Red Hat Enterprise Linux 9

Développer des applications C et C dans RHEL 9

Configurer un poste de travail de développeur, développer et déboguer des applications C et C sous Red Hat Enterprise Linux 9

Last Updated: 2024-02-15

Red Hat Enterprise Linux 9 Développer des applications C et C dans RHEL 9

Configurer un poste de travail de développeur, développer et déboguer des applications C et C sous Red Hat Enterprise Linux 9

Notice légale

Copyright © 2023 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Résumé

Utiliser les différentes fonctionnalités et utilitaires disponibles dans Red Hat Enterprise Linux 9 pour développer et déboguer des applications C et C.

Table des matières

RENDRE L'OPEN SOURCE PLUS INCLUSIF 3

FOURNIR UN RETOUR D'INFORMATION SUR LA DOCUMENTATION DE RED HAT 4

CHAPITRE 1. MISE EN PLACE D'UN POSTE DE TRAVAIL DE DÉVELOPPEMENT 5
- 1.1. CONDITIONS PRÉALABLES 5
- 1.2. ACTIVATION DES DÉPÔTS DE DÉBOGAGE ET DE SOURCES 5
- 1.3. MISE EN PLACE DE LA GESTION DES VERSIONS DES APPLICATIONS 5
- 1.4. MISE EN PLACE DU DÉVELOPPEMENT D'APPLICATIONS EN C ET C 6
- 1.5. MISE EN PLACE DU DÉBOGAGE DES APPLICATIONS 7
- 1.6. MISE EN PLACE D'UN SYSTÈME DE MESURE DE LA PERFORMANCE DES APPLICATIONS 7

CHAPITRE 2. CRÉATION D'APPLICATIONS C OU C 9
- 2.1. GCC DANS RHEL 9 9
- 2.2. CONSTRUIRE UN CODE AVEC GCC 9
- 2.3. UTILISATION DE BIBLIOTHÈQUES AVEC GCC 16
- 2.4. CRÉER DES BIBLIOTHÈQUES AVEC GCC 24
- 2.5. GÉRER PLUS DE CODE AVEC MAKE 28

CHAPITRE 3. DÉBOGAGE DES APPLICATIONS 32
- 3.1. ACTIVATION DU DÉBOGAGE AVEC LES INFORMATIONS DE DÉBOGAGE 32
- 3.2. INSPECTION DE L'ÉTAT INTERNE DE L'APPLICATION AVEC GDB 36
- 3.3. ENREGISTREMENT DES INTERACTIONS AVEC LES APPLICATIONS 44
- 3.4. DÉBOGAGE D'UNE APPLICATION BLOQUÉE 50
- 3.5. CHANGEMENTS DE COMPATIBILITÉ DANS GDB 57

CHAPITRE 4. OUTILS SUPPLÉMENTAIRES POUR LE DÉVELOPPEMENT 60
- 4.1. UTILISATION DE LA BOÎTE À OUTILS GCC 60
- 4.2. JEU D'OUTILS GCC 12 61
- 4.3. UTILISATION DE L'IMAGE CONTENEUR DU JEU D'OUTILS GCC 65
- 4.4. OUTILS DE COMPILATION 67
- 4.5. LE PROJET ANNOBIN 68

RENDRE L'OPEN SOURCE PLUS INCLUSIF

Red Hat s'engage à remplacer les termes problématiques dans son code, sa documentation et ses propriétés Web. Nous commençons par ces quatre termes : master, slave, blacklist et whitelist. En raison de l'ampleur de cette entreprise, ces changements seront mis en œuvre progressivement au cours de plusieurs versions à venir. Pour plus de détails, voir le message de notre directeur technique Chris Wright.

FOURNIR UN RETOUR D'INFORMATION SUR LA DOCUMENTATION DE RED HAT

Nous apprécions vos commentaires sur notre documentation. Faites-nous savoir comment nous pouvons l'améliorer.

Soumettre des commentaires sur des passages spécifiques

1. Consultez la documentation au format **Multi-page HTML** et assurez-vous que le bouton **Feedback** apparaît dans le coin supérieur droit après le chargement complet de la page.
2. Utilisez votre curseur pour mettre en évidence la partie du texte que vous souhaitez commenter.
3. Cliquez sur le bouton **Add Feedback** qui apparaît près du texte en surbrillance.
4. Ajoutez vos commentaires et cliquez sur **Submit**.

Soumettre des commentaires via Bugzilla (compte requis)

1. Connectez-vous au site Web de Bugzilla.
2. Sélectionnez la version correcte dans le menu **Version**.
3. Saisissez un titre descriptif dans le champ **Summary**.
4. Saisissez votre suggestion d'amélioration dans le champ **Description**. Incluez des liens vers les parties pertinentes de la documentation.
5. Cliquez sur **Submit Bug**.

CHAPITRE 1. MISE EN PLACE D'UN POSTE DE TRAVAIL DE DÉVELOPPEMENT

Red Hat Enterprise Linux 9 prend en charge le développement d'applications personnalisées. Pour permettre aux développeurs de le faire, le système doit être configuré avec les outils et utilitaires requis. Ce chapitre énumère les cas d'utilisation les plus courants pour le développement et les éléments à installer.

1.1. CONDITIONS PRÉALABLES

- Le système doit être installé, y compris un environnement graphique, et souscrit.

1.2. ACTIVATION DES DÉPÔTS DE DÉBOGAGE ET DE SOURCES

Une installation standard de Red Hat Enterprise Linux n'active pas les référentiels de débogage et de sources. Ces référentiels contiennent des informations nécessaires pour déboguer les composants du système et mesurer leurs performances.

Procédure

- Activez les canaux du paquet d'informations de source et de débogage : La partie **$(uname -i)** est automatiquement remplacée par une valeur correspondant à l'architecture de votre système :

Nom de l'architecture	Valeur
64-bit Intel et AMD	x86_64
aRM 64 bits	aarch64
IBM POWER	ppc64le
iBM Z 64 bits	s390x

1.3. MISE EN PLACE DE LA GESTION DES VERSIONS DES APPLICATIONS

Un contrôle de version efficace est essentiel pour tous les projets multi-développeurs. Red Hat Enterprise Linux est livré avec Git, un système de contrôle de version distribué.

Procédure

1. Installer le **git** paquet :

```
# dnf install git
```

2. Facultatif : Définissez le nom complet et l'adresse électronique associés à vos commits Git :

```
$ git config --global user.name "Full Name"
$ git config --global user.email "email@example.com"
```

Remplacez *Full Name* et *email@example.com* par votre nom et votre adresse électronique réels.

3. Facultatif : Pour changer l'éditeur de texte par défaut lancé par Git, définissez la valeur de l'option de configuration **core.editor**:

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

Remplacez *command* par la commande à utiliser pour lancer l'éditeur de texte sélectionné.

Ressources supplémentaires

- Pages de manuel Linux pour Git et tutoriels :

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

Notez que de nombreuses commandes Git ont leurs propres pages de manuel. A titre d'exemple, voir *git-commit(1)*.

- *Git User's Manual* - La documentation HTML de Git se trouve à l'adresse suivante : **/usr/share/doc/git/user-manual.html**.
- Pro Git - La version en ligne du livre *Pro Git* fournit une description détaillée de Git, de ses concepts et de son utilisation.
- Référence - Version en ligne des pages de manuel Linux pour Git

1.4. MISE EN PLACE DU DÉVELOPPEMENT D'APPLICATIONS EN C ET C

Red Hat Enterprise Linux comprend des outils pour créer des applications C et C.

Conditions préalables

- Les référentiels debug et source doivent être activés.

Procédure

1. Installez le groupe de paquets **Development Tools** comprenant la collection de compilateurs GNU (GCC), le débogueur GNU (GDB) et d'autres outils de développement :

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

2. Installer la chaîne d'outils LLVM comprenant le compilateur **clang** et le débogueur **lldb**:

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

3. Facultatif : Pour les dépendances Fortran, installez le compilateur GNU Fortran :

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

1.5. MISE EN PLACE DU DÉBOGAGE DES APPLICATIONS

Red Hat Enterprise Linux offre plusieurs outils de débogage et d'instrumentation pour analyser et dépanner le comportement des applications internes.

Conditions préalables

- Les référentiels debug et source doivent être activés.

Procédure

1. Installer les outils utiles au débogage :

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

2. Installez le paquet **dnf-utils** afin d'utiliser l'outil **debuginfo-install**:

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

3. Exécuter un script d'aide SystemTap pour configurer l'environnement.

```
# stap-prep
```

   Notez que **stap-prep** installe les paquets correspondant au noyau actuellement *running*, qui peut être différent du ou des noyau(x) réellement installé(s). Pour s'assurer que **stap-prep** installe les bons paquets **kernel-debuginfo** et **kernel-headers** vérifiez la version actuelle du noyau en utilisant la commande **uname -r** et redémarrez votre système si nécessaire.

4. Assurez-vous que les politiques de **SELinux** permettent aux applications concernées de fonctionner non seulement normalement, mais aussi dans les situations de débogage. Pour plus d'informations, voir Utilisation de SELinux.

1.6. MISE EN PLACE D'UN SYSTÈME DE MESURE DE LA PERFORMANCE DES APPLICATIONS

Red Hat Enterprise Linux comprend plusieurs applications qui peuvent aider un développeur à identifier les causes de la perte de performance d'une application.

Conditions préalables

- Les référentiels debug et source doivent être activés.

Procédure

1. Installer les outils de mesure de la performance :

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

2. Exécuter un script d'aide SystemTap pour configurer l'environnement.

```
# stap-prep
```

Notez que **stap-prep** installe les paquets correspondant au noyau actuellement *running*, qui peut être différent du ou des noyau(x) réellement installé(s). Pour s'assurer que **stap-prep** installe les bons paquets **kernel-debuginfo** et **kernel-headers** vérifiez la version actuelle du noyau en utilisant la commande **uname -r** et redémarrez votre système si nécessaire.

3. Activer et démarrer le service collecteur Performance Co-Pilot (PCP) :

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

CHAPITRE 2. CRÉATION D'APPLICATIONS C OU C

Red Hat propose de nombreux outils pour créer des applications à l'aide des langages C et C. Cette partie du livre énumère certaines des tâches de développement les plus courantes.

2.1. GCC DANS RHEL 9

Red Hat Enterprise Linux 9 est distribué avec GCC 11 comme compilateur standard.

La norme linguistique par défaut de GCC 11 est C 17. Cela équivaut à utiliser explicitement l'option de ligne de commande **-std=gnu 17**.

Les normes de langage ultérieures, telles que C 20 et ainsi de suite, et les fonctions de bibliothèque introduites dans ces normes de langage ultérieures sont encore considérées comme expérimentales.

Ressources supplémentaires

- Portage vers GCC 11
- Portage de votre code en C 17 avec GCC 11

2.2. CONSTRUIRE UN CODE AVEC GCC

Ce chapitre décrit les situations dans lesquelles le code source doit être transformé en code exécutable.

2.2.1. Relation entre les formes de code

Conditions préalables

- Comprendre les concepts de compilation et de liaison

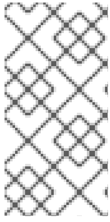
Formes de code possibles

Les langages C et C ont trois formes de code :

- **Source code** écrites en langage C ou C, présentées sous forme de fichiers de texte brut.
  Les fichiers utilisent généralement des extensions telles que **.c**, **.cc**, **.cpp**, **.h**, **.hpp**, **.i**, **.inc**. Pour une liste complète des extensions supportées et de leur interprétation, voir les pages de manuel de gcc :

  ```
  $ man gcc
  ```

- **Object code** *compiling compiler*il s'agit d'une forme intermédiaire.
  Les fichiers de code objet utilisent l'extension **.o**.
- **Executable code**, créé par le code objet *linking* avec un code objet *linker*.
  Les fichiers exécutables des applications Linux n'utilisent aucune extension de nom de fichier. Les fichiers exécutables d'objets partagés (bibliothèques) utilisent l'extension de nom de fichier **.so**.



NOTE

Il existe également des fichiers d'archives de bibliothèques pour l'établissement de liens statiques. Il s'agit d'une variante du code objet qui utilise l'extension de nom de fichier **.a**. La liaison statique n'est pas recommandée. Voir Section 2.3.2, « Liaison statique et dynamique ».

