Red Hat Enterprise Linux 9

RHEL 9 での C および C++ アプリケーションの開発

開発者用ワークステーションのセットアップ、および Red Hat Enterprise Linux 9 での C および C++ アプリケーションの開発とデバッグ

Last Updated: 2024-05-14

Red Hat Enterprise Linux 9 RHEL 9 での C および C++ アプリケーションの開発

開発者用ワークステーションのセットアップ、および Red Hat Enterprise Linux 9 での C および C++ アプリケーションの開発とデバッグ

法律上の通知

概要

Red Hat Enterprise Linux 9 で利用可能なさまざまな機能とユーティリティーを使用して、C および C++ アプリケーションを開発およびデバッグします。

目次

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、Red Hat CTO である Chris Wright のメッセージ をご覧ください。

RED HAT ドキュメントへのフィードバック (英語のみ)

Red Hat ドキュメントに関するご意見やご感想をお寄せください。また、改善点があればお知らせください。

Jira からのフィードバック送信 (アカウントが必要)

1. Jira の Web サイトにログインします。
2. 上部のナビゲーションバーで **Create** をクリックします。
3. **Summary** フィールドにわかりやすいタイトルを入力します。
4. **Description** フィールドに、ドキュメントの改善に関するご意見を記入してください。ドキュメントの該当部分へのリンクも追加してください。
5. ダイアログの下部にある **Create** をクリックします。

第1章 開発ワークステーションの設定

Red Hat Enterprise Linux 9 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、開発で最も一般的なユースケースと、インストールする項目を紹介します。

1.1. 前提条件

- グラフィカル環境のシステムがインストールされ、サブスクライブされている。

1.2. デバッグおよびソースのリポジトリーの有効化

Red Hat Enterprise Linux の標準インストールでは、デバッグリポジトリーおよびソースリポジトリーが有効になっていません。このリポジトリーには、システムコンポーネントのデバッグとパフォーマンスの測定に必要な情報が含まれます。

手順

- ソースおよびデバッグの情報パッケージチャンネルを有効にします。**$(uname -i)** の部分は、システムのアーキテクチャーで一致する値に自動的に置き換えられます。

アーキテクチャー名	値
64 ビット Intel および AMD	x86_64
64 ビット ARM	aarch64
IBM POWER	ppc64le
64 ビット IBM Z	s390x

1.3. 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、効果的なバージョン管理が必須になります。Red Hat Enterprise Linux には、Git という名前の分散型バージョン管理システムが同梱されています。

手順

1. **git** パッケージをインストールします。

```
# dnf install git
```

2. オプション: 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@example.com"
```

Full Name と **email@example.com** を、お客様の名前とメールアドレスに置き換えます。

3. オプション: 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 を、テキストエディターを起動するのに使用するコマンドに置き換えます。

関連情報

- Git およびチュートリアルの Linux の man ページ:

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

多くの Git コマンドには、独自の man ページがあります。例は、**git-commit(1)** を参照してください。

- **Git ユーザーマニュアル** - Git の HTML ドキュメントは **/usr/share/doc/git/user-manual.html** にあります。
- Pro Git - オンライン版の **Pro Git** ブックでは、Git、概念、用途が詳細に説明されています。
- Reference - オンライン版の Git の Linux man ページ

1.4. C および C++ でアプリケーションを開発するための設定

Red Hat Enterprise Linux には、C および C++ のアプリケーションを作成するツールが同梱されています。

前提条件

- デバッグリポジトリーおよびソースリポジトリーが有効である。

手順

1. GNU Compiler Collection (GCC)、GNU Debugger (GDB) などの開発ツールが含まれる **Development Tools** パッケージグループをインストールします。

```
# dnf group install "Development Tools"
```

2. **clang** コンパイラー、**lldb** デバッガーなどの LLVM ベースのツールチェインをインストールします。

```
# dnf install llvm-toolset
```

3. オプション: Fortran 依存関係用に、GNU Fortran コンパイラーをインストールします。

```
# dnf install gcc-gfortran
```

1.5. アプリケーションをデバッグするための設定

Red Hat Enterprise Linux には、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計測のツールが同梱されています。

前提条件

- デバッグリポジトリーおよびソースリポジトリーが有効である。

手順

1. デバッグに役立つツールをインストールします。

```
# dnf install gdb valgrind systemtap ltrace strace
```

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

```
# dnf install dnf-utils
```

3. 環境設定用の SystemTap ヘルパースクリプトを実行します。

```
# stap-prep
```

stap-prep は、現在 **実行中** のカーネルに関連するパッケージをインストールすることに注意してください。これは、実際にインストールされているカーネルと異なる場合があります。**stap-prep** が正しい **kernel-debuginfo** パッケージおよび **kernel-headers** パッケージをインストールするには、**uname -r** コマンドを使用して現在のカーネルバージョンを再度チェックし、必要に応じてシステムを再起動します。

4. **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。詳細は SELinux の使用 を参照してください。

1.6. 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。

前提条件

- デバッグリポジトリーおよびソースリポジトリーが有効である。

手順

1. パフォーマンス測定用のツールをインストールします。

```
# dnf install perf papi pcp-zeroconf valgrind strace sysstat systemtap
```

2. 環境設定用の SystemTap ヘルパースクリプトを実行します。

```
# stap-prep
```

stap-prep は、現在 **実行中** のカーネルに関連するパッケージをインストールすることに注意してください。これは、実際にインストールされているカーネルと異なる場合があります。**stap-prep** が正しい **kernel-debuginfo** パッケージおよび **kernel-headers** パッケージをインストー

ルするには、**uname -r** コマンドを使用して現在のカーネルバージョンを再度チェックし、必要に応じてシステムを再起動します。

3. Performance Co-Pilot (PCP) コレクターサービスを有効にして開始します。

```
# systemctl enable pmcd && systemctl start pmcd
```

第2章 C または C++ の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。

2.1. RHEL 9 の GCC

Red Hat Enterprise Linux 9 には、標準のコンパイラーとして GCC 11 が同梱されています。

GCC 11 のデフォルトの言語標準設定は C++17 です。これは、コマンドラインオプション **-std=gnu++17** を明示的に使用するのと同じです。

C++20 などの新しい言語標準、およびこれらの新しい言語標準で導入されたライブラリー機能は、まだ実験的なものと見なされています。

関連情報

- GCC 11 への移植
- GCC 11 を使用した C++17 へのコードの移植

2.2. GCC でのビルドコード

ソースコードを実行可能コードに変換する必要がある状況について説明します。

2.2.1. 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。

考えられるコード形式

C 言語および C++ 言語には、以下の 3 つのコード形式があります。

- 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リストは、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、ファイル名拡張子 **.a** を使用するオブジェクトコードのバリアントです。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 つの手順を行います。必要となるアプリケーションまたはツールはそれぞれ異なります。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 (コンパイルおよびリンク) に **gcc** コマンドを 1 つ使用できます。GCC は、アクションとそのシーケンスを自動的に選択します。

1. ソースファイルを、オブジェクトファイルにコンパイルする
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)。

GCC を実行して、1 つのステップでコンパイルのみ、リンクのみ、またはコンパイルとリンクの両方を実行できます。これは、入力タイプや必要とされる出力タイプにより決定します。

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ことを検討することが推奨されます。

2.2.2. 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。

2.2.3. GCC で C および C++ のアプリケーションのデバッグの有効化

デバッグの情報が大きいと、デフォルトでは実行ファイルが含まれません。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、ファイルを作成するように、コンパイラーに明示的に指定する必要があります。

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、**-g** 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実行すると、元のソースコードと関連付けることが難しい実行可能コードが生成される場合があります。変数が最適化されたり、ループがアンロールされたり、操作が周囲の操作にマージされたりす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になり、バグを取り除くための動作が変更する可能性があります。
- デバッグ情報にマクロ定義も追加するには、**-g** の代わりに **-g3** オプションを使用します。
- GCC オプション **-fcompare-debug** 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

- GNU コンパイラーコレクション (GCC) の使用 - Options for Debugging Your Program
- GDB を使用したデバッグ - Debugging Information in Separate Files
- GCC の man ページ:

  ```
  $ man gcc
  ```

2.2.4. 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を分析するためにより多くのリソースを割り当てると、より最適な結果が得られます。

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	最適化して、コード実行速度を向上させます (数値が大きいほど、速度は高くなります)。
s	ファイルサイズを最適化します。

レベル	説明
fast	レベルを **3** にして、**fast** にすると、厳密な標準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

リリースビルドの最適化オプションは **-O2** です。

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を行えるように、**-Og** オプションが便利です。バグによっては、特定の最適化レベルでのみ出現するため、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、以下の関連資料を参照してください。

関連情報

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

```
$ man gcc
```

2.2.5. GCC でコードを強化するオプション

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できます。適切なコンパイラーオプションセットを選択すると、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作成できます。

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、以下のオプションリスト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 -fstack-clash-protection -
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は、以下のオプションを使用します。このオプションは、リリースバージョンのオプションと合わせて使用してください。

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

関連情報

- Defensive Coding Guide

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

2.2.6. 実行ファイルを作成するコードのリンク

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。

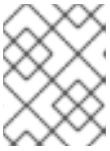
手順

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 ... objfile.o another_object.o ... -o executable-file
```

executable-file という名前の実行ファイルが、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リストの前に、必要なオプションを追加します。



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。

2.2.7. 以下に例を示します。GCC を使用した C プログラムのビルド (1 つの手順でコンパイルとリンク)

以下の例では、簡単な C++ のサンプルプログラムを構築する手順を説明します。

この例では、コードをコンパイルおよびリンクする方法を 1 つの手順で行います。

前提条件

- GCC の使用方法を理解している。

手順

1. **hello-c** 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 hello-c
$ cd hello-c
```

2. 以下の内容を含む **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、リンクします。

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

これにより、コードがコンパイルされ、オブジェクトファイル **hello.o** が作成され、オブジェクトファイルから実行ファイル **helloworld** がリンクされます。

4. 作成された実行可能ファイルを実行します。

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

2.2.8. 以下に例を示します。GCC を使用した C プログラムの構築 (2 つの手順でコンパイルとリンク)

以下の例では、簡単な C++ のサンプルプログラムを構築する手順を説明します。

この例では、コードのコンパイルとリンクは、2 つの別個のステップです。

前提条件

- GCC の使用方法を理解している。

手順

1. **hello-c** 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 hello-c
$ cd hello-c
```

2. 以下の内容を含む **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!
```

2.2.9. 以下に例を示します。GCC を使用した C++ プログラムのビルド (1 つの手順でコンパイルとリンク)

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。

この例では、コードをコンパイルおよびリンクする方法を 1 つの手順で行います。

前提条件

- **gcc** と **g++** の相違点を理解している。

手順

1. **hello-cpp** 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 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、リンクします。

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

これにより、コードがコンパイルされ、オブジェクトファイル **hello.o** が作成され、オブジェクトファイルから実行ファイル **helloworld** がリンクされます。

4. 作成された実行可能ファイルを実行します。

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

2.2.10. 以下に例を示します。GCC を使用した C++ プログラムの構築 (2 つの手順でコンパイルとリンク)

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。

この例では、コードのコンパイルとリンクは、2 つの別個のステップです。

前提条件

- **gcc** と **g++** の相違点を理解している。

手順

1. **hello-cpp** 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 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!
```

