、プログラムの状態をオフラインで分析できます。対象のプロセスで環境にアクセスするのが困難な場合など、実行中のプログラムでこのワークフローを使用することに利点がある場合があります。**gcore** コマンドを使用すると、実行中にプロセスのメモリーをダンプできます。

前提条件

- コアダンプを理解していること
- システムに GDB がインストールされていること

手順

gcore を使用してプロセスメモリーをダンプするには、以下を実行します。

1. プロセス ID (**pid**) を検索します。**ps**、**pgrep**、**top** などのツールを使用します。

```
$ ps -C some-program
```

2. このプロセスのメモリーをダンプします。

```
$ gcore -o filename pid
```

これでファイル **filename** が作成され、その中にプロセスメモリーがダンプされます。メモリーをダンプしている間は、プロセスの実行は停止します。

3. コアダンプが終了すると、プロセスは通常の実行を再開します。

4. SOS レポートを作成して、システムに関する追加情報を提供します。

```
# sosreport
```

これにより、設定ファイルのコピーなど、システムに関する情報が含まれる .tar アーカイブが作成されます。

5. デバッグを行うコンピューターに、プログラムの実行ファイル、コアダンプ、および SOS レポートを移動します。

6. オプション:コアダンプと SOS レポートを転送後に削除し、ディスク領域を解放します。

関連情報

- ナレッジベースアーティクル:「アプリケーションを再起動せずにコアファイルを取得する方法」

21.5. GDB での保護されたプロセスメモリーのダンプ

プロセスのメモリーをダンプしないようにマークできます。これにより、リソースを節約し、プロセスのメモリーに機密データが含まれている場合にセキュリティーを強化することができます。カーネルのコアダンプ (**kdump**) および手動のコアダンプ (**gcore**、GDB) は、このようにマークされたメモリーをダンプしません。

このように保護されていても、プロセスメモリーのすべてのコンテンツをダンプする必要がある場合があります。以下の手順では、GDB デバッガーを使用して、すべてのコンテンツをダンプする方法を説明します。

前提条件

- コアダンプを理解していること
- システムに GDB がインストールされていること
- GDB が保護されているメモリープロセスに割り当てられていること

手順

1. **/proc/PID/coredump_filter** ファイルの設定を無視するように GDB を設定します。

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

2. メモリーページのフラグ **VM_DONTDUMP** を無視するように GDB を設定します。

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

3. メモリーをダンプします。

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

core-file は、メモリーをダンプするファイルの名前に置き換えます。

関連資料

- GDB でのデバッグ: 10.19 How to Produce a Core File from Your Program

パート VI. パフォーマンスの監視

開発者は、パフォーマンスに最も大きな影響を与えるプログラムの部分にフォーカスできるようにプログラムのプロファイルを作成します。収集されるデータのタイプには、プロセッサーの時間を最も多く消費するプログラムのセクションや、メモリーが割り振られる場所などがあります。プロファイルでは、実際のプログラムの実行からデータを収集します。そのため、収集されるデータの質は、プログラムが実施する実際のタスクに影響されます。プロファイル時に実施されるタスクは、実際の使用を表すものでなければなりません。 これにより、プログラムの実際の使用に起因する問題への対応を開発時に行えるようになります。

Red Hat Enterprise Linux には、プロファイリングデータを収集する各種ツール (**Valgrind**、**OProfile**、**perf** および **SystemTap**) が多数含まれています。各ツールは、以下のセクションで説明されているように、特定のプロファイルの実行に適しています。

第22章 VALGRIND

Valgrind は、アプリケーションを詳細にわたりプロファイリングするために使用可能な動的分析ツールを構築するインストルメンテーションフレームワークです。デフォルトのインストールには、5 つの標準ツールが含まれます。Valgrind ツールは通常、メモリー管理やスレッドの問題を調査するのに使用します。Valgrind は、初期化されていないメモリーの使用、メモリーの不適切な割り振り/解放、およびシステム呼び出しの不適切な引数などのエラーをチェックするためのユーザー空間バイナリーのインストルメンテーションを提供します。Valgrind のプロファイルツールは、大半のバイナリーで標準ユーザーによる使用が可能ですが、他のプロファイラーと比較すると、Valgrind プロファイルの実行速度は大幅に遅くなります。バイナリーのプロファイルには、Valgrind は特別な仮想マシン内でこのツールを実行することで、Valgrind が全バイナリーの命令を傍受できるようになります。Valgrind のツールは、ユーザー空間プログラムにおけるメモリー関連の問題を検出する場合に最も役立ちます。 ただし、これは、時間固有の問題、カーネルスペースのインストルメンテーションやデバッグには適していません。