Traitement des formes de code dans GCC

La production d'un code exécutable à partir d'un code source s'effectue en deux étapes, qui nécessitent des applications ou des outils différents. GCC peut être utilisé comme pilote intelligent pour les compilateurs et les éditeurs de liens. Cela vous permet d'utiliser une seule commande **gcc** pour n'importe laquelle des actions requises (compilation et liaison). GCC sélectionne automatiquement les actions et leur ordre :

1. Les fichiers sources sont compilés en fichiers objets.
2. Les fichiers objets et les bibliothèques sont liés (y compris les sources compilées précédemment).

Il est possible d'exécuter GCC de manière à ce qu'il effectue uniquement la compilation, uniquement l'édition de liens, ou à la fois la compilation et l'édition de liens en une seule étape. Ceci est déterminé par les types d'entrées et le type de sortie(s) demandé(s).

Comme les grands projets nécessitent un système de compilation qui exécute généralement GCC séparément pour chaque action, il est préférable de toujours considérer la compilation et l'édition de liens comme deux actions distinctes, même si GCC peut exécuter les deux en même temps.

2.2.2. Compilation des fichiers source en code objet

Pour créer des fichiers de code objet à partir de fichiers source et non un fichier exécutable immédiatement, il faut demander à GCC de ne créer que des fichiers de code objet en sortie. Cette action représente l'opération de base du processus de construction pour les grands projets.

Conditions préalables

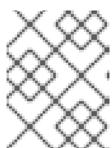
- Fichier(s) de code source C ou C
- GCC installé sur le système

Procédure

1. Se rendre dans le répertoire contenant le(s) fichier(s) du code source.
2. Exécutez **gcc** avec l'option **-c**:

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

Des fichiers objets sont créés, dont les noms reflètent les fichiers du code source original : **source.c** donne lieu à **source.o**.



NOTE

Avec le code source C, remplacez la commande **gcc** par **g** pour une gestion pratique des dépendances de la bibliothèque standard C.

2.2.3. Débogage des applications C et C avec GCC

Les informations de débogage étant volumineuses, elles ne sont pas incluses par défaut dans les fichiers exécutables. Pour permettre le débogage de vos applications C et C avec elle, vous devez explicitement demander au compilateur de la créer.

Pour permettre la création d'informations de débogage avec **GCC** lors de la compilation et de l'édition de liens, utilisez l'option **-g**:

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

- Les optimisations effectuées par le compilateur et l'éditeur de liens peuvent aboutir à un code exécutable difficile à relier au code source original : les variables peuvent être optimisées, les boucles déroulées, les opérations fusionnées avec les opérations environnantes, etc. Cela a un impact négatif sur le débogage. Pour améliorer l'expérience de débogage, envisagez de définir l'optimisation avec l'option **-Og**. Cependant, la modification du niveau d'optimisation modifie le code exécutable et peut changer le comportement réel, y compris la suppression de certains bogues.
- Pour inclure également les définitions de macros dans les informations de débogage, utilisez l'option **-g3** au lieu de **-g**.
- L'option **-fcompare-debug** GCC teste le code compilé par GCC avec des informations de débogage et sans informations de débogage. Le test est réussi si les deux fichiers binaires résultants sont identiques. Ce test garantit que le code exécutable n'est pas affecté par les options de débogage, ce qui permet de s'assurer qu'il n'y a pas de bogues cachés dans le code de débogage. Notez que l'utilisation de l'option **-fcompare-debug** augmente de manière significative le temps de compilation. Voir la page de manuel de GCC pour plus de détails sur cette option.

Ressources supplémentaires

- Utilisation de la collection de compilateurs GNU (GCC) - Options pour déboguer votre programme
- Débogage avec GDB - Informations de débogage dans des fichiers séparés
- La page du manuel GCC :

  ```
  $ man gcc
  ```

2.2.4. Optimisation du code avec GCC

Un même programme peut être transformé en plusieurs séquences d'instructions machine. Vous pouvez obtenir un résultat plus optimal si vous allouez plus de ressources à l'analyse du code lors de la compilation.

Avec GCC, vous pouvez définir le niveau d'optimisation en utilisant l'option **-O*level*** pour définir le niveau d'optimisation. Cette option accepte un ensemble de valeurs à la place de *level*.

Level	Description
0	Optimiser la vitesse de compilation - pas d'optimisation du code (par défaut).

Level	Description
1, **2**, **3**	Optimiser pour augmenter la vitesse d'exécution du code (plus le nombre est élevé, plus la vitesse est grande).
s	Optimiser la taille du fichier.
fast	Identique à un réglage de niveau **3**, mais **fast** ne tient pas compte du respect strict des normes pour permettre des optimisations supplémentaires.
g	Optimiser l'expérience de débogage.

Pour les versions, utilisez l'option d'optimisation **-O2**.

Pendant le développement, l'option **-Og** est utile pour déboguer le programme ou la bibliothèque dans certaines situations. Comme certains bogues ne se manifestent qu'avec certains niveaux d'optimisation, testez le programme ou la bibliothèque avec le niveau d'optimisation de la version.

GCC offre un grand nombre d'options pour permettre des optimisations individuelles. Pour plus d'informations, voir les ressources supplémentaires suivantes.

Ressources supplémentaires

- Utilisation de la collection de compilateurs GNU - Options qui contrôlent l'optimisation
- Page de manuel Linux pour GCC :

  ```
  $ man gcc
  ```

2.2.5. Options de durcissement du code avec GCC

Lorsque le compilateur transforme le code source en code objet, il peut ajouter diverses vérifications afin d'éviter les situations couramment exploitées et d'accroître la sécurité. Le choix du bon ensemble d'options du compilateur peut aider à produire des programmes et des bibliothèques plus sûrs, sans avoir à modifier le code source.

Options de version

La liste d'options suivante est le minimum recommandé pour les développeurs ciblant Red Hat Enterprise Linux :

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

- Pour les programmes, ajoutez les options **-fPIE** et **-pie** Position Independent Executable.
- Pour les bibliothèques liées dynamiquement, l'option obligatoire **-fPIC** (Position Independent Code) augmente indirectement la sécurité.

Options de développement

Utilisez les options suivantes pour détecter les failles de sécurité pendant le développement. Utilisez ces options en conjonction avec les options de la version release :

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

Ressources supplémentaires

- Guide de codage défensif
- Détection d'erreurs de mémoire à l'aide de GCC - Red Hat Developers Blog post

2.2.6. Lier le code pour créer des fichiers exécutables

L'édition de liens est la dernière étape de la construction d'une application C ou C. L'édition de liens combine tous les fichiers objets et les bibliothèques en un fichier exécutable.

Conditions préalables

- Un ou plusieurs fichier(s) objet(s)
- GCC doit être installé sur le système

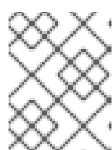
Procédure

1. Se rendre dans le répertoire contenant le(s) fichier(s) de code objet.
2. Exécuter **gcc**:

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

   Un fichier exécutable nommé ***executable-file*** est créé à partir des fichiers objets et des bibliothèques fournis.

   Pour lier des bibliothèques supplémentaires, ajoutez les options requises après la liste des fichiers objets.



NOTE

Avec le code source C, remplacez la commande **gcc** par **g** pour une gestion pratique des dépendances de la bibliothèque standard C.

2.2.7. Exemple : Construction d'un programme C avec GCC (compilation et liaison en une seule étape)

Cet exemple montre les étapes exactes de la construction d'un programme C simple.

Dans cet exemple, la compilation et l'enchaînement du code se font en une seule étape.

Conditions préalables

- Vous devez savoir comment utiliser GCC.

Procédure

1. Créez un répertoire **hello-c** et accédez-y :

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

2. Créez le fichier **hello.c** avec le contenu suivant :

```
#include <stdio.h>

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

3. Compiler et lier le code avec GCC :

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

Cette opération compile le code, crée le fichier objet **hello.o** et lie le fichier exécutable **helloworld** à partir du fichier objet.

4. Exécutez le fichier exécutable obtenu :

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

2.2.8. Exemple : Construction d'un programme C avec GCC (compilation et liaison en deux étapes)

Cet exemple montre les étapes exactes de la construction d'un programme C simple.

Dans cet exemple, la compilation et l'enchaînement du code sont deux étapes distinctes.

Conditions préalables

- Vous devez savoir comment utiliser GCC.

Procédure

1. Créez un répertoire **hello-c** et accédez-y :

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

2. Créez le fichier **hello.c** avec le contenu suivant :

```
#include <stdio.h>

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

3. Compilez le code avec GCC :

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

Le fichier objet **hello.o** est créé.

4. Lier un fichier exécutable **helloworld** à partir du fichier objet :

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

5. Exécutez le fichier exécutable obtenu :

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

2.2.9. Exemple : Construction d'un programme C avec GCC (compilation et liaison en une seule étape)

Cet exemple montre les étapes exactes de la construction d'un exemple de programme C minimal.

Dans cet exemple, la compilation et l'enchaînement du code se font en une seule étape.

Conditions préalables

- Vous devez comprendre la différence entre **gcc** et **g** .

Procédure

1. Créez un répertoire **hello-cpp** et accédez-y :

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

2. Créez le fichier **hello.cpp** avec le contenu suivant :

```
#include <iostream>

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

3. Compiler et lier le code avec **g** :

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

Cette opération compile le code, crée le fichier objet **hello.o** et lie le fichier exécutable **helloworld** à partir du fichier objet.

4. Exécutez le fichier exécutable obtenu :

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

2.2.10. Exemple : Construction d'un programme C avec GCC (compilation et liaison en deux étapes)

Cet exemple montre les étapes exactes de la construction d'un exemple de programme C minimal.

Dans cet exemple, la compilation et l'enchaînement du code sont deux étapes distinctes.

Conditions préalables

- Vous devez comprendre la différence entre **gcc** et **g** .

Procédure

1. Créez un répertoire **hello-cpp** et accédez-y :

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

2. Créez le fichier **hello.cpp** avec le contenu suivant :

```
#include <iostream>

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

3. Compilez le code avec **g** :

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

Le fichier objet **hello.o** est créé.

4. Lier un fichier exécutable **helloworld** à partir du fichier objet :

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

5. Exécutez le fichier exécutable obtenu :

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

2.3. UTILISATION DE BIBLIOTHÈQUES AVEC GCC

Ce chapitre décrit l'utilisation des bibliothèques dans le code.

2.3.1. Conventions de dénomination des bibliothèques

Une convention spéciale sur les noms de fichiers est utilisée pour les bibliothèques : une bibliothèque connue sous le nom de **foo** est censée exister sous la forme d'un fichier **lib*foo*.so** ou **lib*foo*.a**. Cette convention est automatiquement comprise par les options d'entrée de GCC, mais pas par les options de sortie :

- Lors de l'établissement d'un lien avec la bibliothèque, celle-ci ne peut être spécifiée que par son nom **foo** avec l'option **-l**, comme suit **-l*foo***:

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

- Lors de la création de la bibliothèque, le nom complet du fichier **lib*foo*.so** ou **lib*foo*.a** doit être spécifié.

2.3.2. Liaison statique et dynamique

Les développeurs ont le choix d'utiliser la liaison statique ou dynamique lorsqu'ils construisent des applications avec des langages entièrement compilés. Il est important de comprendre les différences entre l'enchaînement statique et l'enchaînement dynamique, en particulier dans le contexte de l'utilisation des langages C et C sur Red Hat Enterprise Linux. En résumé, Red Hat déconseille l'utilisation de l'édition statique de liens dans les applications pour Red Hat Enterprise Linux.

Comparaison de la liaison statique et de la liaison dynamique

L'enchaînement statique intègre les bibliothèques dans le fichier exécutable résultant. L'enchaînement dynamique conserve ces bibliothèques dans des fichiers distincts.