2.3. GCC でのライブラリーの使用

コード内でのライブラリーの使用について説明します。

2.3.1. 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** として知られるライブラリーは、**libfoo.so** ファイルまたは **libfoo.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 ... -l foo ...
```

- 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。

2.3.2. 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。特に Red Hat Enterprise Linux で C および C++ 言語を使用するコンテキストでは、静的リンクと動的リンクの違いを理解することが重要です。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。

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**) から静的ライブラリーを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 **CodeReady Linux Builder** 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。

上述のデメリットにより、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、使用しないようにしてください。

静的リンクの場合

静的リンクは、次のようないくつかのケースでは合理的な選択が可能です。

- 動的リンクが使用できないライブラリーを使用している
- 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。

2.3.3. リンク時間の最適化

リンク時の最適化 (LTO) により、リンク時の中間表現を利用して、コンパイラーがプログラムのすべての翻訳単位に対してさまざまな最適化を行うことができます。その結果、実行可能ファイルとライブラリーが小さくなり、実行速度が速くなります。また、LTO を使用すると、コンパイル時にパッケージのソースコードをより徹底的に分析できます。これにより、潜在的なコーディングエラーに対するさまざまな GCC 診断が向上します。

既知の問題

- 単一定義規則 (ODR) に違反すると、**-Wodr** 警告が生成されます
未定義の動作をもたらす ODR の違反は、**-Wodr** 警告を生成します。これは通常、プログラムのバグを示しています。**-Wodr** 警告はデフォルトで有効になっています。

- LTO はメモリー消費の増加を引き起こします
コンパイラーは、プログラムを設定する変換単位を処理するときに、より多くのメモリーを消費します。メモリーが制限されているシステムでは、プログラムをビルドするときに LTO を無効にするか、並列処理レベルを下げてください。

- GCC は一見未使用の機能を削除します
GCC は、コンパイラーが asm() ステートメントが参照するシンボルを認識できないため、未使用と見なされる関数を削除する場合があります。その結果、コンパイルエラーが発生する場合があります。これを防ぐには、プログラムで使用するシンボルに **__attribute__((used))** を追加します。

- **-fPIC**オプションを使用してコンパイルすると、エラーが発生します
GCC は asm() ステートメントの内容を解析しないため、**-fPIC** コマンドラインオプションを使用してコードをコンパイルすると、エラーが発生する可能性があります。これを防ぐには、変換ユニットをコンパイルするときに **-fno-lto** オプションを使用します。
詳細については、LTO FAQ – Symbol usage from assembly language を参照してください。

- **.symver** ディレクティブを使用したシンボルのバージョン管理は LTO と互換性がありません
asm() ステートメントで **.symver** ディレクティブを使用してシンボルのバージョン管理を実装することは、LTO と互換性がありません。ただし、**symver** 属性を使用してシンボルのバージョン管理を実装することは可能です。以下に例を示します。

  __attribute__ ((_symver_ ("<symbol>@VERS_1"))) void <symbol>_v1 (void) { }

関連情報

関連情報

- GCC マニュアル – 関数属性
- GCC Wiki – リンク時間の最適化

2.3.4. 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 つの部分で設定されます。

- 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

ヘッダーファイルでは、ライブラリーで提供する関数や変数など、ライブラリーのインターフェイスを記述します。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。対象のライブラリーがディレクトリーにリンクされていることが分からないと、GCC は、ライブラリー **foo** をリンクできません。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。

1つの手順でライブラリーを使用するコードをコンパイルおよびリンクする方法

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、上記のオプションを一度に使用します。

関連情報

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

2.3.5. 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。リンクを行うと、作成された実行ファイルの一部となります。

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。
- 静的リンクおよび動的リンクを理解している。
- 有効なプログラムを設定するソースまたはオブジェクトのファイルセット。静的ライブラリー **foo** だけが必要です。
- **foo** ライブラリーは **libfoo.a** ファイルとして利用でき、動的リンクには **libfoo.so** ファイルがありません。

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。

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** として検索可能) 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 を、**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 を、**libfoo.a** ファイルを含むディレクトリーのパスに置き換えます。

4. あとでプログラムを実行するには、次のコマンドを実行します。

```
$ ./program
```

警告

静的リンクに関連する 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 で静的ライブラリーおよび動的ライブラリーの両方を使用」 を参照してください。

2.3.6. 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ファイルとして提供します。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設定するソースまたはオブジェクトファイルセットがある。動的ライブラリー **foo** だけが必要になります。
- **foo** ライブラリーが **libfoo.so** ファイルとして利用できる。

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、次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、リンク時に実行ファイルの一部として保存される特別な値です。その後、実行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。

重要

-Wl,-rpath= オプションのコンマの後にスペースを追加しないでください。

あとでプログラムを実行するには、次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

プログラムの実行ファイルに **rpath** がない場合、ランタイムリンカーは **LD_LIBRARY_PATH** の環境変数を使用します。この変数の値は、プログラムごとに変更する必要があります。この値は、共有ライブラリーオブジェクトがあるパスを表す必要があります。

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ありますが、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な説明は、本書の対象外です。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
  ```

2.3.7. GCC で静的ライブラリーおよび動的ライブラリーの両方を使用

場合によっては、静的ライブラリーと動的ライブラリーの両方をリンクする必要があります。このような場合には、いくつかの課題があります。

前提条件

前提条件

- 静的リンクおよび動的リンクの理解

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。したがって、この検索により、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、デフォルトでは gcc が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、静的と動的の両方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** オプションで、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が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カーに渡した後、以降のライブラリーを **-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、first ライブラリーを静的にリンク (**libfirst.a**) して、second ライブラリーを動的にリンク (**libsecond.so**) 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。

関連情報

- GNU コンパイラーコレクション (GCC) の使用 - 3.14 Options for Linking
- binutils 2.27 のドキュメント - 2.1 Command Line Options

2.4. GCC でのライブラリーの作成

ライブラリーを作成する手順と、Linux オペレーティングシステムでライブラリーに使用される必要な概念について説明します。

2.4.1. 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** として知られるライブラリーは、**libfoo.so** ファイルまたは **libfoo.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。

2.4.2. soname のメカニズム

動的に読み込んだライブラリー (共有オブジェクト) は、**soname** と呼ばれるメカニズムを使用して、複数の互換性のある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を理解している。
- 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、この概念では、以下の問題が発生します。

- 実際のライブラリーバージョンを特定

- 同じライブラリーに対して複数のバージョンが必要
- 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 と呼ばれるメカニズムを使用します。

foo ライブラリーの **X.Y** バージョンは、バージョン番号 (**X**) が同じ値でマイナーバージョンが異なるバージョンと、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** の数字が増えます。互換性がなくなるような、メジャーな変更を加えると、**X** の数字が増えます。

foo ライブラリーバージョン **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。

アプリケーションを構築すると、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、アプリケーションの実行ファイルに記録します。最後に、リンカーにより、名前でもファイル名でもなく、soname を使用してライブラリーで依存関係を宣言する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ファイルから読み込まれます。この soname は **libfoo.so.x** と呼ばれます。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。soname が変更しないため、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、1つの数字である必要はありません。また、ライブラリーによっては、名前にバージョンが組み込まれているものもあります。

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 を表示します。

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

somelibrary は、検証するライブラリーのファイル名に置き換えます。

2.4.3. 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) では以下が可能です。

- コードを再利用してリソースを予約する
- ライブラリーコードの更新を容易化にしてセキュリティーを強化する

以下の手順に従って、ソースから動的ライブラリーを構築してインストールします。

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がある。

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、オリジナルのソースコードファイルと同じファイル名ですが、拡張子が **.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にある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。

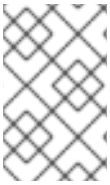
```
# ln -s libfoo.so.x.y libfoo.so.x
# ln -s libfoo.so.x libfoo.so
```

関連情報

- Linux ドキュメントプロジェクト - Program Library HOWTO - 3.共有ライブラリー

2.4.4. 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。詳細は、「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。
- 静的リンクおよび動的リンクを理解している。

- ライブラリーとして共有している関数を含むソースファイルが利用できる。

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 ...
```

関連情報

- Linux の man ページ **ar(1)**:

```
$ man ar
```

2.5. MAKE でのさらなるコードの管理

GNU の make ユーティリティー (通称 **make**) は、ソースファイルから実行ファイルの生成を制御するツールです。**make** は自動的に、複雑なプログラムのどの部分が変更され、再度コンパイルする必要があるのかを判断します。**make** は Makefile と呼ばれる設定ファイルを使用して、プログラムを構築する方法を制御します。

2.5.1. 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 (通常は実行ファイル) を作成するには、必要な手順を完了します。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** で設定されます。ターゲットでは、ルールが実行された場合にどのような出力が表示されるのかを指定します。レシピの行は、TAB 文字で開始する必要があります。

通常、Makefile 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、および階層上部のエントリーポイントとしてのロールを果たすターゲットで設定されます。

1 つのファイル (**hello.c**) で設定される 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
```

この例では、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) が必要で、**hello.c** (**gcc** でコンパイル) を基に作成します。

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、この手順を正規化する変数を使用し、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は、SOURCE 変数が定義されている行に追加します。

関連情報

- GNU make:2 An Introduction to Makefiles

2.5.2. 以下に例を示します。Makefile を使用した C プログラムの構築

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

前提条件

- Makefile と **make** の概念を理解している。

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。

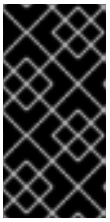
```
CC=gcc
CFLAGS=-c -Wall
SOURCE=hello.c
OBJ=$(SOURCE:.c=.o)
EXE=hello

all: $(SOURCE) $(EXE)

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

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

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



重要

Makefile レシピの行は、Tab 文字で開始する必要があります。上記のテキストをドキュメントからコピーする際に、カットアンドペーストのプロセスでは、タブではなくスペースが貼り付けられる場合があります。この場合は、手動で修正してください。

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
```

2.5.3. make のドキュメント

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** ツールで、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

第3章 アプリケーションのデバッグ

デバッグアプリケーションのトピックは非常に広範囲です。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。

3.1. 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。次のセクションでは、この情報を取得する方法を説明します。

3.1.1. デバッグの情報

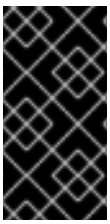
実行なコードをデバッグしている場合は、2 種類の情報により、ツール、さらにはプログラマーがバイナリーコードを理解できます。

- 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

このような情報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維持されます。

ELF ファイルに保存されている DWARF 情報を表示するには、**readelf -w file** コマンドを実行します。



重要

STABS は、UNIX でしばしば使用される、以前の、機能の少ない形式です。Red Hat は、この使用を推奨していません。GCC と GDB は、最適な作業でのみ、STABS の実稼働および使用を提供します。Valgrind や **elfutils** などの他のツールの一部は STABS では動作しません。

関連情報

- DWARF デバッグ仕様

3.1.2. GCC で C および C++ のアプリケーションのデバッグの有効化

デバッグの情報が大きいと、デフォルトでは実行ファイルが含まれません。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、ファイルを作成するように、コンパイラーに明示的に指定する必要があります。

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、**-g** 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実行すると、元のソースコードと関連付けることが難しい実行可能コードが生成される場合があります。変数が最適化されたり、ループがアンロールされたり、操作が周囲の操作にマージされたりす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になり、バグを取り除くための動作が変更する可能性があります。

- デバッグ情報にマクロ定義も追加するには、**-g** の代わりに **-g3** オプションを使用します。
- GCC オプション **-fcompare-debug** 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

- GNU コンパイラーコレクション (GCC) の使用 - Options for Debugging Your Program
- GDB を使用したデバッグ - Debugging Information in Separate Files
- GCC の man ページ:

```
$ man gcc
```

3.1.3. debuginfo パッケージおよび debugsource パッケージ

debuginfo パッケージおよび **debugsource** パッケージには、プログラムおよび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。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、追加のチャンネルから個別の **debuginfo** パッケージおよび **debugsource** パッケージを取得できます。

デバッグの情報パッケージタイプ

デバッグに使用できるパッケージには、以下の 2 つのタイプがあります。

debuginfo パッケージ

debuginfo パッケージは、バイナリーコード機能用に人間が判読可能な名前を提供するために必要なデバッグ情報を提供します。このパッケージには、DWARF デバッグ情報が含まれる **.debug** ファイルが含まれています。このファイルは、**/usr/lib/debug** ディレクトリーにインストールされます。

debugsource パッケージ

debugsource パッケージには、バイナリーコードのコンパイルに使用されるソースファイルが含まれています。適切な **debuginfo** パッケージおよび **debugsource** パッケージの両方がインストールされている状態で、GDB、LLDB などのデバッガーは、バイナリーコードの実行をソースコードに関連付けることができます。ソースコードファイルは、**/usr/src/debug** ディレクトリーにインストールされています。

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

デバッグ情報は、コードをデバッグするために必要です。パッケージからインストールされるコードの場合、GNU デバッガー (GDB) は足りないデバッグ情報を自動的に認識し、パッケージ名を解決し、パッケージの取得方法に関する具体的なアドバイスを提供します。

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。
- GDB と **debuginfo-install** ツールがシステムにインストールされている。詳細については、アプリケーションをデバッグするための設定 を参照してください。
- **debuginfo** および **debugsource** パッケージを提供するリポジトリーを設定し、システムで有効にしている。詳細については、デバッグおよびソースリポジトリーの有効化 を参照してください。

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、足りない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 /bin/ls...Reading symbols from .gnu_debugdata for /usr/bin/ls...(no debugging symbols found)...done.
(no debugging symbols found)...done.
Missing separate debuginfos, use: dnf debuginfo-install coreutils-8.30-6.el8.x86_64
(gdb)
```

2. GDB を終了します。**q** と入力して、**Enter** で確認します。

```
(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。

```
# dnf debuginfo-install coreutils-8.30-6.el8.x86_64
```

dnf パッケージ管理ツールは、変更の概要を提供し、確認を求め、確認後に必要なファイルをすべてダウンロードしてインストールします。

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の debuginfo パッケージの取得 で説明されている手順に従います。

関連情報

- Red Hat ナレッジベースソリューション How can I download or install debuginfo packages for RHEL systems?