Valgrind は、調査中のプログラムやライブラリー向けに **debuginfo** パッケージがインストールされている場合に、最も有用で正確なレポートを提供します。「デバッグ情報を使用したデバッグの有効化」を参照してください。

22.1. VALGRIND ツール

Valgrind スイートは、以下のツールで構成されています。

memcheck

このツールは、次の方法でプログラム内のメモリー管理の問題を検出します。

- メモリーに対する読み込み/書き込みをすべて確認する
- **malloc**、**free**、**new**、または **delete** への呼び出しのようなメモリー操作を傍受します。

その他の方法ではメーモリー管理問題を検出するのが難しいため、**memcheck** が、おそらく最も使用される Valgrind ツールです。このような問題は長期にわたって検出されないままになることが多く、最終的には診断しにくいクラッシュを生じさせます。

特定のツールが選択されていない場合における、デフォルトのツールとしての **memcheck** 関数。

cachegrind

cachegrind は、CPU で I1、D1、および L2 キャッシュの詳細なシミュレーションを実行することで、コード内のキャッシュミスのソースを正確に特定するキャッシュプロファイルです。このプロファイラーは、ソースコードの各行に累積される命令、キャッシュミス数、メモリー参照を表示します。 また、**cachegrind** は関数別、モジュール別、およびプログラム全体の要約や、マシン毎の命令についての数を表示することもできます。

callgrind

cachegrindと同様に、**callgrind** はキャッシュ動作をモデル化できます。ただし、**callgrind** の主な目的は実行したコードのコールグラフデータを記録することです。

massif

massif はヒーププロファイラーです。これは、プログラムが使用するヒープメモリーを測定し、ヒープブロック、ヒープ管理のオーバーヘッドおよびスタックのサイズに関する情報を提供します。ヒーププロファイルは、ヒープメモリーの使用を縮小する方法を特定する場合に有用です。仮想メモリーを使用するシステムでは、ヒープメモリーの使用率が最適化されたプログラムは、メモリーが不足することは少なくなり、必要とするページングの量が少なくなるために処理も速くなります。

helgrind

POSIX pthreads スレッドのプリミティブを使用するプログラムでは、**helgrind** は同期エラーを検出します。以下は、このようなエラーに該当します。

- POSIX pthreads API の誤用
- ロックの順序付けの問題から生じる潜在的なデッドロック
- データレース (ロックが適切でない状態でのメモリーへのアクセス)

22.2. VALGRIND の使用

valgrind パッケージとその依存関係により、**Valgrind** プロファイルの実行に必要なツールがすべてインストールされます。**Valgrind** でプログラムをプロファイリングするには、以下を使用します。

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

toolname の引数の一覧については、「Valgrind ツール」 を参照してください。**Valgrind** ツールのスイート以外に、 **none** は **toolname** の有効な引数として使用できます。この引数を使用すると、プロファイルを実行せずにプログラムを **Valgrind** 下で実行できます。これは、**Valgrind** 自体のデバッグやベンチマークに役立ちます。

Valgrind に、固有のファイルにすべての情報を送信するように指示することも可能です。これを実行するには、**--log-file=filename** オプションを使用します。たとえば、実行可能ファイル **hello** のメモリー使用状況を確認して、**output** にプロファイル情報を送信するには、以下を使用します。

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

Valgrind についての詳細は、「関連情報」 を参照してください。また、ツールの **Valgrind** スイートに関する他の利用可能なドキュメントも併せて参照してください。

22.3. 関連情報

Valgrindの詳細は、**man valgrind** を参照してください。Red Hat Enterprise Linux では、包括的な **Valgrind ドキュメント** が PDF および HTML 形式で提供されています。これらのドキュメントは以下の場所にあります。

- **/usr/share/doc/valgrind-version/valgrind_manual.pdf**
- **/usr/share/doc/valgrind-version/html/index.html**