La liaison dynamique et la liaison statique peuvent être comparées de plusieurs façons :

Utilisation des ressources

L'établissement de liens statiques donne lieu à des fichiers exécutables plus volumineux qui contiennent plus de code. Ce code supplémentaire provenant des bibliothèques ne peut pas être partagé entre plusieurs programmes sur le système, ce qui augmente l'utilisation du système de fichiers et de la mémoire au moment de l'exécution. Plusieurs processus exécutant le même programme lié statiquement partageront toujours le code.
D'autre part, les applications statiques nécessitent moins de relocalisations pendant l'exécution, ce qui réduit le temps de démarrage, et requièrent moins de mémoire privée de taille d'ensemble résident (RSS). Le code généré pour l'enchaînement statique peut être plus efficace que pour l'enchaînement dynamique en raison de la surcharge introduite par le code indépendant de la position (PIC).

Sécurité

Les bibliothèques liées dynamiquement qui assurent la compatibilité ABI peuvent être mises à jour sans modifier les fichiers exécutables qui dépendent de ces bibliothèques. Ceci est particulièrement important pour les bibliothèques fournies par Red Hat dans le cadre de Red Hat Enterprise Linux, où Red Hat fournit des mises à jour de sécurité. L'établissement de liens statiques avec de telles bibliothèques est fortement déconseillé.

Compatibilité

La liaison statique semble fournir des fichiers exécutables indépendants des versions des bibliothèques fournies par le système d'exploitation. Cependant, la plupart des bibliothèques dépendent d'autres bibliothèques. Avec l'édition de liens statiques, cette dépendance devient inflexible et, par conséquent, la compatibilité ascendante et descendante est perdue. L'enchaînement statique est garanti pour fonctionner uniquement sur le système où le fichier exécutable a été construit.

AVERTISSEMENT

Les applications liant statiquement les bibliothèques de la bibliothèque GNU C (**glibc**) nécessitent toujours que **glibc** soit présent sur le système en tant que bibliothèque dynamique. En outre, la variante de la bibliothèque dynamique **glibc** disponible au moment de l'exécution de l'application doit être une version bitwise identique à celle présente lors de l'édition de liens de l'application. Par conséquent, l'établissement de liens statiques n'est garanti que sur le système où le fichier exécutable a été créé.

Couverture de soutien

La plupart des bibliothèques statiques fournies par Red Hat se trouvent dans le canal *CodeReady Linux Builder* et ne sont pas prises en charge par Red Hat.

Fonctionnalité

Certaines bibliothèques, notamment la bibliothèque GNU C (**glibc**), offrent des fonctionnalités réduites lorsqu'elles sont liées de manière statique.
Par exemple, lorsqu'il est lié statiquement, le site **glibc** ne prend pas en charge les threads et toute forme d'appel à la fonction **dlopen()** dans le même programme.

En raison de ces inconvénients, l'établissement de liens statiques doit être évité à tout prix, en particulier pour les applications entières et les bibliothèques **glibc** et **libstdc** .

Cas de la liaison statique

La création de liens statiques peut être un choix raisonnable dans certains cas, par exemple :

- Utilisation d'une bibliothèque qui n'est pas activée pour la liaison dynamique.
- Une liaison entièrement statique peut être nécessaire pour exécuter un code dans un environnement ou un conteneur **chroot** vide. Cependant, l'établissement de liens statiques à l'aide du paquetage **glibc-static** n'est pas pris en charge par Red Hat.

2.3.3. Optimisation du temps de liaison

L'optimisation au moment de la liaison (LTO) permet au compilateur d'effectuer diverses optimisations sur toutes les unités de traduction de votre programme en utilisant sa représentation intermédiaire au moment de la liaison. Par conséquent, vos fichiers exécutables et vos bibliothèques sont plus petits et s'exécutent plus rapidement. En outre, vous pouvez analyser le code source du paquet au moment de la compilation de manière plus approfondie en utilisant l'OLT, ce qui améliore les différents diagnostics de GCC pour les erreurs de codage potentielles.

Problèmes connus

- La violation de la règle de la définition unique (ODR) entraîne l'envoi d'un avertissement à l'adresse **-Wodr**
  Les violations de l'ODR entraînant un comportement indéfini produisent un avertissement à l'adresse **-Wodr**. Cela indique généralement un bogue dans votre programme. L'avertissement **-Wodr** est activé par défaut.

- LTO entraîne une augmentation de la consommation de mémoire
  Le compilateur consomme plus de mémoire lorsqu'il traite les unités de traduction du programme. Sur les systèmes disposant d'une mémoire limitée, désactivez le LTO ou réduisez le niveau de parallélisme lors de la construction de votre programme.

- GCC supprime les fonctions apparemment inutilisées
  GCC peut supprimer des fonctions qu'il considère comme inutilisées parce que le compilateur n'est pas en mesure de reconnaître les symboles auxquels une instruction asm() fait référence. Une erreur de compilation peut en résulter. Pour éviter cela, ajoutez **__attribute__((used))** aux symboles que vous utilisez dans votre programme.

- La compilation avec l'option **-fPIC** provoque des erreurs
  GCC n'analysant pas le contenu des instructions asm(), la compilation de votre code avec l'option de ligne de commande **-fPIC** peut provoquer des erreurs. Pour éviter cela, utilisez l'option **-fno-lto** lors de la compilation de votre unité de traduction.
  Pour plus d'informations, consultez la FAQ LTO - Utilisation de symboles à partir du langage d'assemblage.

- Le versionnage des symboles à l'aide de la directive **.symver** n'est pas compatible avec LTO
  L'implémentation du versionnage des symboles en utilisant la directive **.symver** dans une instruction asm() n'est pas compatible avec le LTO. Cependant, il est possible d'implémenter le versionnage des symboles en utilisant l'attribut **symver**. Par exemple, il est possible d'implémenter la version d'un symbole en utilisant l'attribut :

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

Ressources supplémentaires

- Manuel GCC - Attributs des fonctions
- GCC Wiki - Optimisation du temps de liaison

2.3.4. Utilisation d'une bibliothèque avec GCC

Une bibliothèque est un ensemble de codes qui peuvent être réutilisés dans votre programme. Une bibliothèque C ou C se compose de deux parties :

- Le code de la bibliothèque
- Fichiers d'en-tête

Compiler un code qui utilise une bibliothèque

Les fichiers d'en-tête décrivent l'interface de la bibliothèque : les fonctions et les variables disponibles dans la bibliothèque. Les informations contenues dans les fichiers d'en-tête sont nécessaires à la compilation du code.

Généralement, les fichiers d'en-tête d'une bibliothèque sont placés dans un répertoire différent de celui du code de votre application. Pour indiquer à GCC où se trouvent les fichiers d'en-tête, utilisez l'option **-I**:

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

Remplacez *include_path* par le chemin d'accès au répertoire du fichier d'en-tête.

L'option **-I** peut être utilisée plusieurs fois pour ajouter plusieurs répertoires contenant des fichiers d'en-tête. Lors de la recherche d'un fichier d'en-tête, ces répertoires sont parcourus dans l'ordre de leur apparition dans les options **-I**.

Lier un code qui utilise une bibliothèque

Lors de la liaison du fichier exécutable, le code objet de votre application et le code binaire de la bibliothèque doivent être disponibles. Le code des bibliothèques statiques et dynamiques se présente sous différentes formes :

- Les bibliothèques statiques sont disponibles sous forme de fichiers d'archive. Elles contiennent un groupe de fichiers objets. Le fichier d'archive a une extension de nom de fichier **.a**.
- Les bibliothèques dynamiques sont disponibles sous forme d'objets partagés. Elles constituent une forme de fichier exécutable. Un objet partagé a une extension de nom de fichier **.so**.

Pour indiquer à GCC où se trouvent les archives ou les fichiers d'objets partagés d'une bibliothèque, utilisez l'option **-L**:

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

Remplacez *library_path* par le chemin d'accès au répertoire de la bibliothèque.

L'option **-L** peut être utilisée plusieurs fois pour ajouter plusieurs répertoires. Lors de la recherche d'une bibliothèque, ces répertoires sont parcourus dans l'ordre des options **-L**.

L'ordre des options est important : GCC ne peut pas établir un lien avec une bibliothèque **foo** s'il ne connaît pas le répertoire contenant cette bibliothèque. Par conséquent, utilisez les options **-L** pour spécifier les répertoires de bibliothèques avant d'utiliser les options **-l** pour l'édition de liens avec les bibliothèques.

Compiler et lier le code qui utilise une bibliothèque en une seule étape

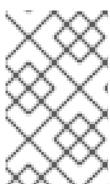
Lorsque la situation permet de compiler et de lier le code en une seule commande **gcc**, utilisez les options pour les deux situations mentionnées ci-dessus en une seule fois.

Ressources supplémentaires

- Utilisation de la collection de compilateurs GNU (GCC) - Options pour la recherche dans les répertoires
- Utilisation de la collection de compilateurs GNU (GCC) - Options pour l'établissement de liens

2.3.5. Utilisation d'une bibliothèque statique avec GCC

Les bibliothèques statiques sont disponibles sous forme d'archives contenant des fichiers objets. Après l'établissement d'un lien, elles font partie du fichier exécutable résultant.

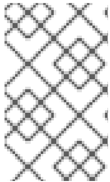


NOTE

Red Hat déconseille l'utilisation de liens statiques pour des raisons de sécurité. Voir Section 2.3.2, « Liaison statique et dynamique ». N'utilisez l'édition de liens statiques qu'en cas de nécessité, en particulier pour les bibliothèques fournies par Red Hat.

Conditions préalables

- GCC doit être installé sur votre système.
- Vous devez comprendre ce que sont les liens statiques et dynamiques.
- Vous disposez d'un ensemble de fichiers source ou objet formant un programme valide, nécessitant une bibliothèque statique **foo** et aucune autre bibliothèque.
- La bibliothèque **foo** est disponible sous la forme d'un fichier **libfoo.a**, et aucun fichier **libfoo.so** n'est fourni pour la liaison dynamique.



NOTE

La plupart des bibliothèques qui font partie de Red Hat Enterprise Linux ne sont prises en charge que pour l'édition de liens dynamiques. Les étapes ci-dessous ne fonctionnent que pour les bibliothèques qui sont *not* activées pour l'édition de liens dynamiques.

Procédure

Pour lier un programme à partir des fichiers source et objet, ajouter une bibliothèque statiquement liée **foo**, qui se trouve dans le fichier **libfoo.a**:

1. Allez dans le répertoire contenant votre code.
2. Compiler les fichiers sources du programme avec les en-têtes de la bibliothèque **foo**:

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

   Remplacez *header_path* par un chemin vers un répertoire contenant les fichiers d'en-tête de la bibliothèque **foo**.
3. Lier le programme à la bibliothèque **foo**:

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

   Remplacez *library_path* par un chemin d'accès à un répertoire contenant le fichier **libfoo.a**.
4. Pour exécuter le programme ultérieurement, il suffit de

   ```
   $ ./program
   ```

AVERTISSEMENT

L'option **-static** de GCC relative à l'édition statique de liens interdit toute édition dynamique de liens. Utilisez plutôt les options **-Wl,-Bstatic** et **-Wl,-Bdynamic** pour contrôler plus précisément le comportement de l'éditeur de liens. Voir Section 2.3.7, « Utiliser des bibliothèques statiques et dynamiques avec GCC ».

2.3.6. Utilisation d'une bibliothèque dynamique avec GCC

Les bibliothèques dynamiques sont disponibles sous forme de fichiers exécutables autonomes, nécessaires à la fois au moment de l'établissement des liens et au moment de l'exécution. Elles restent indépendantes du fichier exécutable de votre application.

Conditions préalables

- GCC doit être installé sur le système.
- Ensemble de fichiers source ou objet formant un programme valide, nécessitant une bibliothèque dynamique **foo** et aucune autre bibliothèque.
- La bibliothèque **foo** doit être disponible sous la forme d'un fichier *libfoo.so*.

Lier un programme à une bibliothèque dynamique

Pour lier un programme à une bibliothèque dynamique **foo**:

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

Lorsqu'un programme est lié à une bibliothèque dynamique, le programme résultant doit toujours charger la bibliothèque au moment de l'exécution. Il existe deux options pour localiser la bibliothèque :

- Utilisation d'une valeur **rpath** stockée dans le fichier exécutable lui-même
- Utilisation de la variable **LD_LIBRARY_PATH** au moment de l'exécution

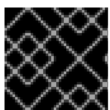
Utilisation d'une valeur rpath stockée dans le fichier exécutable

La valeur **rpath** est une valeur spéciale enregistrée dans un fichier exécutable lorsqu'il est lié. Plus tard, lorsque le programme sera chargé à partir de son fichier exécutable, l'éditeur de liens utilisera la valeur **rpath** pour localiser les fichiers de la bibliothèque.