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

実行ファイルの場所を特定し、インストールするパッケージを見つけることで、インストールする **debuginfo** パッケージを手動で判断できます。

注記

Red Hat は、GDB を使用して、インストールするパッケージを判断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。

- アプリケーションまたはライブラリーが、パッケージからインストールされている。
- **debuginfo-install** ツールが、システムで利用できるようにする必要がある。
- **debuginfo** パッケージを提供するチャネルをシステム上で設定し、有効にする。

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。

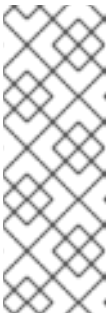
   a. **which** コマンドを使用して、アプリケーションファイルを検索します。

   ```
   $ which less
   /usr/bin/less
   ```

   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.1
   /usr/lib64/libz.so.1.2.11
   ```

   デバッグの元の理由にエラーメッセージが含まれる場合は、ライブラリーのファイル名にエラーメッセージに記載されている番号と同じ追加番号が含まれるものを選択します。疑わしい場合は、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、次のコマンドを実行します。

   ```
   # dnf install mlocate
   # updatedb
   ```

2. ファイルを提供するパッケージの名前およびバージョンを検索します。

   ```
   $ rpm -qf /usr/lib64/libz.so.1.2.7
   zlib-1.2.11-10.el8.x86_64
   ```

   この出力では、インストールされているパッケージの詳細が、name:epoch-version.release.architecture 形式で提供されます。

重要

この手順では結果が生成されないので、どのパッケージがこのバイナリーファイルを提供しているかは判断できません。次のような状況が考えられます。

- このファイルは、**現在** の設定でパッケージ管理ツールに認識されないパッケージからインストールされます。
- このファイルは、ローカルにダウンロードして手動でインストールしたパッケージからインストールされます。この場合、適切な **debuginfo** パッケージを自動的に判断することはできません。
- パッケージ管理ツールの設定が正しく設定されていません。
- このファイルは、どのパッケージからもインストールされません。そのような場合は、それぞれの **debuginfo** パッケージも存在しません。

これ以降の手順はこの手順によって異なるため、この状況を解決するか、この手順を中止する必要があります。正確なトラブルシューティング手順の説明は、この手順の範囲外です。

3. **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。そのコマンドで、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-1.2.11-10.el8.x86_64
```

関連情報

- 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

3.2. GDB を使用したアプリケーションの内部状況の検証

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

3.2.1. GNU デバッガー (GDB)

Red Hat Enterprise Linux には GNU デバッガー (GDB) が含まれ、コマンドラインユーザーインターフェイスを使用して、プログラム内で何が起こっているかを調べることができます。

GDB 機能

1 つの GDB セッションで、以下のタイプのプログラムをデバッグできます。

- マルチスレッドプログラムおよびフォークプログラム
- 一度に複数のプログラム
- 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リモートマシンまたはコンテナー内のプログラム

デバッグの要件

実行コードをデバッグするには、GDB では、その特定のコードのデバッグ情報が必要です。

- ユーザーが開発したプログラムでは、コードの構築中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、debuginfo パッケージをインストールする必要があります。

3.2.2. プロセスへの GDB の割り当て

プロセスを検証するには、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

- GCC 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していない場合は、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するように設定します。**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。

実行中のプロセスに GDB を割り当て

プロセスとして実行中のプログラムに GDB を割り当てるには、以下を行います。

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 -p pid
   ```

   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

実行中のプロセスに実行中の GDB を割り当てるには、以下を行います。

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。

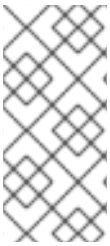
   ```
   (gdb) shell ps -C program -o pid h
    pid
   ```

   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 が適切な実行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
```

関連情報

- 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

3.2.3. GDB を使用したプログラムコードのステップ実行

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

- 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 はデバッグするプログラムに割り当てられている。

コードをステップ実行する 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たかのように、その引数が実行ファイルに渡されます。通常は、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ますが、プログラムのメイン機能の開始時に停止します。**start** を引数と共に実行すると、その引数が、プログラムが通常起動したかのように実行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
- プログラムにより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)

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
- プログラムにより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)

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、**関数呼び出し** を含むソース行で実行が現在停止すると、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、実行が関数から返されたときに停止します。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
- プログラムにより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)

実行を終了し、GDB を終了します。

関連情報

- GDB を使用したデバッグ - Starting your Program
- GDB を使用したデバッグ - Continuing and Stepping

3.2.4. 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。これらのコマンドの中で最も有用なものは次のとおりです。

p (print)

指定された引数の値を表示します。通常、引数は単純な 1 つの値から構造まで、あらゆる複雑な変数の名前です。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、または実行が終了するまで使用される関数のチェーンを表示します。これは、深刻なバグ (セグメント障害など) を調査し、見つけるのが困難な原因に役に立ちます。
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。これは、説明する項目を指定するオプションを取ります。

- **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オプションを表示します。
- **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。

使用できる項目をリスト表示するには、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。

関連情報

- Red Hat Developers Blog エントリー - The GDB Python API
- GDB でのデバッグ: Pretty Printing

3.2.5. GDB ブレークポイントを使用して、定義したコードの場所で実行を停止

多くの場合、コードの一部のみが検証されます。ブレークポイントは、コード内の特定の場所で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のが最も一般的です。その場合、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、次のコマンドを実行します。

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

関連情報

- GDB を使用したデバッグ - Breakpoints, Watchpoints, and Catchpoints

3.2.6. データへのアクセスや変更を停止するための 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。次の例は、最も一般的な使用例です。

前提条件

- GDB の理解

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ています。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、次を使用します。

```
(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** コマンドで返された番号に置き換えます。

関連情報

- GDB を使用したデバッグ - Setting Watchpoints

3.2.7. GDB でのフォーク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、並行コード実行を実現します。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

- プロセスのフォークおよびスレッドの概念を理解している。

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**) が独立したコピー (**子**) を作成する状況です。フォーク発生時の GDB の動作に影響を与えるには、以下の設定およびコマンドを使用します。

- **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実行します。これがデフォルトになります。

  set follow-fork-mode child
  フォークの後に子のプロセスをデバッグします。

  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 (続いていない) 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。これがデフォルトになります。

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、**現在のスレッド** 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リストを表示します。

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** でリスト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 のリストです。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 コマンドを実行して、すぐに gdb プロンプト **(gdb)** に戻りますが、バックグラウンドでコード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。

関連情報

- 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

3.3. 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。この相互作用の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しなくても、アプリケーションの動作を十分に把握できます。または、アプリケーションの相互作用を分析することで、バグが現れる条件を特定するのに役立ちます。

3.3.1. アプリケーションの相互作用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 ツールでは主に、アプリケーションが使用するシステムコール (カーネル関数) のロギングが可能になります。

- **strace** がパラメーターを解釈し、下層のカーネルコードに関する知識が得られるため、**strace** ツールでは呼び出しに関する詳細な出力が得られます。数値は、定数名、フラグリストにデプロイメント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。最新のカーネル機能のサポートがない場合があります。
- 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。
- **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。代替方法として、**ltrace** または SystemTap の使用を検討してください。
- Red Hat Developer Toolset で利用可能な **strace** のバージョンでは、システムコールの改ざんも行えます。この機能は、デバッグに役立ちます。

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** ツールを使用すると、ライブラリーへの呼び出しを追跡できます。
- 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** ツールは軽量で高速で、**strace** に代わる機能を提供します。**strace** でカーネルの関数を追跡する代わりに、**ltrace** で **glibc** など、ライブラリー内の各インターフェイスを追跡できます。
- **ltrace** は、**strace** などの既知の呼び出しを処理しないため、ライブラリー関数に渡す値を記述することができません。**ltrace** の出力には、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、出力にあるライブラリーの実際のインターフェイス宣言を確認する必要があります。

注記

Red Hat Enterprise Linux 9 では、既知の問題により、**ltrace** が実行ファイルを追跡できなくなります。この制限は、ユーザーが構築する実行ファイルには適用されません。

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 (GDB) 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相互作用が重要な主要なアクティビティーであるシナリオでも有用です。

- GDB を使用すると、相互作用イベントを取得して、後続の実行パスの即時デバッグを簡単に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、まれなイベントまたは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。

関連情報

- SystemTap の使用
- Red Hat Developer Toolset User Guide

3.3.2. strace でアプリケーションのシステムコールの監視

strace ツールは、アプリケーションを実行するシステム (カーネル) コールの監視を有効にします。

前提条件

- **strace** がシステムにインストールされている。

手順

1. 監視するシステムコールを特定します。
2. **strace** を起動して、プログラムに割り当てます。
   - 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
     ```

   - 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
```

- **call** を、表示するシステムコールに置き換えます。**-e trace=call** オプションを複数回使用できます。何も指定しない場合、**strace** はすべてのシステムコールタイプを表示します。詳細は、man ページの **strace(1)** を参照してください。
- 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は、**-f** オプションを指定しないでください。

3. **strace** は、アプリケーションで作成したシステムコールとその詳細を表示します。
ほとんどの場合、システムコール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。

4. **strace** ツールは、プログラムの終了時に、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** でプログラムを起動すると、そのプログラムは **strace** とともに中断します。
   - 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** とともに中断します。

5. アプリケーションが実行したシステム呼び出しのリストを分析します。
   - 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。
   - システムコールに渡される値とコールシーケンスのパターンは、アプリケーションの動作の原因に関する洞察を提供します。
   - アプリケーションがクラッシュした場合、重要な情報はおそらくログの最後にあります。
   - 出力には不要な情報が多く含まれています。ただし、目的のシステムコールに対してより正確なフィルターを作成し、この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。これを行うには、**tee** コマンドを使用します。

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

関連情報

- man ページの **strace(1)**:

```
$ man strace
```

- 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 ユーザーガイドの strace の章

3.3.3. ltrace で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) で利用可能な関数へのアプリケーションの呼び出しを監視できます。

注記

Red Hat Enterprise Linux 9 では、既知の問題により、**ltrace** が実行ファイルを追跡できなくなります。この制限は、ユーザーが構築する実行ファイルには適用されません。

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. **ltrace** を起動し、プログラムに割り当てます。
   - 監視するプログラムが実行していない場合は、**ltrace** を起動して、**プログラム** を指定します。

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

   - 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 -f -l library -e function program -ppid
     ```

   - **-e** オプション、**-f** オプション、および **-l** オプションを使用して、出力にフィルターを設定します。
     - 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。何も指定しないと、**ltrace** は全関数への呼び出しを表示します。
     - 関数を指定する代わりに、**-l library** オプションでライブラリー全体を指定できます。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。
     - 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は、**-f** オプションを指定しないでください。

     詳細情報は、man ページの **ltrace**(1) を参照してください。
3. **ltrace** は、アプリケーションにより作成されたライブラリーコールを表示します。
   多くの場合は、フィルターが設定されていないと、アプリケーションは多数の呼び出しを作成し、**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コールのリストを分析します。
   - アプリケーションがクラッシュした場合、重要な情報はおそらくログの最後にあります。
   - 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。これを行うには、**tee** コマンドを使用します。

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

関連情報

- man ページの **ltrace (1)**

```
$ man ltrace
```

- Red Hat Developer Toolset ユーザーガイドの ltrace の章

3.3.4. SystemTap を使用したアプリケーションのシステムコールの監視

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** ツールと比較すると、使用は難しくなりますが、より効率的でより複雑な処理ロジックを使用できます。**strace.stp** と呼ばれる SystemTap スクリプトは SystemTap と共にインストールされ、SystemTap を使用して **strace** に類似の機能を提供します。

前提条件

- SystemTap と対応するカーネルパッケージがシステムにインストールされている必要がある。

手順

1. 監視するプロセスのプロセス ID (**pid**) を検索します。

```
$ ps -aux
```

2. **strace.stp** スクリプトで SystemTap を実行します。

```
# stap /usr/share/systemtap/examples/process/strace.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。

3. 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。

4. 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。

3.3.5. GDB を使用したアプリケーションのシステムコールの傍受

GNU デバッガー (GDB) により、プログラムの実行中に発生するさまざまな状況で実行を停止できます。プログラムがシステムコールを実行するときに実行を停止するには、GDB の **チェックポイント** を使用します。

前提条件

- GDB ブレークポイントの使用量を理解している。
- GDB がプログラムに割り当てられている。

手順

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コールを実行する際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。

syscall-name オプションは、コールの名前を指定します。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** オプションを指定しないと、システムコールで GDB が停止します。

2. プログラムの実行を開始します。
   - プログラムにより、実行が開始していない場合は開始します。