第23章 OPROFILE

OProfile は、**oprofile** パッケージで提供されるオーバーヘッドコストの少ない、システム全体のパフォーマンス監視ツールです。システムのプロセッサー上にあるパフォーマンス監視ハードウェアを使用して、メモリーの参照タイミング、第 2 レベルのキャッシュ要求の回数、受け取るハードウェア割り込みの回数など、システム上のカーネルと実行可能ファイルに関する情報を取得します。**OProfile** は、Java Virtual Machine (JVM) で実行されるアプリケーションのプロファイリングも実行できます。

以下は、**OProfile** が提供するツールの選択です。

ophelp

システムプロセッサーで使用可能なイベントと、各イベントの簡単な説明を表示します。

operf

主なプロファイリングツール。**operf** ツールは、Linux Performance Event サブシステムを使用します。これにより、**OProfile** が、システムのパフォーマンス監視ハードウェアを使用するその他のツールと共に動作できるようになります。
以前に使用していた **opcontrol** ツールとは異なり、初期設定は必要ありません。 **--system-wide** オプションが使用されている場合を除き、root 権限がなくても使用できます。

ocount

イベント発生の絶対数をカウントするためのツール。これは、プロセスごと、CPU ごと、またはスレッドごとに、システム全体のイベントをカウントできます。

opimport

サンプルデータベースファイルをシステム用に外部のバイナリー形式からネイティブの形式に変換します。異なるアーキテクチャーからのサンプルデータベースを解析する場合にのみこのオプションを使用してください。

opannotate

アプリケーションがデバッグシンボルでコンパイルされている場合は、実行可能ファイル用の注釈付きのソースを作成します。

opreport

記録されたパフォーマンスデータを読み取り、プロファイル仕様で指定したサマリーを生成します。異なるプロファイル仕様を使用して、同じプロファイルデータから異なるレポートを生成できます。

23.1. OPROFILE の使用

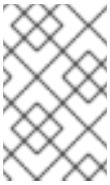
operf はプロファイリングデータを収集するための推奨されるツールです。このツールには初期設定の必要がなく、すべてのオプションはコマンドラインで渡されます。レガシーの **opcontrol** ツールとは異なり、**operf** は **root** 権限なしに実行できます。**operf** ツールの使用方法に関する詳細は、**『システム管理者ガイド』**の「operf の使用」の章をを参照してください。

例23.1 ocount の使用

以下の例は、**sleep** ユーティリティーの実行時に、**ocount** を使用したイベント量のカウントを示しています。

```
$ 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
```



注記

イベントは、プロセッサーの実装に固有です。**perf_event_paranoid** オプションを設定するか、カウントをユーザー空間イベントのみに制限する必要がある場合があります。

例23.2 operf の基本的な使用方法

以下の例では、**operf** ツールを使用して、**ls -l ~** コマンドからプロファイリングデータを収集します。

1. **ls** コマンドのデバッグ情報をインストールします。

```
# debuginfo-install -y coreutils
```

2. プロファイリングを実行します。

```
$ operf ls -l ~
Profiling done.
```

3. 収集したデータを分析します。

```
$ opreport --symbols
CPU: Intel Skylake microarchitecture, speed 3.4e+06 MHz (estimated)
Counted cpu_clk_unhalted events () 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
[...]
```

例23.3 Java プログラムのプロファイリングでの operf の使用

以下の例では、**operf** ツールを使用して Java (JIT) プログラムからプロファイリングデータを収集し、次に **opreport** を使用して、シンボルごとのデータを出力します。

1. この例で使用するデモ用の Java プログラムをインストールします。これは **java-1.8.0-openjdk-demo** パッケージの一部であり、**Optional** チャンネルに含まれます。**Optional** チャンネル の使用方法は、「Optional および Supplementary リポジトリーの追加」 を参照してください。**Optional** チャンネルを有効にしたら以下のパッケージをインストールします。

```
# yum install java-1.8.0-openjdk-demo
```

2. **OProfile** の **oprofile-jit** パッケージをインストールして、Java プログラムからプロファイリングデータを収集できるようにします。

```
# yum install oprofile-jit
```

3. **OProfile** データのディレクトリーを作成します。

```
$ mkdir ~/oprofile_data
```

4. デモプログラムが含まれるディレクトリーに移動します。

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

5. プロファイリングを開始します。

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

6. ホームディレクトリーに移動して、収集したデータを分析します。

```
$ cd
```

```
$ opreport --symbols --threshold 0.5
```