Lors de l'établissement d'un lien avec **GCC**, le chemin *library_path* est enregistré sous **rpath**:

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

Le chemin *library_path* doit pointer vers un répertoire contenant le fichier *libfoo.so*.



IMPORTANT

N'ajoutez pas d'espace après la virgule dans l'option **-Wl,-rpath=**.

Pour exécuter le programme ultérieurement :

```
$ ./program
```

Utilisation de la variable d'environnement LD_LIBRARY_PATH

Si le fichier exécutable du programme ne contient pas **rpath**, l'éditeur de liens utilisera la variable d'environnement **LD_LIBRARY_PATH**. La valeur de cette variable doit être modifiée pour chaque programme. Cette valeur doit représenter le chemin où se trouvent les objets de la bibliothèque partagée.

Pour exécuter le programme sans **rpath**, avec les bibliothèques présentes dans le chemin *library_path*:

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

L'omission de la valeur **rpath** offre une certaine souplesse, mais nécessite la définition de la variable **LD_LIBRARY_PATH** à chaque fois que le programme doit être exécuté.

Placer la bibliothèque dans les répertoires par défaut

La configuration de l'éditeur de liens d'exécution spécifie un certain nombre de répertoires comme emplacement par défaut des fichiers de bibliothèque dynamique. Pour utiliser ce comportement par défaut, copiez votre bibliothèque dans le répertoire approprié.

Une description complète du comportement de l'éditeur de liens dynamiques n'entre pas dans le cadre de ce document. Pour plus d'informations, voir les ressources suivantes :

- Pages de manuel Linux pour l'éditeur de liens dynamiques :

  ```
  $ man ld.so
  ```

- Contenu du fichier de configuration **/etc/ld.so.conf**:

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

- Rapport des bibliothèques reconnues par l'éditeur de liens dynamiques sans configuration supplémentaire, qui inclut les répertoires :

  ```
  $ ldconfig -v
  ```

2.3.7. Utiliser des bibliothèques statiques et dynamiques avec GCC

Il est parfois nécessaire de lier certaines bibliothèques de manière statique et d'autres de manière dynamique. Cette situation pose quelques problèmes.

Conditions préalables

- Comprendre les liens statiques et dynamiques

Introduction

gcc reconnaît les bibliothèques dynamiques et statiques. Lorsque l'option **-l*foo*** est rencontrée, **gcc** tentera d'abord de localiser un objet partagé (un fichier **.so**) contenant une version liée dynamiquement de la bibliothèque **foo**, puis cherchera le fichier d'archive (**.a**) contenant une version statique de la bibliothèque. Ainsi, les situations suivantes peuvent résulter de cette recherche :

- Seul l'objet partagé est trouvé et **gcc** s'y réfère de manière dynamique.
- Seule l'archive est trouvée, et **gcc** s'y réfère de manière statique.
- L'objet partagé et l'archive sont tous deux trouvés et, par défaut, **gcc** sélectionne la liaison dynamique avec l'objet partagé.
- Aucun objet partagé ni archive n'est trouvé et la liaison échoue.

En raison de ces règles, la meilleure façon de sélectionner la version statique ou dynamique d'une bibliothèque pour l'édition de liens est de n'avoir que la version trouvée par **gcc**. Ceci peut être contrôlé

dans une certaine mesure en utilisant ou en supprimant les répertoires contenant les versions des bibliothèques, lors de la spécification des options **-L*path*** options.

En outre, comme l'enchaînement dynamique est la valeur par défaut, la seule situation où l'enchaînement doit être explicitement spécifié est celle où une bibliothèque dont les deux versions sont présentes doit être liée de manière statique. Il existe deux solutions possibles :

- Spécification des bibliothèques statiques par chemin d'accès au lieu de l'option **-l**
- Utilisation de l'option **-Wl** pour passer des options à l'éditeur de liens

Spécifier les bibliothèques statiques par fichier

Habituellement, **gcc** est chargé d'établir un lien avec la bibliothèque **foo** à l'aide de l'option **-l*foo***. Cependant, il est possible de spécifier le chemin complet du fichier **lib*foo*.a** contenant la bibliothèque :

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

D'après l'extension du fichier **.a**, **gcc** comprendra qu'il s'agit d'une bibliothèque à lier au programme. Cependant, spécifier le chemin complet du fichier de la bibliothèque est une méthode moins souple.

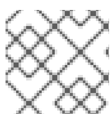
Utilisation de l'option -Wl

L'option **gcc -Wl** est une option spéciale permettant de passer des options à l'éditeur de liens sous-jacent. La syntaxe de cette option diffère de celle des autres options **gcc**. L'option **-Wl** est suivie d'une liste d'options de l'éditeur de liens séparée par des virgules, alors que les autres options **gcc** nécessitent une liste d'options séparées par des espaces.

L'éditeur de liens **ld** utilisé par **gcc** propose les options **-Bstatic** et **-Bdynamic** pour spécifier si les bibliothèques suivant cette option doivent être liées statiquement ou dynamiquement, respectivement. Après avoir transmis **-Bstatic** et une bibliothèque à l'éditeur de liens, le comportement de liaison dynamique par défaut doit être rétabli manuellement pour que les bibliothèques suivantes soient liées dynamiquement avec l'option **-Bdynamic**.

Pour lier un programme, liez la bibliothèque **first** de manière statique (**libfirst.a**) et **second** de manière dynamique (**libsecond.so**) :

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



NOTE

gcc peut être configuré pour utiliser d'autres linkers que celui par défaut **ld**.

Ressources supplémentaires

- Utilisation de la collection de compilateurs GNU (GCC) - 3.14 Options d'édition de liens
- Documentation pour binutils 2.27 - 2.1 Options de ligne de commande

2.4. CRÉER DES BIBLIOTHÈQUES AVEC GCC

Ce chapitre décrit les étapes de création des bibliothèques et explique les concepts nécessaires utilisés par le système d'exploitation Linux pour les bibliothèques.

2.4.1. Conventions de dénomination des bibliothèques

Une convention spéciale sur les noms de fichiers est utilisée pour les bibliothèques : une bibliothèque connue sous le nom de **foo** est censée exister sous la forme d'un fichier **lib*foo*.so** ou **lib*foo*.a**. Cette convention est automatiquement comprise par les options d'entrée de GCC, mais pas par les options de sortie :

- Lors de l'établissement d'un lien avec la bibliothèque, celle-ci ne peut être spécifiée que par son nom **foo** avec l'option **-l**, comme suit **-l*foo***:

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

- Lors de la création de la bibliothèque, le nom complet du fichier **lib*foo*.so** ou **lib*foo*.a** doit être spécifié.

2.4.2. Le mécanisme du soname

Les bibliothèques chargées dynamiquement (objets partagés) utilisent un mécanisme appelé *soname* pour gérer plusieurs versions compatibles d'une bibliothèque.

Conditions préalables

- Vous devez comprendre la liaison dynamique et les bibliothèques.
- Vous devez comprendre le concept de compatibilité ABI.
- Vous devez comprendre les conventions de dénomination des bibliothèques.
- Vous devez comprendre les liens symboliques.

Introduction du problème

Une bibliothèque chargée dynamiquement (objet partagé) existe en tant que fichier exécutable indépendant. Cela permet de mettre à jour la bibliothèque sans mettre à jour les applications qui en dépendent. Ce concept pose toutefois les problèmes suivants :

- Identification de la version actuelle de la bibliothèque
- Nécessité de disposer de plusieurs versions de la même bibliothèque
- Signalisation de la compatibilité ABI de chacune des multiples versions

Le mécanisme du soname

Pour résoudre ce problème, Linux utilise un mécanisme appelé soname.

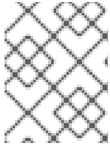
La version de la bibliothèque **foo** *X.Y* est ABI-compatible avec les autres versions dont le numéro de version contient la même valeur que *X*. Les modifications mineures préservant la compatibilité augmentent le numéro *Y*. Les modifications majeures qui rompent la compatibilité augmentent le numéro *X*.

La version actuelle de la bibliothèque **foo** *X.Y* existe sous la forme d'un fichier **libfoo.so.*x.y***. Dans le fichier de la bibliothèque, un nom de son est enregistré avec la valeur **libfoo.so.x** pour signaler la compatibilité.

Lorsque les applications sont construites, l'éditeur de liens recherche la bibliothèque dans le fichier **libfoo.so**. Un lien symbolique portant ce nom doit exister et pointer vers le fichier de la bibliothèque. L'éditeur de liens lit ensuite le nom de son du fichier de la bibliothèque et l'enregistre dans le fichier

exécutable de l'application. Enfin, l'éditeur de liens crée l'application qui déclare dépendre de la bibliothèque en utilisant le nom de son, et non un nom ou un nom de fichier.

Lorsque l'éditeur de liens dynamiques d'exécution lie une application avant de l'exécuter, il lit le soname dans le fichier exécutable de l'application. Ce nom de son est **libfoo.so.*x***. Un lien symbolique portant ce nom doit exister et pointer vers le fichier de la bibliothèque. Cela permet de charger la bibliothèque, quel que soit le composant *Y* d'une version, car le nom de son ne change pas.



NOTE

La composante *Y* du numéro de version n'est pas limitée à un seul chiffre. En outre, certaines bibliothèques codent leur version dans leur nom.

Lecture du nom de famille à partir d'un fichier

Pour afficher le nom de son d'un fichier de bibliothèque **somelibrary**:

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

Remplacez *somelibrary* par le nom du fichier de la bibliothèque que vous souhaitez examiner.

2.4.3. Créer des bibliothèques dynamiques avec GCC

Les bibliothèques liées dynamiquement (objets partagés) permettent :

- conservation des ressources par la réutilisation des codes
- une sécurité accrue en facilitant la mise à jour du code de la bibliothèque

Suivez les étapes suivantes pour créer et installer une bibliothèque dynamique à partir des sources.

Conditions préalables

- Vous devez comprendre le mécanisme du soname.
- GCC doit être installé sur le système.
- Vous devez disposer du code source d'une bibliothèque.

Procédure

1. Se rendre dans le répertoire contenant les sources de la bibliothèque.
2. Compilez chaque fichier source en un fichier objet avec l'option de code indépendant de la position **-fPIC**:

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

   Les fichiers objets portent le même nom que les fichiers du code source original, mais leur extension est **.o**.
3. Lier la bibliothèque partagée à partir des fichiers objets :

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

Le numéro de version majeure utilisé est X et le numéro de version mineure Y.

4. Copier le fichier ***libfoo.so.x.y*** à un emplacement approprié, où l'éditeur de liens dynamiques du système peut le trouver. Sur Red Hat Enterprise Linux, le répertoire des bibliothèques est **/usr/lib64**:

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

Notez que vous devez disposer des droits de root pour manipuler les fichiers de ce répertoire.

5. Créer la structure de lien symbolique pour le mécanisme de soname :

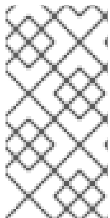
```
# ln -s libfoo.so.x.y libfoo.so.x
# ln -s libfoo.so.x libfoo.so
```

Ressources supplémentaires

- The Linux Documentation Project - Guide pratique sur les bibliothèques de programmes - 3 Bibliothèques partagées

2.4.4. Créer des bibliothèques statiques avec GCC et ar

La création de bibliothèques pour l'établissement de liens statiques est possible grâce à la conversion des fichiers objets en un type spécial de fichier d'archive.



NOTE

Red Hat décourage l'utilisation de la liaison statique pour des raisons de sécurité. N'utilisez l'édition de liens statiques qu'en cas de nécessité, en particulier pour les bibliothèques fournies par Red Hat. Consultez Section 2.3.2, « Liaison statique et dynamique » pour plus de détails.

Conditions préalables

- GCC et binutils doivent être installés sur le système.
- Vous devez comprendre ce que sont les liens statiques et dynamiques.
- Le(s) fichier(s) source(s) contenant des fonctions à partager en tant que bibliothèque est (sont) disponible(s).