```
(gdb) r
```

   - プログラムの実行が停止した場合は、再開します。

```
(gdb) c
```

3. GDB は、プログラムが指定のシステムコールを実行した後に実行を停止します。

関連情報

- GDB を使用したデバッグ - Setting Watchpoints

3.3.6. GDB を使用したアプリケーションによるシグナル処理のインターセプト

GNU デバッガー (GDB) により、プログラムの実行中に発生するさまざまな状況で実行を停止できます。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するとき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 ブレークポイントの使用量を理解している。
- GDB がプログラムに割り当てられている。

手順

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. プログラムを実行します。

- プログラムにより、実行が開始していない場合は開始します。

```
(gdb) r
```

- プログラムの実行が停止した場合は、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 を使用したデバッグ - 5.1.3 Setting Catchpoints

3.4. 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。

3.4.1. コアダンプ: その概要と使用方法

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。現在の制限を表示するには、次のコマンドを実行します。

```
$ ulimit -a
```

3.4.2. コアダンプによる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するには、**/etc/systemd/system.conf** ファイルに以下の行が含まれていることを確認します。

```
DumpCore=yes
DefaultLimitCORE=infinity
```

   これらの設定が以前に存在したかどうか、以前の値が何であったかを説明するコメントを追加することもできます。これにより、必要に応じて、この変更を後で元に戻すことができます。コメントは、**#** 文字で始まる行です。

   ファイルを変更するには、管理者レベルのアクセスが必要です。

2. 新しい設定を適用します。

```
# systemctl daemon-reexec
```

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** ではなく、**0** を指定してコマンドを実行します。

4. システム情報を収集する **sosreport** ユーティリティーを提供する **sos** パッケージをインストールします。

```
# dnf install sos
```

5. 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、**systemd-coredump** により処理されます。

6. 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。

7. コアダンプを探してエクスポートします。

```
$ coredumpctl list executable-name
$ coredumpctl dump executable-name > /path/to/file-for-export
```

   アプリケーションが複数回クラッシュした場合、最初のコマンドの出力には、取得されたコアダンプがさらにリスト表示されます。その場合、2 番目のコマンドに対して、他の情報を使用してより正確なクエリーを作成します。詳細は、man ページ **coredumpctl(1)** を参照してください。

8. デバッグを行うコンピューターに、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、コアファイルのその後の分析で特定します。

9. オプション: コアダンプと SOS レポートに移動後に削除して、ディスク領域を解放します。

関連情報

- **基本的なシステム設定の設定** の systemd の概要
- 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 上での sosreport のロールと取得方法

3.4.3. コアダンプでアプリケーションのクラッシュ状態の検査

前提条件

- クラッシュが発生したシステムのコアダンプファイルと sosreport 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。

手順

1. クラッシュが発生した実行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リスト表示されます。

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。

- システムで同じ実行ファイルを使用することもできます。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. コアファイルを詳細に調べるには、GDB で実行ファイルとコアダンプファイルを読み込みます。

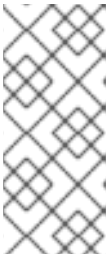
```
$ gdb -e executable_file -c core_file
```

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。

アプリケーションの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
```

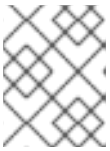
program.debug は、実際のファイル名に置き換えます。



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査します。GDB を使用したアプリケーションの内部状況の検証 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訳ではありません。実行を制御するコマンドは影響を受けません。

関連情報

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

3.4.4. coredumpctl を使用したコアダンプの作成およびアクセス

systemd の **coredumpctl** ツールは、クラッシュが発生したマシン上のコアダンプの処理を大幅に合理化できます。この手順では、応答しないプロセスのコアダンプを取得する方法を説明します。

前提条件

- システムは、コアダンプの処理に **systemd-coredump** を使用するように設定している。true かどうか確認するには、次のコマンドを実行します。

```
$ sysctl kernel.core_pattern
```

次の内容で出力が始まる場合は、設定が適切です。

```
kernel.core_pattern = |/usr/lib/systemd/systemd-coredump
```

手順

1. 実行ファイル名の既知の部分に基づいて、ハングしたプロセスの PID を検索します。

```
$ pgrep -a executable-name-fragment
```

このコマンドは、フォームの行を出力します。

```
PID command-line
```

command-line 値を使用して、PID が目的のプロセスに属することを確認します。

以下に例を示します。

```
$ pgrep -a bc
5459 bc
```

2. 中断シグナルをプロセスに送信します。

```
# kill -ABRT PID
```

3. コアが **coredumpctl** で取得されていることを確認します。

```
$ coredumpctl list PID
```

以下に例を示します。

```
$ coredumpctl list 5459
TIME                            PID   UID   GID SIG COREFILE  EXE
Thu 2019-11-07 15:14:46 CET    5459  1000  1000   6 present   /usr/bin/bc
```

4. 必要に応じて、コアファイルをさらに検証または使用します。
PID と他の値でコアダンプを指定できます。詳細は、man ページの **coredumpctl(1)** を参照してください。

- コアファイルの詳細を表示します。

```
$ coredumpctl info PID
```

- GDB デバッガーでコアファイルを読み込むには、次のコマンドを実行します。

```
$ coredumpctl debug PID
```

デバッグ情報の可用性によっては、GDB は次のようなコマンドを実行するコマンドを提案します。

```
Missing separate debuginfos, use: dnf debuginfo-install bc-1.07.1-5.el8.x86_64
```

このプロセスの詳細は、GDB を使用したアプリケーションまたはライブラリーの debuginfo パッケージの取得 を参照してください。

- その後の処理を別の場所でするためにコアファイルをエクスポートするには、次のコマンドを実行します。

```
$ coredumpctl dump PID > /path/to/file_for_export
```

/path/to/file_for_export を、コアダンプを配置するファイルに置き換えます。

3.4.5. 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。場合によっては、このプロセスの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できます。**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

- コアダンプの概要および作成方法を理解している。
- GDB がシステムにインストールされている。

手順

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 レポートに移動後に削除して、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 - How to obtain a core file without restarting an application?

3.4.6. 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、銀行、会計アプリケーション、または仮想マシン全体など、プロセスメモリーに機密データが含まれる場合は、リソースを節約し、セキュリティーを強化できます。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。

場合によっては、これらの保護に関係なく、プロセスメモリーの内容全体をダンプする必要があります。この手順では、GDB デバッガーを使用してこれを行う方法を説明します。

前提条件

- コアダンプとは何かを理解する必要がある。
- GDB がシステムにインストールされている。
- GDB は、メモリーが保護されているプロセスに割り当てられている。

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 を、メモリーをダンプするファイルの名前に置き換えます。

関連情報

- GDB を使用したデバッグ - How to Produce a Core File from Your Program

3.5. GDB で互換性に影響を与える変更

Red Hat Enterprise Linux 9 で提供される GDB のバージョンには、互換性に影響を与える変更が多数含まれています。次のセクションは、この変更の詳細を提供します。

コマンド

- **gdb -P python-script.py** コマンドはサポートされなくなりました。
  代わりに **gdb -ex 'source python-script.py'** コマンドを使用してください。

- **gdb COREFILE** コマンドはサポートされなくなりました。
  代わりに **gdb EXECUTABLE --core COREFILE** コマンドを使用して、コアファイルで指定されている実行可能ファイルをロードします。

- GDB がデフォルトで出力のスタイルを設定するようになりました。
  この新しい変更により、GDB の出力を解析しようとするスクリプトが中断される可能性があります。スクリプト内のスタイル設定を無効にするには、**gdb -ex 'set style Enabled off'** コマンドを使用します。

- コマンドが、言語に従ってシンボルの構文を定義するようになりました。
  info functions、**info types**、**info variables**、および **rbreak** コマンドが、**set language** コマンドで選択された言語に従ってエンティティーの構文を定義するようになりました。**set language auto** に設定すると、GDB が表示されたエンティティーの言語を自動的に選択します。

- **set print raw Frame-arguments** コマンドと **show print raw Frame-arguments** コマンドが非推奨になりました。
  これらのコマンドは、**set print raw-frame-arguments** コマンドと **show print raw-frame-arguments** コマンドに置き換えられます。古いコマンドは将来のバージョンで削除される可能性があります。

- 次の TUI コマンドで、大文字と小文字が区別されるようになりました。
  - **focus**
  - **winheight**

- **+**
- **-**
- **>**
- **<**

- **help** および **apropos** コマンドが、コマンド情報を 1 回だけ表示するようになりました。
これらのコマンドは、コマンドに 1 つ以上のエイリアスがある場合でも、コマンドのドキュメントを 1 回だけ表示するようになりました。これらのコマンドでは、コマンド名、そのすべてのエイリアス、コマンドの説明が順に表示されます。

MI インタープリター

- MI インタープリターのデフォルトバージョンが 3 になりました。
複数の場所のブレークポイントに関する情報の出力 (MI 2 では構文的に正しくありません) が MI 3 で変更されました。これは次のコマンドとイベントに影響します。
  - **-break-insert**
  - **-break-info**
  - **=breakpoint-created**
  - **=breakpoint-modified**

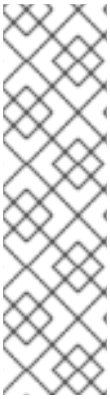
  以前の MI バージョンでこの動作を有効にするには、**-fix-multi-location-breakpoint-output** コマンドを使用します。

Python API

- 次のシンボルが非推奨になりました。
  - **gdb.SYMBOL_VARIABLES_DOMAIN**
  - **gdb.SYMBOL_FUNCTIONS_DOMAIN**
  - **gdb.SYMBOL_TYPES_DOMAIN**
- **gdb.Value** 型に新しいコンストラクターが追加されました。これは、Python バッファーオブジェクトと **gdb.Type** から **gdb.Value** を構築するために使用します。
- Python フレームフィルタリングコードによって出力されるフレーム情報が、フィルターがない場合、または **backtrace** コマンドの **-no-filters** オプションを使用している場合の **backtrace** コマンドの出力内容と一致するようになりました。

3.6. コンテナー内のアプリケーションのデバッグ

トラブルシューティングのさまざまな側面に合わせてカスタマイズされたさまざまなコマンドラインツールを使用できます。以下に、一般的なコマンドラインツールとともにカテゴリーを示します。



注記

これはコマンドラインツールの完全なリストではありません。コンテナーアプリケーションをデバッグするためのツールの選択は、コンテナーイメージとユースケースに大きく依存します。

たとえば、**systemctl**、**journalctl**、**ip**、**netstat**、**ping**、**traceroute**、**perf**、**iostat** ツールは、ネットワーク、systemd サービス、ハードウェアパフォーマンスカウンターなどのシステムレベルのリソースと対話するため、ルートアクセスが必要になる場合があります。これらのリソースは、セキュリティー上の理由から、ルートレスコンテナーでは制限されています。

ルートレスコンテナーは昇格された権限を必要とせずに動作し、ユーザー名前空間内で非ルートユーザーとして実行されるため、セキュリティーが向上し、ホストシステムから分離されます。ホストとのやり取りが制限され、攻撃対象領域が縮小され、権限昇格の脆弱性のリスクが軽減されるため、セキュリティーが強化されます。

ルートフルコンテナーは、通常はルートユーザーとして昇格された権限で実行され、システムリソースと機能への完全なアクセス権が付与されます。ルートフルコンテナーは柔軟性と制御性に優れていますが、権限昇格の可能性があり、ホストシステムが脆弱性にさらされる可能性があるため、セキュリティーリスクが生じます。

ルートフルコンテナーとルートレスコンテナーの詳細は、ルートレスコンテナーの設定、ルートレスコンテナーへのアップグレード、および ルートレスコンテナーに関する特別な考慮事項 を参照してください。