出力例は以下のとおりです。

```
$ 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    /usr/lib64/libfreetype.so.6.10.0
128      0.5124   libc-2.17.so             __memset_sse2
```

23.2. OPROFILE のドキュメント

OProfileについての詳細は、**oprofile**(1) の man ページを参照してください。Red Hat Enterprise Linux には、**OProfile** に関する包括的な 2 つのガイドが含まれており、このガイドは **file:///usr/share/doc/oprofile-version/** にあります。

OProfile Manual

OProfile の詳細にわたる設定や用途の説明が含まれる包括的なマニュアルは、**file:///usr/share/doc/oprofile-version/oprofile.html** にあります。

OProfile Internals

OProfile の内部の機能に関する情報は、**file:///usr/share/doc/oprofile-version/internals.html** にあり、 OProfile アップストリームへの寄稿を検討しておられるプログラマー向けの役立つ情報が含まれています。

第24章 SYSTEMTAP

SystemTap は、Linux システム上で実行中のプロセスおよびカーネルアクティビティーを調査するための有用なインストルメンテーションプラットフォームです。プローブを実行するには、以下の手順に従います。

1. どのシステムイベント (たとえば、仮想ファイルシステムの読み込み、パケット送信) が特定のアクション (たとえば、印刷、解析、またはデータ操作) をトリガーするかを指定する **SystemTap スクリプト** を書き込みます。

2. SystemTap がスクリプトを C プログラムに変換し、さらにカーネルモジュールにコンパイルします。

3. SystemTap がこのカーネルモジュールを読み込み、実際のプローブを実行します。

SystemTap スクリプトは、通常のシステム運用への割り込みを最小限に抑えてシステム運用を監視し、システムの問題を診断する際に役立ちます。インストルメント化されたコードを再コンパイルしたり再インストールすることなく、実行中のシステムテストの仮説をすばやくインストルメント化できます。**kernel-space** をプローブする SystemTap スクリプトをコンパイルするために、SystemTap は 3 つの異なる **カーネル情報パッケージ**からの情報を使用します。

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

これらのカーネル情報パッケージは、プローブ対象のカーネルと一致する必要があります。さらに、複数のカーネル用に SystemTap スクリプトをコンパイルするには、各カーネルのカーネル情報パッケージがインストールされている必要もあります。

24.1. 追加情報

SystemTap の詳細は、以下の Red Hat ドキュメントを参照してください。

- 『SystemTap ビギナーズガイド』
- SystemTap タップセットリファレンス

第25章 PERFORMANCE COUNTERS FOR LINUX (PCL) ツールおよび PERF

Performance Counters for Linux (PCL) は、パフォーマンスデータを収集し、分析するためのフレームワークを提供するカーネルベースのサブシステムです。Red Hat Enterprise Linux 7 には、データやユーザー空間ツール **perf** を収集するためにこのカーネルサブシステムが含まれており、これを使用して収集したパフォーマンスデータを分析します。 PCL サブシステムは、リタイヤした命令やプロセッサーのクロックサイクルなどの、ハードウェアイベントを測定するために使用できます。主なページの障害やコンテキストスイッチなど、ソフトウェアイベントを測定することも可能です。たとえば、PCL カウンターは、リタイアしたプロセスの命令数やプロセッサーのクロックサイクルを基に **Instructions Per Clock** (IPC) を算出することができます。IPC の割合が低い場合は、コードが CPU をあまり使用していないことが分かります。他のハードウェアイベントを使用して、CPU パフォーマンスの低さを診断することも可能です。

パフォーマンスカウンターは、サンプルを記録するように設定することもできます。サンプルの相対的な量を使用して、コードのどの領域がパフォーマンスに最も影響があるかを特定することができます。

25.1. PERF ツールコマンド

役に立つ **perf** コマンドには、以下が含まれます。

perf stat
: この **perf** コマンドは、実行された命令や消費したクロックサイクルなど、一般的なパフォーマンスイベントに関する全体的な統計を提供します。オプションを指定すると、デフォルトの計測イベント以外のイベントを選択することができます。

perf record
: この **perf** コマンドはパフォーマンスデータをファイルに記録し、後で **perf report** を使用して分析を行うことができます。

perf report
: この **perf** コマンドは、ファイルからパフォーマンスデータを読み取り、記録されたデータの分析を行います。

perf list
: この **perf** コマンドは、特定のマシンで利用可能なイベントを一覧表示します。これらのイベントは、パフォーマンス監視ハードウェアや、システムのソフトウェア設定によって異なります。

perf コマンドの詳細な一覧を取得するには、**perf help** を使用します。**perf** コマンドごとに **man** ページの情報を取得するには、**perf help コマンド**を使用します。