Procédure

1. Créer des fichiers objets intermédiaires avec GCC.

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

Ajoutez d'autres fichiers sources si nécessaire. Les fichiers objets résultants partagent le même nom de fichier mais utilisent l'extension de nom de fichier **.o**.

2. Transformez les fichiers objets en une bibliothèque statique (archive) à l'aide de l'outil **ar** du paquet **binutils**.

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

Le fichier **libfoo.a** est créé.

3. Utilisez la commande **nm** pour inspecter l'archive résultante :

```
$ nm libfoo.a
```

4. Copiez le fichier de la bibliothèque statique dans le répertoire approprié.

5. Lors de l'édition de liens avec la bibliothèque, GCC reconnaît automatiquement, à partir de l'extension du nom de fichier **.a**, que la bibliothèque est une archive pour l'édition de liens statiques.

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

Ressources supplémentaires

- Page de manuel Linux pour *ar(1)*:

```
$ man ar
```

2.5. GÉRER PLUS DE CODE AVEC MAKE

L'utilitaire GNU Make, communément abrégé **make**, est un outil permettant de contrôler la génération d'exécutables à partir de fichiers sources. **make** détermine automatiquement quelles parties d'un programme complexe ont été modifiées et doivent être recompilées. **make** utilise des fichiers de configuration appelés Makefiles pour contrôler la manière dont les programmes sont construits.

2.5.1. Aperçu de GNU make et Makefile

Pour créer une forme utilisable (généralement des fichiers exécutables) à partir des fichiers sources d'un projet particulier, il convient d'effectuer plusieurs étapes nécessaires. Enregistrez les actions et leur séquence afin de pouvoir les répéter ultérieurement.

Red Hat Enterprise Linux contient GNU **make**, un système de construction conçu à cet effet.

Conditions préalables

- Comprendre les concepts de compilation et de liaison

GNU make

GNU **make** lit les Makefiles qui contiennent les instructions décrivant le processus de construction. Un Makefile contient plusieurs *rules* qui décrivent une manière de satisfaire une certaine condition (*target*) avec une action spécifique (*recipe*). Les règles peuvent dépendre hiérarchiquement d'une autre règle.

En lançant **make** sans aucune option, il recherche un Makefile dans le répertoire courant et tente d'atteindre la cible par défaut. Le nom du fichier Makefile peut être l'un des noms suivants : **Makefile**, **makefile**, et **GNUmakefile**. La cible par défaut est déterminée à partir du contenu du Makefile.

Détails du Makefile

Les Makefiles utilisent une syntaxe relativement simple pour définir *variables* et *rules*, qui consiste en une *target* et une *recipe*. La cible spécifie ce qui est produit si une règle est exécutée. Les lignes contenant des recettes doivent commencer par le caractère TAB.

Typiquement, un Makefile contient des règles pour la compilation des fichiers sources, une règle pour lier les fichiers objets résultants, et une cible qui sert de point d'entrée au sommet de la hiérarchie.

Considérons l'exemple suivant **Makefile** pour la construction d'un programme C qui consiste en un seul fichier, **hello.c**.

```
all: hello

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

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

Cet exemple montre que pour atteindre la cible **all**, le fichier **hello** est nécessaire. Pour obtenir **hello**, il faut **hello.o** (lié à **gcc**), qui est à son tour créé à partir de **hello.c** (compilé par **gcc**).

La cible **all** est la cible par défaut car c'est la première cible qui ne commence pas par un point (.). L'exécution de **make** sans argument est donc identique à l'exécution de **make all**, lorsque le répertoire courant contient ce **Makefile**.

Fichier makefile typique

Un Makefile plus typique utilise des variables pour la généralisation des étapes et ajoute une cible "clean" - supprimer tout sauf les fichiers sources.

```
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)
```

L'ajout de fichiers sources supplémentaires à un tel Makefile ne nécessite que de les ajouter à la ligne où la variable SOURCE est définie.

Ressources supplémentaires

- GNU make : Introduction - 2 Introduction aux Makefiles

2.5.2. Exemple : Construction d'un programme C à l'aide d'un fichier Makefile

Créez un exemple de programme C à l'aide d'un fichier Makefile en suivant les étapes de cet exemple.

Conditions préalables

- Vous devez comprendre les concepts de Makefiles et de **make**.

Procédure

1. Créez un répertoire **hellomake** et allez dans ce répertoire :

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

2. Créez un fichier **hello.c** avec le contenu suivant :

```
#include <stdio.h>

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

3. Créez un fichier **Makefile** avec le contenu suivant :

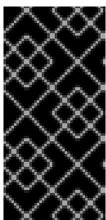
```
CC=gcc
CFLAGS=-c -Wall
SOURCE=hello.c
OBJ=$(SOURCE:.c=.o)
EXE=hello

all: $(SOURCE) $(EXE)

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

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

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



IMPORTANT

Les lignes de la recette du Makefile doivent commencer par le caractère tabulation ! Lorsque vous copiez le texte ci-dessus à partir de la documentation, le processus de copier-coller peut coller des espaces à la place des tabulations. Si cela se produit, corrigez le problème manuellement.

4. Exécuter **make**:

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

Cela crée un fichier exécutable **hello**.

5. Exécutez le fichier exécutable **hello**:

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

6. Exécutez la cible Makefile **clean** pour supprimer les fichiers créés :

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

2.5.3. Ressources documentaires pour make

Pour plus d'informations sur **make**, voir les ressources listées ci-dessous.

Documentation installée

- Utilisez les outils **man** et **info** pour consulter les pages de manuel et les pages d'information installées sur votre système :

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

Documentation en ligne

- Le manuel GNU Make hébergé par la Free Software Foundation
- Guide de l'utilisateur de Red Hat Developer Toolset - Chapitre 3. GNU make

CHAPITRE 3. DÉBOGAGE DES APPLICATIONS

Le débogage des applications est un sujet très vaste. Cette partie fournit au développeur les techniques les plus courantes de débogage dans de multiples situations.

3.1. ACTIVATION DU DÉBOGAGE AVEC LES INFORMATIONS DE DÉBOGAGE

Pour déboguer les applications et les bibliothèques, des informations de débogage sont nécessaires. Les sections suivantes décrivent comment obtenir ces informations.

3.1.1. Informations de débogage

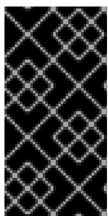
Lors du débogage d'un code exécutable, deux types d'informations permettent aux outils, et par extension au programmeur, de comprendre le code binaire :

- le texte du code source
- une description de la manière dont le texte du code source est lié au code binaire

Ces informations sont appelées "informations de débogage".

Red Hat Enterprise Linux utilise le format ELF pour les binaires exécutables, les bibliothèques partagées ou les fichiers **debuginfo**. Dans ces fichiers ELF, le format DWARF est utilisé pour contenir les informations de débogage.

Pour afficher les informations DWARF stockées dans un fichier ELF, exécutez la commande **readelf -w** ***file*** commande.



IMPORTANT

STABS est un format plus ancien et moins performant, utilisé occasionnellement avec UNIX. Son utilisation est déconseillée par Red Hat. GCC et GDB fournissent la production et la consommation de STABS sur une base de meilleur effort seulement. D'autres outils tels que Valgrind et **elfutils** ne fonctionnent pas avec STABS.

Ressources supplémentaires

- La norme de débogage DWARF

3.1.2. Débogage des applications C et C avec GCC

Les informations de débogage étant volumineuses, elles ne sont pas incluses par défaut dans les fichiers exécutables. Pour permettre le débogage de vos applications C et C avec elle, vous devez explicitement demander au compilateur de la créer.

Pour permettre la création d'informations de débogage avec **GCC** lors de la compilation et de l'édition de liens, utilisez l'option **-g**:

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

- Les optimisations effectuées par le compilateur et l'éditeur de liens peuvent aboutir à un code exécutable difficile à relier au code source original : les variables peuvent être optimisées, les boucles déroulées, les opérations fusionnées avec les opérations environnantes, etc. Cela a un

impact négatif sur le débogage. Pour améliorer l'expérience de débogage, envisagez de définir l'optimisation avec l'option **-Og**. Cependant, la modification du niveau d'optimisation modifie le code exécutable et peut changer le comportement réel, y compris la suppression de certains bogues.

- Pour inclure également les définitions de macros dans les informations de débogage, utilisez l'option **-g3** au lieu de **-g**.
- L'option **-fcompare-debug** GCC teste le code compilé par GCC avec des informations de débogage et sans informations de débogage. Le test est réussi si les deux fichiers binaires résultants sont identiques. Ce test garantit que le code exécutable n'est pas affecté par les options de débogage, ce qui permet de s'assurer qu'il n'y a pas de bogues cachés dans le code de débogage. Notez que l'utilisation de l'option **-fcompare-debug** augmente de manière significative le temps de compilation. Voir la page de manuel de GCC pour plus de détails sur cette option.

Ressources supplémentaires

- Utilisation de la collection de compilateurs GNU (GCC) - Options pour déboguer votre programme
- Débogage avec GDB - Informations de débogage dans des fichiers séparés
- La page du manuel GCC :

```
$ man gcc
```

3.1.3. Paquets debuginfo et debugsource

Les paquets **debuginfo** et **debugsource** contiennent des informations de débogage et le code source de débogage pour les programmes et les bibliothèques. Pour les applications et les bibliothèques installées dans des paquets provenant des dépôts de Red Hat Enterprise Linux, vous pouvez obtenir des paquets distincts **debuginfo** et **debugsource** à partir d'un canal supplémentaire.

Types de paquets d'informations de débogage

Il existe deux types de paquets disponibles pour le débogage :

Paquets Debuginfo

Les paquets **debuginfo** fournissent les informations de débogage nécessaires pour fournir des noms lisibles par l'homme pour les caractéristiques du code binaire. Ces paquets contiennent des fichiers **.debug**, qui contiennent des informations de débogage DWARF. Ces fichiers sont installés dans le répertoire **/usr/lib/debug**.

Paquets Debugsource

Les paquets **debugsource** contiennent les fichiers sources utilisés pour compiler le code binaire. Lorsque les paquets **debuginfo** et **debugsource** sont installés, les débogueurs tels que GDB ou LLDB peuvent relier l'exécution du code binaire au code source. Les fichiers du code source sont installés dans le répertoire **/usr/src/debug**.

3.1.4. Obtenir les paquets d'informations de débogage pour une application ou une bibliothèque à l'aide de GDB

Les informations de débogage sont nécessaires pour déboguer le code. Pour le code installé à partir d'un paquetage, le débogueur GNU (GDB) reconnaît automatiquement les informations de débogage

manquantes, résout le nom du paquetage et fournit des conseils concrets sur la manière d'obtenir le paquetage.

Conditions préalables

- L'application ou la bibliothèque que vous souhaitez déboguer doit être installée sur le système.
- GDB et l'outil **debuginfo-install** doivent être installés sur le système. Pour plus de détails, voir Configuration pour le débogage d'applications .
- Les dépôts fournissant les paquets **debuginfo** et **debugsource** doivent être configurés et activés sur le système. Pour plus de détails, voir Activation des référentiels de débogage et de sources.

Procédure

1. Lancez GDB attaché à l'application ou à la bibliothèque que vous souhaitez déboguer. GDB reconnaît automatiquement les informations de débogage manquantes et suggère une commande à exécuter.

```
$ 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. Quittez GDB : tapez **q** et confirmez avec **Enter**.

```
(gdb) q
```

3. Exécutez la commande suggérée par GDB pour installer les paquets **debuginfo** requis :

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

   L'outil de gestion des paquets **dnf** fournit un résumé des changements, demande une confirmation et, une fois que vous avez confirmé, télécharge et installe tous les fichiers nécessaires.

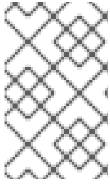
4. Si GDB n'est pas en mesure de suggérer le paquet **debuginfo**, suivez la procédure décrite dans Obtenir manuellement les paquets debuginfo pour une application ou une bibliothèque .

Ressources supplémentaires

- Guide d'utilisation de Red Hat Developer Toolset, section Installation des informations de débogage
- Comment puis-je télécharger ou installer des paquets debuginfo pour les systèmes RHEL ? - Solution de la base de connaissances de Red Hat

3.1.5. Obtenir manuellement les paquets d'informations de débogage pour une application ou une bibliothèque

Vous pouvez déterminer manuellement les paquets **debuginfo** que vous devez installer en localisant le fichier exécutable, puis le paquet qui l'installe.