Systemd とプロセス管理ツール

systemctl
: コンテナー内の systemd サービスを制御し、開始、停止、有効化、無効化の操作を可能にします。

journalctl
: systemd サービスによって生成されたログを表示し、コンテナーの問題のトラブルシューティングに役立ちます。

ネットワークツール

ip
: コンテナー内のネットワークインターフェイス、ルーティング、およびアドレスを管理します。

netstat
: ネットワーク接続、ルーティングテーブル、およびインターフェイス統計を表示します。

ping
: コンテナーまたはホスト間のネットワーク接続を検証します。

traceroute
: パケットが宛先に到達するまでのパスを識別します。ネットワークの問題の診断に役立ちます。

プロセスとパフォーマンスツール

ps
: コンテナー内で現在実行中のプロセスをリスト表示します。

top
: コンテナー内のプロセスによるリソース使用状況に関するリアルタイムの分析情報を提供します。

htop

リソース使用率を監視するためのインタラクティブなプロセスビューアー。

perf

CPU パフォーマンスのプロファイリング、トレース、およびモニタリングにより、システムまたはアプリケーション内のパフォーマンスのボトルネックを正確に特定できます。

vmstat

コンテナー内の仮想メモリーの統計を報告し、パフォーマンス分析に役立ちます。

iostat

コンテナー内のブロックデバイスの入出力統計を監視します。

gdb (GNU デバッガー)

ユーザーが実行を追跡および制御し、変数を検査し、実行時にメモリーとレジスターを分析できるようにすることで、プログラムの調査とデバッグを支援するコマンドラインデバッガー。詳細は、Red Hat OpenShift コンテナー内のアプリケーションのデバッグの を参照してください。

strace

プログラムによって行われたシステムコールを傍受して記録し、プログラムとオペレーティングシステム間のやり取りを明らかにすることでトラブルシューティングに役立ちます。

セキュリティーとアクセス制御ツール

sudo

昇格された権限でコマンドを実行できるようにします。

chroot

コマンドのルートディレクトリーを変更します。異なるルートディレクトリー内でのテストやトラブルシューティングに役立ちます。

Podman 固有のツール

podman logs

実行時に 1 つ以上のコンテナーに存在するログをバッチで取得します。

podman inspect

名前または ID で識別されるコンテナーとイメージの低レベル情報を表示します。

podman events

Podman で発生するイベントを監視して出力します。各イベントには、タイムスタンプ、タイプ、ステータス、名前 (該当する場合)、およびイメージ (該当する場合) が含まれます。デフォルトのロギングメカニズムは **journald** です。

podman run --health-cmd

ヘルスチェックを使用して、コンテナー内で実行されているプロセスの状態または準備ができているかどうかを判断できます。

podman top

コンテナーの実行中のプロセスを表示します。

podman exec

実行中のコンテナー内でコマンドを実行したり、実行中のコンテナーにアタッチしたりすることは、コンテナー内で何が起こっているかをよりよく理解するのに非常に役立ちます。

podman export

コンテナーに障害が発生すると、何が起こったのかを知ることは基本的に不可能です。コンテナーからファイルシステム構造をエクスポートすると、マウントされたボリュームに存在しない可能性のある他のログファイルを確認できるようになります。

関連情報

- Red Hat OpenShift コンテナー内のアプリケーションのデバッグ
  - **gdb**
- クラッシュしたアプリケーションのデバッグ
  - コアダンプ、**sosreport**、**gdb**、**ps**、**core**。
- Kubernetes のトラブルシューティング
  - Docker exec + env、**netstat**、**kubectl**、**etcdctl**、**journalctl**、docker ログ
- コンテナー化サービスに関するヒントとコツ
  - ウォッチ、**podman logs**、**systemctl**、**podman exec/kill/restart**、**podman insect**、**podman top**、**podman exec**、**podman export**、**paunch**
- 外部リンク
  - Docker コンテナーをデバッグするための 10 のヒント

第4章 開発用の追加ツールセット

4.1. GCC TOOLSET の使用

4.1.1. GCC Toolset とは

Red Hat Enterprise Linux 9 は、開発およびパフォーマンス分析ツールの最新バージョンを含むアプリケーションストリームである GCC Toolset のサポートを継続します。GCC Toolset は、RHEL 7 の Red Hat Developer Toolset に類似したツールセットです。

GCC Toolset は、**AppStream** リポジトリーにおいて、Software Collection の形式で、Application Stream として利用できます。GCC Toolset は、Red Hat Enterprise Linux サブスクリプション契約で完全にサポートされており、機能的に完全で、実稼働環境での使用を対象としています。GCC Toolset が提供するアプリケーションおよびライブラリーは、Red Hat Enterprise Linux システムのバージョンを置き換えず、上書きせず、自動的にデフォルトまたは推奨される選択肢になるわけではありません。Software Collection というフレームワークを使用すると、追加の開発者ツールセットが **/opt/** ディレクトリーにインストールされ、ユーザーが **scl** ユーティリティーを使用してオンデマンドで明示的に有効にします。特定のツールや機能について特に記載がない限り、Red Hat Enterprise Linux が対応するすべてのアーキテクチャーで GCC Toolset が利用できます。

4.1.2. GCC Toolset のインストール

システムに GCC Toolset をインストールすると、メインのツールと、必要な依存関係がすべてインストールされます。ツールセットの一部はデフォルトではインストールされず、個別にインストールする必要があります。

手順

- GCC Toolset バージョン **N** をインストールするには、次のコマンドを実行します。

```
# dnf install gcc-toolset-N
```

4.1.3. GCC Toolset からの個別パッケージのインストール

ツールセット全体ではなく、GCC Toolset から特定のツールのみをインストールするには、利用可能なパッケージの一覧を表示し、**dnf** パッケージ管理ツールで選択したツールをインストールします。この手順では、デフォルトではツールセットでインストールされていないパッケージにも便利です。

手順

1. GCC Toolset バージョン **N** で利用可能なパッケージのリストを表示します。

```
$ dnf list available gcc-toolset-N-\*
```

2. このパッケージのいずれかをインストールするには、次のコマンドを実行します。

```
# dnf install package_name
```

package_name を、インストールするパッケージのリストに置き換えます。パッケージ名はスペースで区切られます。たとえば、**gcc-toolset-9-gdb-gdbserver** パッケージおよび **gcc-toolset-9-gdb-doc** パッケージをインストールするには、次のコマンドを実行します。

```
# dnf install gcc-toolset-9-gdb-gdbserver gcc-toolset-9-gdb-doc
```

4.1.4. GCC Toolset のアンインストール

システムから GCC Toolset を削除するには、**dnf** パッケージ管理ツールを使用してアンインストールします。

手順

- GCC Toolset バージョン **N** をアンインストールするには、次のコマンドを実行します。

```
# dnf remove gcc-toolset-N\*
```

4.1.5. GCC Toolset のツールの実行

GCC Toolset のツールを実行するには、**scl** ユーティリティーを使用します。

手順

- GCC Toolset バージョン **N** のツールを実行するには、次のコマンドを実行します。

```
$ scl enable gcc-toolset-N tool
```

4.1.6. GCC Toolset でシェルセッションの実行

GCC Toolset では、**scl** コマンドを明示的に使用せずに、このようなツールのシステムバージョンの代わりに、GCC Toolset ツールのバージョンを使用しているシェルセッションを実行できます。これは、開発設定のセットアップ時またはテスト時など、ツールを何度もインタラクティブに起動する必要がある場合に便利です。

手順

- GCC Toolset バージョン **N** のツールバージョンが、このようなツールのシステムバージョンをオーバーライドするシェルセッションを実行するには、次のコマンドを実行します。

```
$ scl enable gcc-toolset-N bash
```

4.1.7. 関連情報

- Red Hat Developer Toolset User Guide

4.2. GCC TOOLSET 12

GCC Toolset バージョン 12 とこのバージョンに含まれるツールに固有の情報について説明します。

4.2.1. GCC Toolset 12 が提供するツールおよびバージョン

GCC Toolset 12 は、以下のツールおよびバージョンを提供します。

表4.1 GCC Toolset 12 のツールバージョン

名前	バージョン	説明
GCC	12.1.1	C、C++、および Fortran に対応するポータブルなコンパイラースイート。
GDB	11.2	C、C++、および Fortran で記述されたプログラムのコマンドラインデバッガー。
binutils	2.38	オブジェクトファイルおよびバイナリーを検査および操作するためのバイナリーツールおよびその他のユーティリティーのコレクション。
dwz	0.14	ELF 共有ライブラリーおよび ELF 実行ファイルに含まれる DWARF デバッグ情報 (サイズ) を最適化するツール。
annobin	10.76	ビルドセキュリティーチェックツール。

4.2.2. GCC Toolset 12 での C++ 互換性

重要

ここで示されている互換性情報は、GCC Toolset 12 の GCC にのみ適用されます。

GCC Toolset の GCC コンパイラーは、以下の C++ 規格を使用できます。

C++14

この言語の規格は、GCC Toolset 12 で利用できます。
適切なフラグでコンパイルされた C++ オブジェクトがすべて、GCC バージョン 6 以降を使用してビルドされている場合は、C++14 言語バージョンの使用に対応します。

C++11

この言語の規格は、GCC Toolset 12 で利用できます。
適切なフラグでコンパイルされた C++ オブジェクトがすべて、GCC バージョン 5 以降を使用してビルドされている場合は、C++11 言語バージョンの使用に対応しています。

C++98

この言語の規格は、GCC Toolset 12 で利用できます。この規格を使用して構築されたバイナリー、共有ライブラリー、およびオブジェクトは、GCC Toolset、Red Hat Developer Toolset、ならびに RHEL 5、6、7、および 8 の GCC でビルドされているかどうかにかかわらず、自由に組み合わせることができます。

C++17

この言語の規格は、GCC Toolset 12 で利用できます。
これは、GCC Toolset 12 のデフォルトの言語標準設定で、GNU 拡張機能は、**-std=gnu++17** オプションを明示的に使用するのと同じです。

適切なフラグでコンパイルされた C++ オブジェクトがすべて、GCC バージョン 10 以降を使用してビルドされている場合は、C++17 言語バージョンの使用に対応しています。

C++20 および C++23

このような言語の規格は、GCC Toolset 12 では実験的で、不安定な、サポート対象外の機能としてのみ利用できます。さらに、この規格を使用して構築されたオブジェクト、バイナリーファイル、およびライブラリーの互換性は保証できません。
C++20 サポートを有効にするには、コマンドラインオプション **-std=c++20** を g++ コマンドラインに追加します。

C++23 サポートを有効にするには、コマンドラインオプション **-std=c++23** を g++ コマンドラインに追加します。

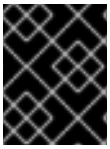
すべての言語規格は、規格に準拠したバリアントまたは GNU 拡張機能の両方で利用できます。

GCC Toolset で構築されたオブジェクトを、RHEL ツールチェーン (特に **.o** ファイルまたは **..a** ファイル) で構築したオブジェクトと混在する場合、GCC Toolset ツールチェーンはどの連携にも使用する必要があります。これにより、GCC Toolset が提供する新しいライブラリー機能は、リンク時に解決されます。

4.2.3. GCC Toolset 12 での GCC の詳細

ライブラリーの静的リンク

最新のライブラリー機能の一部は、複数のバージョンの Red Hat Enterprise Linux での実行に対応するために、GCC Toolset で構築されたアプリケーションに静的にリンクされています。標準の Red Hat Enterprise Linux エラータではこのコードが変更されないため、これにより、若干のセキュリティーリスクが発生します。Red Hat は、このリスクにより、開発者がアプリケーションを再構築する必要がある場合でも、セキュリティーエラータを使用してこのアプリケーションと通信します。



重要

このようなセキュリティーリスクが発生するため、開発者は同じ理由によりアプリケーション全体を静的にリンクしないことが強く推奨されます。

連結時に、オブジェクトファイルの後にライブラリーを指定

GCC Toolset では、ライブラリーは、静的アーカイブで一部のシンボルを指定できるリンカースクリプトを使用してリンクされます。これは、Red Hat Enterprise Linux の複数のバージョンとの互換性を確保するために必要になります。ただし、リンカーのスクリプトは、対応する共有オブジェクトファイルの名前を使用します。したがって、リンカーは、オブジェクトファイルを指定するオプションの前に、ライブラリーを追加するオプションを指定する際に、想定とは異なるシンボル処理ルールを使用して、オブジェクトファイルが必要とするシンボルを認識しません。

```
$ scl enable gcc-toolset-12 'gcc -lsomelib objfile.o'
```

この方法で GCC Toolset のライブラリーを使用すると、リンカーのエラーメッセージで、**シンボルの参照が未定義** になります。この問題を回避するには、標準のリンクプラクティスに従い、オブジェクトファイルを指定するオプションの後に、ライブラリーを追加するオプションを指定します。

```
$ scl enable gcc-toolset-12 'gcc objfile.o -lsomelib'
```