25.2. PERF の使用

プログラムの実行の統計またはサンプルを収集するために基本的な PCL インフラストラクチャーを使用することが比較的単純な方法になります。以下のセクションでは、全体的な統計やサンプリングの簡単な例を紹介します。

make およびその子についての統計情報を収集するには、以下のコマンドを使用します。

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

perf コマンドは、多数の異なるハードウェアおよびソフトウェアカウンターを収集します。次に、以下の情報を出力します。

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

 244011.782059  task-clock-msecs         #      0.925 CPUs
        53328  context-switches         #      0.000 M/sec
          515  CPU-migrations           #      0.000 M/sec
      1843121  page-faults              #      0.008 M/sec
 789702529782  cycles                   #   3236.330 M/sec
1050912611378  instructions             #      1.331 IPC
 275538938708  branches                 #   1129.203 M/sec
   2888756216  branch-misses            #      1.048 %
   4343060367  cache-references         #     17.799 M/sec
    428257037  cache-misses             #      1.755 M/sec

 263.779192511  seconds time elapsed
```

perf ツールはサンプルを記録することもできます。たとえば、**make** コマンドおよびその子に関するデータを記録するには、以下を使用します。

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

これで収集されたサンプル数とサンプルが保存されているファイルが表示されます。

```
[ 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) ツールと OProfile の競合

OProfile と Performance Counters for Linux (PCL) は、どちらも同じハードウェアの Performance Monitoring Unit (PMU) を使用します。PCL **perf** コマンドの使用を試行する際に OProfile が実行中の場合は、OProfile の開始時に以下のようなエラーメッセージが表示されます。

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

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

perf コマンドを使用するには、まず OProfile をシャットダウンします。

```
# opcontrol --deinit
```

次に、**perf.data** を分析してサンプルの相対頻度を測定することができます。レポート出力には、コマンド、オブジェクト、サンプルの機能などが含まれます。**perf report** を使って **perf.data** の分析を出力します。たとえば、以下のコマンドは最も時間がかかる実行可能ファイルのレポートを作成します。

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

出力は以下のようになります。

```
# 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
```

左のコラムはサンプルの相対頻度を示しています。この出力では、**make** が **xsltproc** および **pdfxmltex** で最も多くの時間を消費していることを示しています。**make** が完了する時間を短縮するには、**xsltproc** と **pdfxmltex** にフォーカスします。**xsltproc** が実行する機能を一覧表示するには、以下を実行します。

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

以下が生成されます。

```
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              [.] xmlXPathNodeSetSort__internal_alias
     2.60%12271959697  libxml2.so.2.7.6              [.] valuePop
     2.41%11379910918  libxml2.so.2.7.6              [.] xmlXPathIsNaN__internal_alias
     2.19%10340901937  libxml2.so.2.7.6              [.] valuePush__internal_alias
```

付録A 改訂履歴

改訂番号 7-6.1、2018 年 10 月 30 日 (火)、Vladimír Slávik

7.6 GA リリース用のビルド

改訂番号 7-6、2018 年 8 月 21 日 (火)、Vladimír Slávik

7.6 ベータ版のビルド

改訂番号 7-5.1、2018 年 4 月 10 日 (火)、Vladimír Slávik

7.5 GA リリース向けに作成

改訂番号 7-5、2018 年 1 月 9 日 (火)、Vladimír Slávik

新規バージョン 7.5 のベータ版のプレビューを公開

改訂番号 7-4.1、2017 年 8 月 22 日 (火)、Vladimír Slávik

関連製品の新規リリース用に更新

改訂番号 7-4、2017 年 7 月 26 日 (水)、Vladimír Slávik

7.4 GA リリース用にビルド。開発用のワークステーション設定に関する章を追加

改訂番号 1-12、2017 年 5 月 26 日 (金)、Vladimír Slávik

古くなった情報を削除するために更新

改訂番号 7-3.9、2017 年 5 月 15 日 (月)、Robert Krátký

7.4 ベータ版のビルド