NOTE

Red Hat vous recommande d'utiliser GDB pour déterminer les paquetages à installer. Utilisez cette procédure manuelle uniquement si GDB n'est pas en mesure de suggérer le paquetage à installer.

Conditions préalables

- L'application ou la bibliothèque doit être installée sur le système.
- L'application ou la bibliothèque a été installée à partir d'un paquetage.
- L'outil **debuginfo-install** doit être disponible sur le système.
- Les canaux fournissant les paquets **debuginfo** doivent être configurés et activés sur le système.

Procédure

1. Rechercher le fichier exécutable de l'application ou de la bibliothèque.

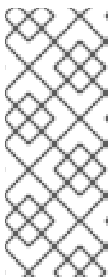
   a. Utilisez la commande **which** pour trouver le fichier d'application.

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

   b. Utilisez la commande **locate** pour trouver le fichier de la bibliothèque.

   ```
   $ locate libz | grep so
   /usr/lib64/libz.so.1
   /usr/lib64/libz.so.1.2.11
   ```

   Si les raisons initiales du débogage comprennent des messages d'erreur, choisissez le résultat où le nom de fichier de la bibliothèque contient les mêmes nombres supplémentaires que ceux mentionnés dans les messages d'erreur. En cas de doute, essayez de suivre le reste de la procédure avec le résultat où le nom de fichier de la bibliothèque ne comporte pas de chiffres supplémentaires.

   NOTE

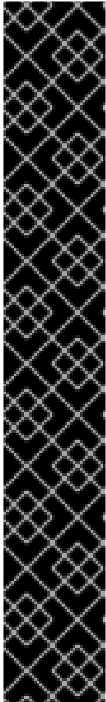
   La commande **locate** est fournie par le paquet **mlocate**. Pour l'installer et permettre son utilisation :

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

2. Rechercher le nom et la version du paquet qui a fourni le fichier :

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

La sortie fournit des détails sur le paquet installé au format *name*:*epoch*-*version*.*release*.*architecture*.



IMPORTANT

Si cette étape ne donne aucun résultat, il n'est pas possible de déterminer quel paquet a fourni le fichier binaire. Plusieurs cas sont possibles :

- Le fichier est installé à partir d'un paquet qui n'est pas connu des outils de gestion des paquets dans leur configuration *current*.
- Le fichier est installé à partir d'un paquet téléchargé localement et installé manuellement. Dans ce cas, il est impossible de déterminer automatiquement un paquetage **debuginfo** approprié.
- Vos outils de gestion des paquets sont mal configurés.
- Le fichier n'est pas installé à partir d'un paquetage. Dans ce cas, il n'existe pas de paquetage **debuginfo** correspondant.

Comme les étapes suivantes dépendent de celle-ci, vous devez résoudre cette situation ou interrompre la procédure. La description des étapes exactes du dépannage dépasse le cadre de cette procédure.

3. Installez les paquets **debuginfo** à l'aide de l'utilitaire **debuginfo-install**. Dans la commande, utilisez le nom du paquet et les autres détails que vous avez déterminés à l'étape précédente :

```
# debuginfo-install zlib-1.2.11-10.el8.x86_64
```

Ressources supplémentaires

- Guide d'utilisation de Red Hat Developer Toolset, section Installation des informations de débogage
- Comment puis-je télécharger ou installer des paquets debuginfo pour les systèmes RHEL ? - Article de la base de connaissances

3.2. INSPECTION DE L'ÉTAT INTERNE DE L'APPLICATION AVEC GDB

Pour savoir pourquoi une application ne fonctionne pas correctement, il faut contrôler son exécution et examiner son état interne à l'aide d'un débogueur. Cette section décrit comment utiliser le débogueur GNU (GDB) pour cette tâche.

3.2.1. Débogueur GNU (GDB)

Red Hat Enterprise Linux contient le débogueur GNU (GDB) qui vous permet d'examiner ce qui se passe à l'intérieur d'un programme par le biais d'une interface utilisateur en ligne de commande.

Capacités de GDB

Une seule session GDB peut déboguer les types de programmes suivants :

- Programmes multithreads et forkings
- Plusieurs programmes à la fois

- Programmes sur des machines distantes ou dans des conteneurs avec l'utilitaire **gdbserver** connecté via une connexion réseau TCP/IP

Exigences en matière de débogage

Pour déboguer un code exécutable, GDB a besoin d'informations de débogage pour ce code particulier :

- Pour les programmes que vous avez développés, vous pouvez créer les informations de débogage pendant la construction du code.
- Pour les programmes système installés à partir de paquets, vous devez installer leurs paquets debuginfo.

3.2.2. Attacher GDB à un processus

Afin d'examiner un processus, GDB doit se trouver à l'adresse *attached*.

Conditions préalables

- GDB doit être installé sur le système

Démarrer un programme avec GDB

Lorsque le programme n'est pas exécuté en tant que processus, démarrez-le avec GDB :

```
$ gdb program
```

Remplacez *program* par un nom de fichier ou un chemin d'accès au programme.

GDB se met en place pour démarrer l'exécution du programme. Vous pouvez configurer les points d'arrêt et l'environnement **gdb** avant de commencer l'exécution du processus à l'aide de la commande **run**.

Attacher GDB à un processus déjà en cours d'exécution

Pour attacher GDB à un programme déjà en cours d'exécution en tant que processus :

1. Recherchez l'ID du processus (*pid*) à l'aide de la commande **ps**:

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

   Remplacez *program* par un nom de fichier ou un chemin d'accès au programme.

2. Attachez la GDB à ce processus :

   ```
   $ gdb -p pid
   ```

   Remplacer *pid* par un numéro d'identification de processus réel provenant de la sortie **ps**.

Attacher une GDB déjà en cours d'exécution à un processus déjà en cours d'exécution

Pour attacher une GDB déjà en cours d'exécution à un programme déjà en cours d'exécution :

1. Utilisez la commande GDB **shell** pour exécuter la commande **ps** et trouver l'ID du processus du programme (*pid*) :

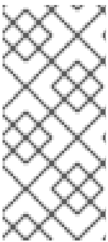
```
(gdb) shell ps -C program -o pid h
 pid
```

Remplacez *program* par un nom de fichier ou un chemin d'accès au programme.

2. Utilisez la commande **attach** pour attacher GDB au programme :

```
(gdb) attacher pid
```

Remplacer *pid* par un numéro d'identification de processus réel provenant de la sortie **ps**.



NOTE

Dans certains cas, GDB peut ne pas être en mesure de trouver le fichier exécutable correspondant. Utilisez la commande **file** pour spécifier le chemin d'accès :

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

Ressources supplémentaires

- Débogage avec GDB - 2.1 Invocation de GDB
- Débogage avec GDB - 4.7 Débogage d'un processus en cours d'exécution

3.2.3. Parcourir le code d'un programme avec GDB

Une fois que le débogueur **GDB** est connecté à un programme, vous pouvez utiliser un certain nombre de commandes pour contrôler l'exécution du programme.

Conditions préalables

- Vous devez disposer des informations de débogage nécessaires :
  - Le programme est compilé et construit avec des informations de débogage, ou
  - Les paquets debuginfo pertinents sont installés
- GDB doit être attaché au programme à déboguer

Commandes GDB pour parcourir le code

r (courir)

Lance l'exécution du programme. Si **run** est exécuté avec des arguments, ceux-ci sont transmis à l'exécutable comme si le programme avait été lancé normalement. Les utilisateurs lancent normalement cette commande après avoir défini des points d'arrêt.

start

Lancer l'exécution du programme mais s'arrêter au début de la fonction principale du programme. Si **start** est exécuté avec des arguments, ceux-ci sont transmis à l'exécutable comme si le programme avait été lancé normalement.

c (suite)

Poursuivre l'exécution du programme à partir de l'état actuel. L'exécution du programme se poursuivra jusqu'à ce que l'une des conditions suivantes soit remplie :

- Un point d'arrêt est atteint.
- Une condition spécifiée est remplie.
- Un signal est reçu par le programme.
- Une erreur s'est produite.
- Le programme se termine.

n (suite)

Poursuivre l'exécution du programme à partir de l'état actuel, jusqu'à ce que la ligne de code suivante dans le fichier source actuel soit atteinte. L'exécution du programme se poursuivra jusqu'à ce que l'une des conditions suivantes soit remplie :

- Un point d'arrêt est atteint.
- Une condition spécifiée est remplie.
- Un signal est reçu par le programme.
- Une erreur s'est produite.
- Le programme se termine.

s (étape)

La commande **step** arrête également l'exécution à chaque ligne séquentielle de code dans le fichier source actuel. Cependant, si l'exécution est actuellement arrêtée à une ligne de source contenant un **function call**, GDB arrête l'exécution après avoir entré l'appel de fonction (au lieu de l'exécuter).

until ***location***

L'exécution se poursuit jusqu'à ce que l'emplacement du code spécifié par l'option *location* soit atteint.

fini (finition)

Reprendre l'exécution du programme et s'arrêter lorsque l'exécution revient d'une fonction. L'exécution du programme se poursuit jusqu'à ce que l'une des conditions suivantes soit remplie :

- Un point d'arrêt est atteint.
- Une condition spécifiée est remplie.
- Un signal est reçu par le programme.
- Une erreur s'est produite.
- Le programme se termine.

q (quitter)

Termine l'exécution et quitte GDB.

Ressources supplémentaires

- Débogage avec GDB - 4.2 Démarrage du programme
- Débogage avec GDB - 5.2 Continuer et avancer

3.2.4. Afficher les valeurs internes du programme avec GDB

L'affichage des valeurs des variables internes d'un programme est important pour comprendre ce que fait le programme. GDB propose plusieurs commandes que vous pouvez utiliser pour inspecter les variables internes. Les commandes suivantes sont les plus utiles :

p (en caractères d'imprimerie)

Affiche la valeur de l'argument donné. En général, l'argument est le nom d'une variable de toute complexité, d'une simple valeur unique à une structure. Un argument peut également être une expression valide dans le langage courant, y compris l'utilisation de variables de programme et de fonctions de bibliothèque, ou de fonctions définies dans le programme testé.
Il est possible d'étendre GDB avec *pretty-printer* Python ou Guile scripts pour un affichage personnalisé des structures de données (telles que les classes, les structures) en utilisant la commande **print**.

bt (backtrace)

Affiche la chaîne d'appels de fonctions utilisée pour atteindre le point d'exécution actuel, ou la chaîne de fonctions utilisée jusqu'à la fin de l'exécution. Cette fonction est utile pour enquêter sur des bogues graves (tels que les erreurs de segmentation) dont les causes sont difficiles à cerner.
L'ajout de l'option **full** à la commande **backtrace** permet également d'afficher les variables locales.

Il est possible d'étendre GDB avec des scripts Python *frame filter* pour un affichage personnalisé des données affichées à l'aide des commandes **bt** et **info frame**. Le terme *frame* fait référence aux données associées à un seul appel de fonction.

info

La commande **info** est une commande générique qui permet de fournir des informations sur divers éléments. Elle prend une option spécifiant l'élément à décrire.

- La commande **info args** affiche les options de l'appel de fonction correspondant au cadre sélectionné.
- La commande **info locals** affiche les variables locales dans le cadre sélectionné.

Pour obtenir une liste des éléments possibles, exécutez la commande **help info** dans une session GDB :

```
(gdb) help info
```

l (liste)

Affiche la ligne du code source où le programme s'est arrêté. Cette commande n'est disponible que lorsque l'exécution du programme est arrêtée. Bien qu'il ne s'agisse pas à proprement parler d'une commande permettant d'afficher l'état interne, **list** aide l'utilisateur à comprendre les changements qui seront apportés à l'état interne lors de la prochaine étape de l'exécution du programme.