この推奨事項は、Red Hat Enterprise Linux のベースバージョンの **GCC** を使用する場合にも適用されることに注意してください。

4.2.4. GCC Toolset 12 における binutils の詳細

ライブラリーの静的リンク

最新のライブラリー機能の一部は、複数のバージョンの Red Hat Enterprise Linux での実行に対応するために、GCC Toolset で構築されたアプリケーションに静的にリンクされています。標準の Red Hat Enterprise Linux エラータではこのコードが変更されないため、これにより、若干のセキュリティーリスクが発生します。Red Hat は、このリスクにより、開発者がアプリケーションを再構築する必要がある場合でも、セキュリティーエラータを使用してこのアプリケーションと通信します。

重要

このようなセキュリティーリスクが発生するため、開発者は同じ理由によりアプリケーション全体を静的にリンクしないことが強く推奨されます。

連結時に、オブジェクトファイルの後にライブラリーを指定

GCC Toolset では、ライブラリーは、静的アーカイブで一部のシンボルを指定できるリンカースクリプトを使用してリンクされます。これは、Red Hat Enterprise Linux の複数のバージョンとの互換性を確保するために必要になります。ただし、リンカーのスクリプトは、対応する共有オブジェクトファイルの名前を使用します。したがって、リンカーは、オブジェクトファイルを指定するオプションの前に、ライブラリーを追加するオプションを指定する際に、想定とは異なるシンボル処理ルールを使用して、オブジェクトファイルが必要とするシンボルを認識しません。

```
$ scl enable gcc-toolset-12 'ld -lsomelib objfile.o'
```

この方法で GCC Toolset のライブラリーを使用すると、リンカーのエラーメッセージで、**シンボルの参照が未定義** になります。この問題を回避するには、標準のリンクプラクティスに従い、オブジェクトファイルを指定するオプションの後に、ライブラリーを追加するオプションを指定します。

```
$ scl enable gcc-toolset-12 'ld objfile.o -lsomelib'
```

また、この推奨事項は、Red Hat Enterprise Linux のベースバージョンの **binutils** を使用している場合にも適用されることに注意してください。

4.2.5. GCC Toolset 12 での annobin の詳細

場合によっては、GCC Toolset 12 における **annobin** と **gcc** 間の同期問題により、コンパイルが失敗し、次のようなエラーメッセージが表示されることがあります。

```
cc1: fatal error: inaccessible plugin file
opt/rh/gcc-toolset-12/root/usr/lib/gcc/architecture-linux-gnu/12/plugin/gcc-annobin.so
expanded from short plugin name gcc-annobin: No such file or directory
```

この問題を回避するには、プラグインディレクトリーに **annobin.so** ファイルから **gcc-annobin.so** ファイルへのシンボリックリンクを作成します。

```
# cd /opt/rh/gcc-toolset-12/root/usr/lib/gcc/architecture-linux-gnu/12/plugin
# ln -s annobin.so gcc-annobin.so
```

architecture はシステムでお使いのアーキテクチャーに置き換えてください。

- **aarch64**
- **i686**

- **ppc64le**
- **s390x**
- **x86_64**

4.3. GCC TOOLSET 13

GCC Toolset バージョン 13 とこのバージョンに含まれるツールに固有の情報について説明します。

4.3.1. GCC Toolset 13 が提供するツールおよびバージョン

GCC Toolset 13 は、以下のツールおよびバージョンを提供します。

表4.2 GCC Toolset 13 のツールバージョン

名前	バージョン	説明
GCC	13.1.1	C、C++、および Fortran に対応するポータブルなコンパイラースイート。
GDB	12.1	C、C++、および Fortran で記述されたプログラムのコマンドラインデバッガー。
binutils	2.40	オブジェクトファイルおよびバイナリーを検査および操作するためのバイナリーツールおよびその他のユーティリティーのコレクション。
dwz	0.14	ELF 共有ライブラリーおよび ELF 実行ファイルに含まれる DWARF デバッグ情報 (サイズ) を最適化するツール。
annobin	12.20	ビルドセキュリティーチェックツール。

4.3.2. GCC Toolset 13 の C++ 互換性

重要

ここで示されている互換性情報は、GCC Toolset 13 の GCC にのみ適用されます。

GCC Toolset の GCC コンパイラーは、以下の C++ 規格を使用できます。

C++14

この言語の規格は、GCC Toolset 13 で利用できます。
適切なフラグでコンパイルされた C++ オブジェクトがすべて、GCC バージョン 6 以降を使用してビルドされている場合は、C++14 言語バージョンの使用に対応します。

C++11

この言語の規格は、GCC Toolset 13 で利用できます。
適切なフラグでコンパイルされた C++ オブジェクトがすべて、GCC バージョン 5 以降を使用してビルドされている場合は、C++11 言語バージョンの使用に対応しています。

C++98

この言語の規格は、GCC Toolset 13 で利用できます。この規格を使用して構築されたバイナリー、共有ライブラリー、およびオブジェクトは、GCC Toolset、Red Hat Developer Toolset、ならびに RHEL 5、6、7、および 8 の GCC でビルドされているかどうかにかかわらず、自由に組み合わせることができます。

C++17

この言語の規格は、GCC Toolset 13 で利用できます。
これは、GCC Toolset 13 のデフォルトの言語規格設定で、GNU 拡張機能は、**-std=gnu++17** オプションを明示的に使用するのと同じです。

適切なフラグでコンパイルされた C++ オブジェクトがすべて、GCC バージョン 10 以降を使用してビルドされている場合は、C++17 言語バージョンの使用に対応しています。

C++20 および C++23

このような言語の規格は、GCC Toolset 13 では実験的で、不安定な、サポート対象外の機能としてのみ利用できます。さらに、この規格を使用して構築されたオブジェクト、バイナリーファイル、およびライブラリーの互換性は保証できません。
C++20 規格を有効にするには、コマンドラインオプション **-std=c++20** を g++ コマンドラインに追加します。

C++23 規格を有効にするには、コマンドラインオプション **-std=c++23** を g++ コマンドラインに追加します。

すべての言語規格は、規格に準拠したバリアントまたは GNU 拡張機能の両方で利用できます。

GCC Toolset で構築されたオブジェクトを、RHEL ツールチェーン (特に **.o** ファイルまたは **..a** ファイル) で構築したオブジェクトと混在する場合、GCC Toolset ツールチェーンはどの連携にも使用する必要があります。これにより、GCC Toolset が提供する新しいライブラリー機能は、リンク時に解決されます。

4.3.3. GCC Toolset 13 の GCC の詳細

ライブラリーの静的リンク

最新のライブラリー機能の一部は、複数のバージョンの Red Hat Enterprise Linux での実行に対応するために、GCC Toolset で構築されたアプリケーションに静的にリンクされています。標準の Red Hat Enterprise Linux エラータではこのコードが変更されないため、これにより、若干のセキュリティーリスクが発生します。Red Hat は、このリスクにより、開発者がアプリケーションを再構築する必要がある場合でも、セキュリティーエラータを使用してこのアプリケーションと通信します。

重要

このようなセキュリティーリスクが発生するため、開発者は同じ理由によりアプリケーション全体を静的にリンクしないことが強く推奨されます。

連結時に、オブジェクトファイルの後にライブラリーを指定

GCC Toolset では、ライブラリーは、静的アーカイブで一部のシンボルを指定できるリンカースクリプトを使用してリンクされます。これは、Red Hat Enterprise Linux の複数のバージョンとの互換性を確保するために必要になります。ただし、リンカーのスクリプトは、対応する共有オブジェクトファイルの名前を使用します。したがって、リンカーは、オブジェクトファイルを指定するオプションの前に、ライブラリーを追加するオプションを指定する際に、想定とは異なるシンボル処理ルールを使用して、オブジェクトファイルが必要とするシンボルを認識しません。

```
$ scl enable gcc-toolset-13 'gcc -lsomelib objfile.o'
```

この方法で GCC Toolset のライブラリーを使用すると、リンカーのエラーメッセージで、**シンボルの参照が未定義** になります。この問題を回避するには、標準のリンクプラクティスに従い、オブジェクトファイルを指定するオプションの後に、ライブラリーを追加するオプションを指定します。

```
$ scl enable gcc-toolset-13 'gcc objfile.o -lsomelib'
```

この推奨事項は、Red Hat Enterprise Linux のベースバージョンの **GCC** を使用する場合にも適用されることに注意してください。

4.3.4. GCC Toolset 13 の binutils の詳細

ライブラリーの静的リンク

最新のライブラリー機能の一部は、複数のバージョンの Red Hat Enterprise Linux での実行に対応するために、GCC Toolset で構築されたアプリケーションに静的にリンクされています。標準の Red Hat Enterprise Linux エラータではこのコードが変更されないため、これにより、若干のセキュリティーリスクが発生します。Red Hat は、このリスクにより、開発者がアプリケーションを再構築する必要がある場合でも、セキュリティーエラータを使用してこのアプリケーションと通信します。

重要

このようなセキュリティーリスクが発生するため、開発者は同じ理由によりアプリケーション全体を静的にリンクしないことが強く推奨されます。

連結時に、オブジェクトファイルの後にライブラリーを指定

GCC Toolset では、ライブラリーは、静的アーカイブで一部のシンボルを指定できるリンカースクリプトを使用してリンクされます。これは、Red Hat Enterprise Linux の複数のバージョンとの互換性を確保するために必要になります。ただし、リンカーのスクリプトは、対応する共有オブジェクトファイルの名前を使用します。したがって、リンカーは、オブジェクトファイルを指定するオプションの前に、ライブラリーを追加するオプションを指定する際に、想定とは異なるシンボル処理ルールを使用して、オブジェクトファイルが必要とするシンボルを認識しません。

```
$ scl enable gcc-toolset-13 'ld -lsomelib objfile.o'
```

この方法で GCC Toolset のライブラリーを使用すると、リンカーのエラーメッセージで、**シンボルの参照が未定義** になります。この問題を回避するには、標準のリンクプラクティスに従い、オブジェクトファイルを指定するオプションの後に、ライブラリーを追加するオプションを指定します。

```
$ scl enable gcc-toolset-13 'ld objfile.o -lsomelib'
```

また、この推奨事項は、Red Hat Enterprise Linux のベースバージョンの **binutils** を使用している場合にも適用されることに注意してください。

4.3.5. GCC Toolset 13 の annobin の詳細

状況によっては、GCC Toolset 13 の **annobin** と **gcc** 間の同期の問題が原因で、コンパイルが失敗し、次のようなエラーメッセージが表示されることがあります。

```
cc1: fatal error: inaccessible plugin file
opt/rh/gcc-toolset-13/root/usr/lib/gcc/architecture-linux-gnu/13/plugin/gcc-annobin.so
expanded from short plugin name gcc-annobin: No such file or directory
```

この問題を回避するには、プラグインディレクトリーに **annobin.so** ファイルから **gcc-annobin.so** ファイルへのシンボリックリンクを作成します。

```
# cd /opt/rh/gcc-toolset-13/root/usr/lib/gcc/architecture-linux-gnu/13/plugin
# ln -s annobin.so gcc-annobin.so
```

architecture はシステムでお使いのアーキテクチャーに置き換えてください。

- **aarch64**
- **i686**
- **ppc64le**
- **s390x**
- **x86_64**

4.4. GCC TOOLSET コンテナーイメージの使用

GCC Toolset 13 コンテナーイメージ **のみ** がサポートされています。以前のバージョンの GCC Toolset コンテナーイメージが非推奨になりました。

GCC Toolset 13 コンポーネントは、**GCC Toolset 13 Toolchain** コンテナーイメージで利用できます。

GCC Toolset コンテナーイメージは **rhel9** ベースイメージに基づいており、RHEL 9 でサポートされるすべてのアーキテクチャーで使用できます。

- AMD アーキテクチャーおよび Intel 64 ビットアーキテクチャー
- 64 ビット ARM アーキテクチャー
- IBM Power Systems (リトルエンディアン)
- 64 ビット IBM Z

4.4.1. GCC Toolset コンテナーイメージの内容

GCC Toolset 13 コンテナーイメージで提供されるツールバージョンは、GCC Toolset 13 のコンポーネントバージョン と同じです。

GCC Toolset 13 Toolchain のコンテンツ

rhel9/gcc-toolset-13-toolchain イメージには、GCC コンパイラー、GDB デバッガー、その他の開発関連ツールが含まれます。コンテナーイメージは、以下のコンポーネントで設定されています。

コンポーネント	パッケージ
gcc	gcc-toolset-13-gcc

コンポーネント	パッケージ
g++	gcc-toolset-13-gcc-c++
gfortran	gcc-toolset-13-gcc-gfortran
gdb	gcc-toolset-13-gdb