Ressources supplémentaires

- L'API Python GDB - Red Hat Developers Blog entry
- Débogage avec GDB - Jolie impression

3.2.5. Utilisation des points d'arrêt GDB pour arrêter l'exécution à des endroits définis du code

Souvent, seules de petites portions de code sont étudiées. Les points d'arrêt sont des marqueurs qui indiquent à GDB d'arrêter l'exécution d'un programme à un certain endroit du code. Les points d'arrêt sont le plus souvent associés à des lignes de code source. Dans ce cas, pour placer un point d'arrêt, il faut spécifier le fichier source et le numéro de ligne.

- Pour **place a breakpoint**:
    - Spécifiez le nom du code source *file* et le *line* dans ce fichier :

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

    - Si *file* n'est pas présent, le nom du fichier source au point d'exécution actuel est utilisé :

      ```
      (gdb) br line
      ```

    - Vous pouvez également utiliser le nom d'une fonction pour placer le point d'arrêt au début de celle-ci :

      ```
      (gdb) br function_name
      ```

- Un programme peut rencontrer une erreur après un certain nombre d'itérations d'une tâche. Pour spécifier un **condition** supplémentaire pour arrêter l'exécution :

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

  Remplacez *condition* par une condition en langage C ou C. La signification de *file* et *line* est la même que ci-dessus.

- Pour **inspect** l'état de tous les points d'arrêt et de surveillance :

  ```
  (gdb) info br
  ```

- Pour **remove** un point d'arrêt en utilisant son *number* comme affiché dans la sortie de **info br**:

  ```
  (gdb) supprimer number
  ```

- Pour **remove** un point d'arrêt à un endroit donné :

  ```
  (gdb) effacer file:line
  ```

Ressources supplémentaires

- Débogage avec GDB - 5.1 Points d'arrêt, points de surveillance et points de capture

3.2.6. Utilisation des points de contrôle GDB pour arrêter l'exécution lors de l'accès aux données et de leur modification

Dans de nombreux cas, il est avantageux de laisser le programme s'exécuter jusqu'à ce que certaines données soient modifiées ou consultées. Les exemples suivants représentent les cas d'utilisation les plus courants.

Conditions préalables

- Comprendre **GDB**

Utilisation de points de contrôle dans GDB

Les points de contrôle sont des marqueurs qui indiquent à **GDB** d'arrêter l'exécution d'un programme. Les points de surveillance sont associés à des données : pour placer un point de surveillance, il faut spécifier une expression décrivant une variable, plusieurs variables ou une adresse mémoire.

- Pour **place** un point de surveillance pour les données **change** (écriture) :

  ```
  (gdb) veille expression
  ```

  Remplacez *expression* par une expression qui décrit ce que vous voulez regarder. Pour les variables, *expression* est égal au nom de la variable.

- Pour **place** un point de surveillance pour les données **access** (lecture) :

  ```
  (gdb) rwatch expression
  ```

- Pour **place** un point de surveillance pour **any** l'accès aux données (lecture et écriture) :

  ```
  (gdb) awatch expression
  ```

- Pour **inspect** l'état de tous les points de contrôle et d'arrêt :

  ```
  (gdb) info br
  ```

- Pour **remove** un point de surveillance :

  ```
  (gdb) supprimer num
  ```

  Remplacer l'option ***num*** par le numéro indiqué par la commande **info br**.

Ressources supplémentaires

- Débogage avec GDB - Définition des points de surveillance

3.2.7. Déboguer des programmes à fourche ou à threads avec GDB

Certains programmes utilisent le forking ou les threads pour obtenir une exécution parallèle du code. Le débogage de plusieurs chemins d'exécution simultanés nécessite des considérations particulières.

Conditions préalables

- Vous devez comprendre les concepts de processus de bifurcation et de threads.

Déboguer des programmes forkés avec GDB

La bifurcation est une situation dans laquelle un programme (**parent**) crée une copie indépendante de lui-même (**child**). Les paramètres et commandes suivants permettent d'influencer l'action de GDB en cas de bifurcation :

- Le paramètre **follow-fork-mode** détermine si GDB suit le parent ou l'enfant après la fourche.

  set follow-fork-mode parent

Après un fork, déboguer le processus parent. C'est l'option par défaut.

set follow-fork-mode child

Après un fork, déboguer le processus enfant.

show follow-fork-mode

Affichage du réglage actuel de **follow-fork-mode**.

- Le paramètre **set detach-on-fork** détermine si la GDB garde le contrôle de l'autre processus (non suivi) ou le laisse s'exécuter.

set detach-on-fork on

Le processus qui n'est pas suivi (en fonction de la valeur de **follow-fork-mode**) est détaché et s'exécute de manière indépendante. Il s'agit de la valeur par défaut.

set detach-on-fork off

GDB garde le contrôle des deux processus. Le processus qui est suivi (en fonction de la valeur de **follow-fork-mode**) est débogué comme d'habitude, tandis que l'autre est suspendu.

show detach-on-fork

Affichage du réglage actuel de **detach-on-fork**.

Débogage de programmes threadés avec GDB

GDB a la capacité de déboguer des threads individuels, de les manipuler et de les examiner indépendamment. Pour que GDB n'arrête que le thread examiné, utilisez les commandes **set non-stop on** et **set target-async on**. Vous pouvez ajouter ces commandes au fichier **.gdbinit**. Une fois cette fonctionnalité activée, GDB est prêt à procéder au débogage des threads.

GDB utilise le concept de *current thread*. Par défaut, les commandes ne s'appliquent qu'au thread en cours.

info threads

Affiche une liste de fils de discussion avec leurs numéros **id** et **gid**, indiquant le fil de discussion en cours.

thread *id*

Définit le fil d'exécution avec l'adresse **id** spécifiée comme le fil d'exécution actuel.

thread apply *ids command*

Appliquer la commande ***command*** à tous les threads listés par ***ids***. L'option ***ids*** est une liste d'identifiants de threads séparés par des espaces. La valeur spéciale **all** applique la commande à tous les threads.

break *location* thread *id* if *condition*

Définir un point d'arrêt à un certain ***location*** avec un certain ***condition*** uniquement pour le numéro de thread ***id***.

watch *expression* thread *id*

Fixer un point de surveillance défini par ***expression*** uniquement pour le numéro de thread ***id***.

command&

Exécutez la commande **command** et revenez immédiatement à l'invite gdb **(gdb)**, en poursuivant l'exécution du code en arrière-plan.

interrupt

Arrêter l'exécution en arrière-plan.

Ressources supplémentaires

- Débogage avec GDB - 4.10 Débogage de programmes avec plusieurs threads
- Débogage avec GDB - 4.11 Débogage des fourches

3.3. ENREGISTREMENT DES INTERACTIONS AVEC LES APPLICATIONS

Le code exécutable des applications interagit avec le code du système d'exploitation et les bibliothèques partagées. L'enregistrement d'un journal d'activité de ces interactions peut fournir suffisamment d'informations sur le comportement de l'application sans déboguer le code réel de l'application. Par ailleurs, l'analyse des interactions d'une application peut aider à identifier les conditions dans lesquelles un bogue se manifeste.

3.3.1. Outils utiles pour l'enregistrement des interactions avec les applications

Red Hat Enterprise Linux propose plusieurs outils pour analyser les interactions d'une application.

stresse

L'outil **strace** permet principalement d'enregistrer les appels système (fonctions du noyau) utilisés par une application.

- L'outil **strace** peut fournir un résultat détaillé sur les appels, car **strace** interprète les paramètres et les résultats en connaissant le code du noyau sous-jacent. Les nombres sont transformés en noms de constantes respectifs, les drapeaux combinés par bit sont développés en liste de drapeaux, les pointeurs vers les tableaux de caractères sont déréférencés pour fournir la chaîne de caractères actuelle, et plus encore. La prise en charge des fonctionnalités plus récentes du noyau peut être insuffisante.
- Vous pouvez filtrer les appels tracés pour réduire la quantité de données capturées.
- L'utilisation de **strace** ne nécessite aucune configuration particulière, si ce n'est la mise en place du filtre de journalisation.
- Tracer le code de l'application avec **strace** entraîne un ralentissement significatif de l'exécution de l'application. Par conséquent, **strace** n'est pas adapté à de nombreux déploiements de production. Comme alternative, envisagez d'utiliser **ltrace** ou SystemTap.
- La version de **strace** disponible dans Red Hat Developer Toolset peut également altérer les appels système. Cette capacité est utile pour le débogage.

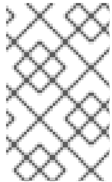
ltrace

L'outil **ltrace** permet d'enregistrer les appels de l'espace utilisateur d'une application vers des objets partagés (bibliothèques dynamiques).

- L'outil **ltrace** permet de tracer les appels à n'importe quelle bibliothèque.
- Vous pouvez filtrer les appels tracés pour réduire la quantité de données capturées.
- L'utilisation de **ltrace** ne nécessite aucune configuration particulière, si ce n'est la mise en place du filtre de journalisation.

- L'outil **ltrace** est léger et rapide, et offre une alternative à **strace**: il est possible de tracer les interfaces respectives dans des bibliothèques telles que **glibc** avec **ltrace** au lieu de tracer les fonctions du noyau avec **strace**.

- Étant donné que **ltrace** ne gère pas un ensemble connu d'appels comme **strace**, il ne tente pas d'expliquer les valeurs transmises aux fonctions de la bibliothèque. La sortie de **ltrace** ne contient que des nombres bruts et des pointeurs. L'interprétation de la sortie **ltrace** nécessite la consultation des déclarations d'interface des bibliothèques présentes dans la sortie.



NOTE

Dans Red Hat Enterprise Linux 9, un problème connu empêche **ltrace** de tracer les fichiers exécutables du système. Cette limitation ne s'applique pas aux fichiers exécutables créés par les utilisateurs.

SystemTap

SystemTap est une plateforme d'instrumentation permettant de sonder les processus en cours d'exécution et l'activité du noyau sur le système Linux. SystemTap utilise son propre langage de script pour programmer des gestionnaires d'événements personnalisés.

- Par rapport à l'utilisation de **strace** et **ltrace**, la création de scripts d'enregistrement implique plus de travail lors de la phase d'installation initiale. Cependant, les capacités de script étendent l'utilité de SystemTap au-delà de la simple production de journaux.

- SystemTap fonctionne en créant et en insérant un module dans le noyau. L'utilisation de SystemTap est efficace et ne crée pas de ralentissement significatif du système ou de l'exécution de l'application en soi.

- SystemTap est accompagné d'une série d'exemples d'utilisation.

GDB

Le débogueur GNU (GDB) est principalement destiné au débogage et non à l'enregistrement. Cependant, certaines de ses fonctionnalités le rendent utile même dans le cas où l'interaction d'une application est la principale activité d'intérêt.

- Avec GDB, il est possible de combiner facilement la capture d'un événement d'interaction avec le débogage immédiat du chemin d'exécution qui s'ensuit.

- La GDB est mieux adaptée à l'analyse de la réponse à des événements peu fréquents ou singuliers, après l'identification initiale d'une situation problématique par d'autres outils. L'utilisation de la BDG dans un scénario avec des événements fréquents devient inefficace, voire impossible.

Ressources supplémentaires

- Red Hat Enterprise Linux SystemTap Guide du débutant

- Guide de l'utilisateur de Red Hat Developer Toolset

3.3.2. Contrôler les appels système d'une application avec strace

L'outil **strace** permet de surveiller les appels système (noyau) effectués par une application.

Conditions préalables

- Vous devez avoir installé **strace** sur le système.