4.4.2. GCC Toolset コンテナーイメージへのアクセスおよび実行

次のセクションでは、GCC Toolset コンテナーイメージにアクセスして実行する方法を説明します。

前提条件

- Podman がインストールされている。

手順

1. カスタマーポータルの認証情報を使用して Red Hat コンテナーレジストリー にアクセスします。

```
$ podman login registry.redhat.io
Username: username
Password: ********
```

2. root で適切なコマンドを実行して、必要なコンテナーイメージをプルします。

```
# podman pull registry.redhat.io/rhel9/gcc-toolset-13-toolchain
```

注記

root 以外のユーザーとしてコンテナーを操作するようにシステムをセットアップすることもできます。詳細は、ルートレスコンテナーのセットアップ を参照してください。

3. オプション: ローカルシステムにあるすべてのコンテナーイメージをリスト表示するコマンドを実行して、プルが正常に完了したことを確認します。

```
# podman images
```

4. コンテナー内で bash シェルを起動して、コンテナーを実行します。

```
# podman run -it image_name /bin/bash
```

-i オプションは対話式のセッションを作成します。このオプションを指定しないと、シェルが開き、即座に終了します。

-t オプションは端末セッションを開きます。このオプションがないと、シェルに何も入力できません。

関連情報

関連情報

- RHEL 9 での Linux コンテナーの構築、実行、および管理
- Red Hat ブログ記事:Understanding root in a container
- Red Hat Container Registry のエントリー: GCC Toolset コンテナーイメージ

4.4.3. 以下に例を示します。GCC Toolset 13 ツールチェーンコンテナーイメージの使用

この例では、GCC Toolset 13 Toolchain コンテナーイメージを取得して使用を開始する方法を示します。

前提条件

- Podman がインストールされている。

手順

1. カスタマーポータルの認証情報を使用して Red Hat コンテナーレジストリーにアクセスします。

```
$ podman login registry.redhat.io
Username: username
Password: ********
```

2. root でコンテナーイメージをプルします。

```
# podman pull registry.redhat.io/rhel9/gcc-toolset-13-toolchain
```

3. root で対話式シェルを使用してコンテナーイメージを起動します。

```
# podman run -it registry.redhat.io/rhel9/gcc-toolset-13-toolchain /bin/bash
```

4. GCC Toolset ツールを想定どおりに実行します。たとえば、**gcc** コンパイラーのバージョンを確認するには、次のコマンドを実行します。

```
bash-4.4$ gcc -v
...
gcc version 13.1.1 20231102 (Red Hat 13.1.1-4) (GCC)
```

5. コンテナーで提供されるパッケージのリストを表示するには、以下を実行します。

```
bash-4.4$ rpm -qa
```

4.5. コンパイラーツールセット

RHEL 9 は、以下のコンパイラーツールセットを、アプリケーションストリームとして提供します。

- LLVM ツールは、LLVM コンパイラーインフラストラクチャーフレームワーク、C 言語および C++ 言語用の Clang コンパイラー、LLDB デバッガー、コード解析の関連ツールを提供します。

- Rust Toolset は、Rust プログラミング言語コンパイラー **rustc**、**cargo** ビルドツールおよび依存マネージャー、**cargo-vendor** プラグイン、および必要なライブラリーを提供します。
- Go Toolset は、Go プログラミング言語ツールおよびライブラリーを提供します。Go は、**golang** としても知られています。

使用方法の詳細と情報については、Red Hat DeveloperTools ページのコンパイラーツールセットのユーザーガイドを参照してください。

4.6. ANNOBIN プロジェクト

Annobin プロジェクトは Watermark 仕様プロジェクトの実装です。Watermark 仕様プロジェクトは、マーカーを Executable and Linkable Format (ELF) オブジェクトに追加してそのプロパティーを判断するためのものです。Annobin プロジェクトは、**annobin** プラグインと **annockeck** プログラムで設定されます。

annobin プラグインは、GNU コンパイラーコレクション (GCC) コマンドライン、コンパイル状態、およびコンパイルプロセスをスキャンし、ELF ノートを生成します。ELF ノートでは、バイナリーの構築方法を記録し、セキュリティー強化チェックを実行する **annocheck** プログラムの情報を得ることができます。

セキュリティー強化チェッカーは、**annocheck** プログラムの一部で、デフォルトで有効になっています。バイナリーファイルをチェックして、必要なセキュリティー強化オプションでプログラムがビルドされたかどうかを判断し、適切にコンパイルします。**annocheck** は、ELF オブジェクトファイルのディレクトリー、アーカイブ、および RPM パッケージを再帰的にスキャンできます。

注記

ファイルは ELF 形式でなければなりません。**annocheck** は、他のバイナリーファイルタイプを処理しません。

次のセクションでは、以下を行う方法を説明します。

- **annobin** プラグインの使用
- **annocheck** プログラムの使用
- 冗長な **annobin** ノートの削除

4.6.1. annobin プラグインの使用

次のセクションでは、以下を行う方法を説明します。

- **annobin** プラグインの有効化
- **annobin** プラグインにオプションを渡します。

4.6.1.1. annobin プラグインの有効化

次のセクションでは、**gcc** および **clang** を使用して **annobin** プラグインを有効にする方法を説明します。

手順

- **gcc** で **annobin** プラグインを有効にするには、以下を使用します。

```
$ gcc -fplugin=annobin
```

- **gcc** で **annobin** プラグインを見つることができない場合は、以下を使用します。

```
$ gcc -iplugindir=/path/to/directory/containing/annobin/
```

/path/to/directory/containing/annobin/ は、**annobin** を含むディレクトリーへの絶対パスに置き換えます。

- **annobin** プラグインを含むディレクトリーを検索するには、以下を使用します。

```
$ gcc --print-file-name=plugin
```

- **clang** で **annobin** プラグインを有効にするには、以下を使用します。

```
$ clang -fplugin=/path/to/directory/containing/annobin/
```

/path/to/directory/containing/annobin/ は、**annobin** を含むディレクトリーへの絶対パスに置き換えます。

4.6.1.2. annobin プラグインへのオプションの指定

次のセクションでは、**gcc** および **clang** を使用して **annobin** プラグインにオプションを渡す方法を説明します。

手順

- **gcc** を使用して **annobin** プラグインにオプションを渡すには、以下を使用します。

```
$ gcc -fplugin=annobin -fplugin-arg-annobin-option file-name
```

option は **annobin** コマンドライン引数に、file-name はファイル名に置き換えます。

例

- 実行する **annobin** に関する追加情報を表示するには、以下を使用します。

```
$ gcc -fplugin=annobin -fplugin-arg-annobin-verbose file-name
```

file-name は、ファイルの名前に置き換えます。

- **clang** を使用して **annobin** プラグインにオプションを渡すには、以下を使用します。

```
$ clang -fplugin=/path/to/directory/containing/annobin/ -Xclang -plugin-arg-annobin -Xclang option file-name
```

option は **annobin** コマンドライン引数に、/path/to/directory/containing/annobin/ は、**annobin** を含むディレクトリーへの絶対パスに置き換えます。

例

- 実行する **annobin** に関する追加情報を表示するには、以下を使用します。

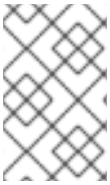
```
$ clang -fplugin=/usr/lib64/clang/10/lib/annobin.so -Xclang -plugin-arg-annobin -Xclang
verbose file-name
```

file-name は、ファイルの名前に置き換えます。

4.6.2. annocheck プログラムの使用

次のセクションでは、**annocheck** を使用して検証する方法を説明します。

- ファイル
- ディレクトリー
- RPM パッケージ
- **annocheck** の追加ツール



注記

annocheck は、ELF オブジェクトファイルのディレクトリー、アーカイブ、RPM パッケージを再帰的にスキャンします。ファイルは ELF 形式でなければなりません。**annocheck** は、他のバイナリーファイルタイプを処理しません。

4.6.2.1. annocheck を使用したファイルの検証

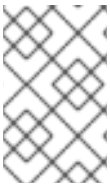
次のセクションでは、**annocheck** を使用して ELF ファイルを検証する方法を説明します。

手順

- ファイルを検証するには、以下を使用します。

```
$ annocheck file-name
```

file-name は、ファイルの名前に置き換えます。



注記

ファイルは ELF 形式でなければなりません。**annocheck** は、他のバイナリーファイルタイプを処理しません。**annocheck** は、ELF オブジェクトファイルを含む静的ライブラリーを処理します。

関連情報

- **annocheck** および使用可能なコマンドラインオプションに関する情報は、**annocheck** の man ページを参照してください。

4.6.2.2. annocheck を使用したディレクトリーの検証

次のセクションでは、**annocheck** を使用してディレクトリー内の ELF ファイルを検証する方法を説明します。

手順

- ディレクトリーをスキャンするには、以下を使用します。

```
$ annocheck directory-name
```

directory-name をディレクトリー名に置き換えます。**annocheck** はディレクトリーの内容、そのサブディレクトリー、およびディレクトリー内のアーカイブおよび RPM パッケージを自動的に検査します。

注記

annocheck は ELF ファイルのみを検索します。その他のファイルタイプは無視されます。

関連情報

- **annocheck** および使用可能なコマンドラインオプションに関する情報は、**annocheck** の man ページを参照してください。

4.6.2.3. annocheck を使用した RPM パッケージの検証

次のセクションでは、**annocheck** を使用して RPM パッケージの ELF ファイルを検証する方法を説明します。

手順

- RPM パッケージをスキャンするには、以下のコマンドを使用します。

```
$ annocheck rpm-package-name
```

rpm-package-name を、RPM パッケージの名前に置き換えます。**annocheck** は、RPM パッケージ内のすべての ELF ファイルを再帰的にスキャンします。

注記

annocheck は ELF ファイルのみを検索します。その他のファイルタイプは無視されます。

- debug info RPM が含まれる RPM パッケージをスキャンするには、以下を使用します。

```
$ annocheck rpm-package-name --debug-rpm debuginfo-rpm
```

rpm-package-name は RPM パッケージの名前に、 **debuginfo-rpm** はバイナリー RPM に関連付けられた debug info RPM の名前に置き換えます。

関連情報

- **annocheck** および使用可能なコマンドラインオプションに関する情報は、**annocheck** の man ページを参照してください。

4.6.2.4. annocheck の追加ツールの使用

annocheck には、バイナリーファイルを検証する複数のツールが含まれます。コマンドラインオプションを使用してこれらのツールを有効にできます。

次のセクションでは、以下を有効にする方法を説明します。

- **built-by** ツール
- **notes** ツール
- **section-size** ツール

複数のツールを同時に有効にできます。

注記

強化チェッカーはデフォルトで有効になっています。

4.6.2.4.1. built-by ツールの有効化

annocheck built-by ツールを使用して、バイナリーファイルを構築したコンパイラーの名前を検索できます。

手順

- **built-by** ツールを有効にするには以下を使用します。

```
$ annocheck --enable-built-by
```

関連情報

- **built-by** ツールの詳細は、コマンドラインオプション **--help** を参照してください。

4.6.2.4.2. notes ツールの有効化

annocheck notes ツールを使用して、**annobin** プラグインが作成したバイナリーファイルに保存されたノートを表示できます。

手順

- **notes** ツールを有効にするには、以下を使用します。

```
$ annocheck --enable-notes
```

このノートは、アドレス範囲順に表示されます。

関連情報

- **notes** ツールの詳細は、コマンドラインオプション **--help** を参照してください。

4.6.2.4.3. section-size ツールの有効化

annocheck section-size ツールを使用すると、名前付きセクションのサイズを表示できます。

手順

- **section-size** ツールを有効にするには、以下を使用します。

```
$ annocheck --section-size=name
```

name は、名前付きセクションの名前に置き換えます。出力は、特定のセクションに限定されます。累積結果は、最後に生成されます。

関連情報

- **section-size** ツールの詳細は、コマンドラインオプション **--help** を参照してください。

4.6.2.4.4. 強化チェッカーの基本

強化チェッカーはデフォルトで有効になっています。コマンドラインオプション **--disable-hardened** で、強化チェッカーを無効にできます。

4.6.2.4.4.1. 強化チェッカーのオプション

annocheck プログラムは、以下のオプションをチェックします。

- **-z now** リンカーオプションを使用して遅延結合が無効になる。
- プログラムのメモリーの実行可能なリージョン内にスタックがない。
- GOT テーブルの再配置が読み取り専用に設定されている。
- プログラムセグメントには、読み取り、書き込み、および実行権限ビットセットの 3 つすべてがある。
- 実行コードに対する再配置がない。
- ランタイム時に共有ライブラリーを見つけるための runpath 情報には、/usr にルート指定されたディレクトリーのみが含まれている。
- プログラムが **annobin** ノートを有効にしてコンパイルされている。
- プログラムが **-fstack-protector-strong** オプションを有効にしてコンパイルされている。
- プログラムが **-D_FORTIFY_SOURCE=2** でコンパイルされている。
- プログラムが **-D_GLIBCXX_ASSERTIONS** でコンパイルされている。
- プログラムが **-fexceptions** を有効にしてコンパイルされている。
- プログラムが **-fstack-clash-protection** を有効にしてコンパイルされている。
- プログラムが **-O2** 以降でコンパイルされている。
- プログラムには書き込み可能な再配置がない。
- 動的実行可能ファイルには動的セグメントがある。
- 共有ライブラリーが **-fPIC** または **-fPIE** でコンパイルされている。
- 動的実行可能ファイルは **-fPIE** でコンパイルされ、**-pie** でリンクされている。
- 利用可能な場合は、**-fcf-protection=full** オプションが使用されている。
- 利用可能な場合は、**-mbranch-protection** オプションが使用されている。

- 利用可能な場合は、**-mstackrealign** オプションが使用されている。

4.6.2.4.4.2. 強化チェッカーの無効化

次のセクションでは、強化チェッカーを無効にする方法を説明します。

手順

- 強化チェッカーがないファイルでノートをスキャンするには、以下を使用します。

```
$ annocheck --enable-notes --disable-hardened file-name
```

file-name は、ファイルの名前に置き換えます。

4.6.3. 冗長する annobin ノートの削除

annobin を使用すると、バイナリーのサイズが増えます。**annobin** でコンパイルしたバイナリーのサイズを縮小するには、冗長な **annobin** ノートを削除できます。冗長する **annobin** ノートを削除するには、**binutils** パッケージに含まれる **objcopy** プログラムを使用します。

手順

- 冗長な **annobin** ノートを削除するには、以下を使用します。

```
$ objcopy --merge-notes file-name
```

file-name は、ファイルの名前に置き換えます。

4.6.4. GCC Toolset 12 での annobin の詳細

場合によっては、GCC Toolset 12 における **annobin** と **gcc** 間の同期問題により、コンパイルが失敗し、次のようなエラーメッセージが表示されることがあります。

```
cc1: fatal error: inaccessible plugin file
opt/rh/gcc-toolset-12/root/usr/lib/gcc/architecture-linux-gnu/12/plugin/gcc-annobin.so
expanded from short plugin name gcc-annobin: No such file or directory
```

この問題を回避するには、プラグインディレクトリーに **annobin.so** ファイルから **gcc-annobin.so** ファイルへのシンボリックリンクを作成します。

```
# cd /opt/rh/gcc-toolset-12/root/usr/lib/gcc/architecture-linux-gnu/12/plugin
# ln -s annobin.so gcc-annobin.so
```

architecture はシステムでお使いのアーキテクチャーに置き換えてください。

- **aarch64**
- **i686**
- **ppc64le**
- **s390x**
- **x86_64**

第5章 補足

5.1. コンパイラーおよび開発ツールにおける互換性に影響を与える変更点

非定数の PTHREAD_STACK_MIN、MINSIGSTKSZ、および SIGSTKSZ マクロ

スケーラブルベクトルレジスターの可変スタックサイズを必要とするアーキテクチャーのサポートを向上するために、**PTHREAD_STACK_MIN**、**MINSIGSTKSZ**、および **SIGSTKSZ** マクロの定数値が、非定数値 (**sysconf** 呼び出しなど) に変更されました。

PTHREAD_STACK_MIN、**MINSIGSTKSZ**、および **SIGSTKSZ** マクロを定数値のように扱う方法は、使用できなくなりました。**PTHREAD_STACK_MIN**、**MINSIGSTKSZ**、および **SIGSTKSZ** マクロに対して返される値は、long データ型になり、符号なしの値 (**size_t** など) と比較するとコンパイラー警告が発生する可能性があります。

libc.so.6 にマージされたライブラリー

この更新により、次のライブラリーが **libc** ライブラリーにマージされました。これにより、スムーズなインプレースアップグレードエクスペリエンスを実現し、プロセスによるスレッドの安全な随時使用をサポートし、内部実装を簡素化します。

- **libpthread**
- **libdl**
- **libutil**
- **libanl**

さらに、Name Switch Service (NSS) ファイルと Domain Name System (DNS) プラグインの **libc** ライブラリーへの直接移動をサポートするために、**libresolv** ライブラリーの一部が libc に移動されました。NSS ファイルと DNS プラグインは、**libc** ライブラリーに直接組み込まれ、アップグレード時に、または **chroot** やコンテナーの境界を越えて使用できるようになりました。**chroot** またはコンテナーの境界を越えてこれらを使用すると、これらのソースからの Identity Management (IdM) データの安全なクエリーがサポートされます。

zdump ユーティリティーの場所変更

zdump ユーティリティーの場所が **/usr/bin/zdump** に変更されました。

sys_siglist、_sys_siglist、および sys_sigabbrev シンボルの非推奨化

sys_siglist、**_sys_siglist**、および **sys_sigabbrev** シンボルは、古いバイナリーをサポートするための互換性シンボルとしてのみエクスポートされます。すべてのプログラムで、代わりに **strsignal** シンボルを使用する必要があります。

sys_siglist、**_sys_siglist**、および **sys_sigabbrev** シンボルを使用すると、配列アクセスに対する明示的な境界チェックがないため、コピーの再配置や、エラーが発生しやすいアプリケーションバイナリーインターフェイス (ABI) などの問題が発生します。

この変更は、一部のパッケージのソースからのビルドに影響を与える可能性があります。この問題を解決するには、代わりに **strsignal** シンボルを使用するようにプログラムを書き直します。以下に例を示します。

```
#include <signal.h>
#include <stdio.h>
```

```
static const char *strsig (int sig)
{
  return sys_siglist[sig];
}

int main (int argc, char *argv[])
{
  printf ("%s\n", strsig (SIGINT));
  return 0;
}
```

以下のように調整する必要があります。

```
#include <signal.h>
#include <stdio.h>
#include <string.h>

static const char *strsig (int sig)
{
  return strsignal(sig);
}

int main (int argc, char *argv[])
{
  printf ("%s\n", strsig (SIGINT));
  return 0;
}
```

または、**glibc-2.32** GNU 拡張の **sigabbrev_np** または **sigdescr_np** を使用するには、次のようにします。

```
#define _GNU_SOURCE
#include <signal.h>
#include <stdio.h>
#include <string.h>

static const char *strsig (int sig)
{
  const char *r = sigdescr_np (sig);
  return r == NULL ? "Unknown signal" : r;
}

int main (int argc, char *argv[])
{
  printf ("%s\n", strsig (SIGINT));
  printf ("%s\n", strsig (-1));
  return 0;
}
```

どちらの拡張も、非同期シグナルセーフおよびマルチスレッドセーフです。

sys_errlist、_sys_errlist、sys_nerr、および _sys_nerr シンボルの非推奨化

sys_errlist、**_sys_errlist**、**sys_nerr**、および **_sys_nerr** シンボルは、古いバイナリーをサポートするための互換性シンボルとしてのみエクスポートされます。すべてのプログラムで、代わりに **strerror** または **strerror_r** シンボルを使用する必要があります。

sys_errlist、**_sys_errlist**、**sys_nerr**、および **_sys_nerr** シンボルを使用すると、配列アクセスに対する明示的な境界チェックがないため、コピーの再配置や、エラーが発生しやすい ABI などの問題が発生します。

この変更は、一部のパッケージのソースからのビルドに影響を与える可能性があります。この問題を解決するには、**strerror** または **strerror_r** シンボルを使用してプログラムを書き直します。以下に例を示します。

```
#include <stdio.h>
#include <errno.h>

static const char *strerr (int err)
{
  if (err < 0 || err > sys_nerr)
    return "Unknown";
  return sys_errlist[err];
}

int main (int argc, char *argv[])
{
  printf ("%s\n", strerr (-1));
  printf ("%s\n", strerr (EINVAL));
  return 0;
}
```

以下のように調整する必要があります。

```
#include <stdio.h>
#include <errno.h>

static const char *strerr (int err)
{
  return strerror (err);
}

int main (int argc, char *argv[])
{
  printf ("%s\n", strerr (-1));
  printf ("%s\n", strerr (EINVAL));
  return 0;
}
```

または、**glibc-2.32** GNU 拡張の **strerrorname_np** または **strerrordesc_np** を使用するには、次のようにします。

```
#define _GNU_SOURCE
#include <stdio.h>
#include <errno.h>
#include <string.h>

static const char *strerr (int err)
```

```
{
  const char *r = strerrordesc_np (err);
  return r == NULL ? "Unknown error" : r;
}

int main (int argc, char *argv[])
{
  printf ("%s\n", strerr (-1));
  printf ("%s\n", strerr (EINVAL));
  return 0;
}
```

どちらの拡張も、非同期シグナルセーフおよびマルチスレッドセーフです。

ユーザー空間メモリーアロケーター、malloc の変更

mallwatch および **tr_break** シンボルは非推奨になり、**mtrace** 関数では使用されなくなりました。GDB 内の **mtrace** 関数内で条件付きブレークポイントを使用すると、同様の機能を実現できます。

__morecore および **__after_morecore_hook malloc** フックとデフォルトの実装 **__default_morecore** は、API から削除されました。既存のアプリケーションは引き続きこれらのシンボルに対してリンクしますが、インターフェイスは **malloc** に影響を与えなくなります。

MALLOC_CHECK_ 環境変数 (または **glibc.malloc.check** 調整可能パラメーター)、**mtrace()**、および **mcheck()** などの **malloc** のデバッグ機能は、メイン C ライブラリーでデフォルトで無効になりました。これらの機能を使用するには、新しい **libc_malloc_debug.so** デバッグ DSO をプリロードします。

非推奨の関数 **malloc_get_state** および **malloc_set_state** は、コア C ライブラリーから **libc_malloc_debug.so** ライブラリーに移動されました。これらの関数をまだ使用しているレガシーアプリケーションでは、**LD_PRELOAD** 環境変数を使用して環境に **libc_malloc_debug.so** ライブラリーをプリロードする必要があります。

非推奨のメモリー割り当てフック **__malloc_hook**、**__realloc_hook**、**__memalign_hook**、および **__free_hook** は、API から削除されました。互換性シンボルはレガシープログラムをサポートするために存在しますが、新しいアプリケーションはこれらのシンボルにリンクできなくなりました。これらのフックは、**glibc** の機能に影響を与えなくなりました。**malloc** のデバッグ DSO **libc_malloc_debug.so** は、現在フックをサポートしています。これをプリロードすることで、古いプログラムでこの機能を復元できます。ただし、これは過渡的な措置であり、GNU C ライブラリーの今後のリリースでは削除される可能性があります。独自の **malloc** 割り込みライブラリーを作成してプリロードすることで、これらのフックから移植できます。

遅延結合の失敗によるプロセスの終了

ELF コンストラクターの実行中に、**dlopen** 関数で遅延結合の失敗が発生した場合、プロセスが終了します。以前は、動的ローダーが **dlopen** から **NULL** を返し、遅延結合エラーが **dlerror** メッセージでキャプチャーされていました。一般に、任意の関数呼び出しでスタックをリセットすることはできないため、これは安全ではありません。

stime 関数の非推奨化

stime 関数は、新しくリンクされたバイナリーでは使用できなくなり、その宣言が **<time.h>** から削除されました。システム時間を設定するプログラムには、代わりに **clock_settime** 関数を使用してください。

popen 関数と system 関数が atfork ハンドラーを実行しなくなる

これは POSIX 違反の可能性がありますが、**atfork** ハンドラーに関する **pthread_atfork** ドキュメントに

記載されている POSIX の根拠は、マルチスレッドプロセスでの fork 呼び出し後の整合性のないミューテックス状態を処理することです。**popen** 関数でも **system** 関数でも、ユーザー定義のミューテックスに直接アクセスすることはできません。

C++ 標準ライブラリーの非推奨機能

- **std::string::reserve(n)** が、文字列の現在の容量よりも小さい引数を使用して呼び出された場合に、文字列の容量を削減しなくなりました。引数なしで **reserve()** を呼び出すことで文字列の容量を減らすことができますが、そのような形式は非推奨です。代わりに、同等の **shrin_to_fit()** を使用してください。

- 非標準の **std::__is_nullptr_t** 型特性は非推奨になりました。代わりに、標準の **std::is_null_pointer** 特性を使用してください。