Procédure

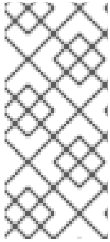
1. Identifier les appels système à surveiller.
2. Lancez **strace** et joignez-le au programme.
   - Si le programme que vous voulez surveiller n'est pas en cours d'exécution, lancez **strace** et indiquez l'adresse *program*:

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

   - Si le programme est déjà en cours d'exécution, recherchez son identifiant de processus (*pid*) et attachez-y **strace**:

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

   - Remplacez *call* par les appels système à afficher. Vous pouvez utiliser l'option **-e trace=*call*** plusieurs fois. Si elle n'est pas remplacée, **strace** affichera tous les types d'appels système. Voir la page de manuel *strace(1)* pour plus d'informations.
   - Si vous ne souhaitez pas suivre les processus ou les threads qui ont fait l'objet d'une bifurcation, n'utilisez pas l'option **-f**.
3. L'outil **strace** affiche les appels système effectués par l'application et leurs détails.
   Dans la plupart des cas, une application et ses bibliothèques effectuent un grand nombre d'appels et la sortie **strace** apparaît immédiatement, si aucun filtre pour les appels système n'est défini.
4. L'outil **strace** se termine lorsque le programme se termine.
   Pour mettre fin à la surveillance avant que le programme suivi ne se termine, appuyez sur **Ctrl C.**
   - Si **strace** a démarré le programme, celui-ci se termine en même temps que **strace**.
   - Si vous avez joint **strace** à un programme déjà en cours d'exécution, le programme se termine en même temps que **strace**.
5. Analyser la liste des appels système effectués par l'application.
   - Les problèmes d'accès ou de disponibilité des ressources sont signalés dans le journal par des appels renvoyant des erreurs.
   - Les valeurs transmises aux appels système et les modèles de séquences d'appels permettent de comprendre les causes du comportement de l'application.
   - Si l'application se bloque, les informations importantes se trouvent probablement à la fin du journal.
   - Le résultat contient beaucoup d'informations inutiles. Cependant, vous pouvez construire un filtre plus précis pour les appels système qui vous intéressent et répéter la procédure.



NOTE

Il est avantageux de voir la sortie et de l'enregistrer dans un fichier. Pour ce faire, utilisez la commande **tee**:

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

Ressources supplémentaires

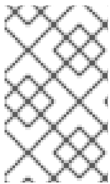
- La page du manuel *strace(1)*:

```
$ man strace
```

- Comment utiliser strace pour tracer les appels système effectués par une commande ? - Article de la base de connaissances
- Guide de l'utilisateur de Red Hat Developer Toolset - Chapitre strace

3.3.3. Contrôler les appels de fonctions de la bibliothèque d'une application avec ltrace

L'outil **ltrace** permet de surveiller les appels d'une application à des fonctions disponibles dans des bibliothèques (objets partagés).



NOTE

Dans Red Hat Enterprise Linux 9, un problème connu empêche **ltrace** de tracer les fichiers exécutables du système. Cette limitation ne s'applique pas aux fichiers exécutables créés par les utilisateurs.

Conditions préalables

- Vous devez avoir installé **ltrace** sur le système.

Procédure

1. Identifiez les bibliothèques et les fonctions qui vous intéressent, si possible.
2. Lancez **ltrace** et joignez-le au programme.
   - Si le programme que vous souhaitez surveiller n'est pas en cours d'exécution, lancez **ltrace** et indiquez *program*:

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

   - Si le programme est déjà en cours d'exécution, recherchez son identifiant de processus (*pid*) et attachez-y **ltrace**:

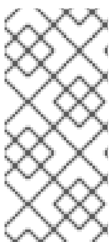
   ```
   $ ps -C program
   (...)
   $ ltrace -f -l library -e function program -ppid
   ```

   - Utilisez les options **-e**, **-f** et **-l** pour filtrer les résultats :

- Fournir les noms des fonctions à afficher sous la forme *function*. L'option **-e *function*** peut être utilisée plusieurs fois. Si elle est omise, **ltrace** affiche les appels à toutes les fonctions.
- Au lieu de spécifier des fonctions, vous pouvez spécifier des bibliothèques entières avec l'option **-l *library*** pour spécifier des bibliothèques entières. Cette option se comporte de la même manière que l'option **-e *function*** cette option se comporte de la même manière que l'option
- Si vous ne souhaitez pas suivre les processus ou les threads qui ont fait l'objet d'une bifurcation, n'utilisez pas l'option **-f**.

Voir la page du manuel *ltrace*(1)_ pour plus d'informations.

3. **ltrace** affiche les appels à la bibliothèque effectués par l'application.
   Dans la plupart des cas, une application effectue un grand nombre d'appels et la sortie **ltrace** s'affiche immédiatement, si aucun filtre n'est défini.
4. **ltrace** se termine lorsque le programme se termine.
   Pour mettre fin à la surveillance avant que le programme suivi ne se termine, appuyez sur **ctrl C.**
   - Si **ltrace** a démarré le programme, celui-ci se termine en même temps que **ltrace**.
   - Si vous avez joint **ltrace** à un programme déjà en cours d'exécution, le programme se termine en même temps que **ltrace**.
5. Analyser la liste des appels à la bibliothèque effectués par l'application.
   - Si l'application se bloque, les informations importantes se trouvent probablement à la fin du journal.
   - Le résultat contient beaucoup d'informations inutiles. Cependant, vous pouvez construire un filtre plus précis et répéter la procédure.



NOTE

Il est avantageux de voir la sortie et de l'enregistrer dans un fichier. Pour ce faire, utilisez la commande **tee**:

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

Ressources supplémentaires

- La page du manuel *ltrace(1)*:

```
$ man ltrace
```

- Guide de l'utilisateur de Red Hat Developer Toolset - Chapitre ltrace

3.3.4. Contrôler les appels système d'une application avec SystemTap

L'outil SystemTap permet d'enregistrer des gestionnaires d'événements personnalisés pour les événements du noyau. Par rapport à l'outil **strace**, il est plus difficile à utiliser mais plus efficace et permet une logique de traitement plus complexe. Un script SystemTap appelé **strace.stp** est installé avec SystemTap et fournit une approximation de la fonctionnalité de **strace** en utilisant SystemTap.

Conditions préalables

- SystemTap et les paquets de noyau respectifs doivent être installés sur le système.

Procédure

1. Recherchez l'ID du processus (*pid*) du processus que vous souhaitez surveiller :

```
$ ps -aux
```

2. Exécutez SystemTap avec le script **strace.stp**:

```
# stap /usr/share/systemtap/examples/process/strace.stp -x pid
```

   La valeur de *pid* est l'identifiant du processus.

   Le script est compilé dans un module du noyau, qui est ensuite chargé. Cela introduit un léger délai entre la saisie de la commande et l'obtention de la sortie.

3. Lorsque le processus effectue un appel système, le nom de l'appel et ses paramètres sont imprimés sur le terminal.

4. Le script se termine lorsque le processus se termine ou lorsque vous appuyez sur **Ctrl C**.

3.3.5. Utilisation de GDB pour intercepter les appels système des applications

Le débogueur GNU (GDB) vous permet d'arrêter l'exécution d'un programme dans diverses situations. Pour arrêter l'exécution lorsque le programme effectue un appel système, utilisez une commande GDB *catchpoint*.

Conditions préalables

- Vous devez comprendre l'utilisation des points d'arrêt GDB.
- GDB doit être attaché au programme.

Procédure

1. Fixer le point d'inflexion :

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

   La commande **catch syscall** définit un type spécial de point d'arrêt qui interrompt l'exécution lorsque le programme effectue un appel système.

   L'option ***syscall-name*** spécifie le nom de l'appel. Vous pouvez spécifier plusieurs points de capture pour différents appels système. L'omission de l'option ***syscall-name*** permet à GDB de s'arrêter sur n'importe quel appel système.

2. Lancer l'exécution du programme.

   - Si le programme n'a pas commencé à s'exécuter, démarrez-le :

```
(gdb) r
```

- Si l'exécution du programme est interrompue, reprenez-la :

```
(gdb) c
```

3. GDB interrompt l'exécution après que le programme a exécuté un appel système spécifié.

Ressources supplémentaires

- Débogage avec GDB - Définition des points de surveillance

3.3.6. Utilisation de GDB pour intercepter la gestion des signaux par les applications

Le débogueur GNU (GDB) vous permet d'arrêter l'exécution dans diverses situations qui surviennent au cours de l'exécution du programme. Pour arrêter l'exécution lorsque le programme reçoit un signal du système d'exploitation, utilisez une commande GDB *catchpoint*.

Conditions préalables

- Vous devez comprendre l'utilisation des points d'arrêt GDB.
- GDB doit être attaché au programme.

Procédure

1. Fixer le point d'inflexion :

```
(gdb) signal de capture signal-type
```

La commande **catch signal** définit un type spécial de point d'arrêt qui interrompt l'exécution lorsqu'un signal est reçu par le programme. L'option ***signal-type*** spécifie le type de signal. Utilisez la valeur spéciale **'all'** pour capturer tous les signaux.

2. Laissez le programme s'exécuter.

- Si le programme n'a pas commencé à s'exécuter, démarrez-le :

```
(gdb) r
```

- Si l'exécution du programme est interrompue, reprenez-la :

```
(gdb) c
```

3. GDB interrompt l'exécution après que le programme a reçu un signal spécifié.

Ressources supplémentaires

- Déboguer avec GDB - 5.1.3 Définir des points d'arrêt

3.4. DÉBOGAGE D'UNE APPLICATION BLOQUÉE

Parfois, il n'est pas possible de déboguer une application directement. Dans ce cas, vous pouvez collecter des informations sur l'application au moment de son arrêt et les analyser par la suite.

3.4.1. Core dumps : ce qu'ils sont et comment les utiliser

Un core dump est une copie d'une partie de la mémoire de l'application au moment où l'application a cessé de fonctionner, stockée au format ELF. Il contient toutes les variables internes et la pile de l'application, ce qui permet d'inspecter l'état final de l'application. Lorsqu'il est complété par le fichier exécutable correspondant et les informations de débogage, il est possible d'analyser un fichier core dump avec un débogueur d'une manière similaire à l'analyse d'un programme en cours d'exécution.

Le noyau du système d'exploitation Linux peut enregistrer des core dumps automatiquement, si cette fonctionnalité est activée. Vous pouvez également envoyer un signal à toute application en cours d'exécution pour qu'elle génère un core dump, quel que soit son état actuel.

AVERTISSEMENT

Certaines limites peuvent affecter la capacité à générer un core dump. Pour connaître les limites actuelles :

```
$ ulimit -a
```

3.4.2. Enregistrement des plantages d'application avec les core dumps

Pour enregistrer les pannes d'application, configurez l'enregistrement de la vidange du noyau et ajoutez des informations sur le système.

Procédure

1. Pour activer les vidages de noyau, assurez-vous que le fichier **/etc/systemd/system.conf** contient les lignes suivantes :

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

   Vous pouvez également ajouter des commentaires décrivant si ces paramètres étaient présents auparavant, et quelles étaient les valeurs précédentes. Cela vous permettra d'annuler ces modifications ultérieurement, si nécessaire. Les commentaires sont des lignes commençant par le caractère **#**.

   La modification du fichier nécessite un accès de niveau administrateur.

2. Appliquer la nouvelle configuration :

   ```
   # systemctl daemon-reexec
   ```

3. Supprimer les limites de taille du core dump :

   ```
   # ulimit -c unlimited
   ```

   Pour inverser ce changement, exécutez la commande avec la valeur **0** au lieu de ***unlimited***.

4. Installez le paquet **sos** qui fournit l'utilitaire **sosreport** pour collecter des informations sur le système :

```
# dnf install sos
```

5. Lorsqu'une application se bloque, un core dump est généré et traité par **systemd-coredump**.

6. Créer un rapport SOS pour fournir des informations supplémentaires sur le système :

```
# sosreport
```

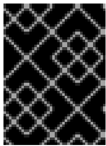
Cette opération crée une archive **.tar** contenant des informations sur votre système, telles que des copies des fichiers de configuration.

7. Localiser et exporter le core dump :

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

Si l'application s'est écrasée plusieurs fois, la sortie de la première commande énumère davantage de vidages de noyau capturés. Dans ce cas, construisez pour la deuxième commande une requête plus précise en utilisant les autres informations. Voir la page de manuel *coredumpctl(1)* pour plus de détails.

8. Transférez le core dump et le rapport SOS sur l'ordinateur où le débogage aura lieu. Transférez également le fichier exécutable, s'il est connu.



IMPORTANT

Lorsque le fichier exécutable n'est pas connu, l'analyse ultérieure du fichier central permet de l'identifier.

9. Facultatif : Supprimez le core dump et le rapport SOS après les avoir transférés, afin de libérer de l'espace disque.

Ressources supplémentaires

- Introduction à systemd dans le document *Configuring basic system settings*
- Comment activer les vidages de fichiers centraux lorsqu'une application se bloque ou qu'il y a des erreurs de segmentation - un article de la base de connaissances
- Qu'est-ce qu'un sosreport et comment en créer un dans Red Hat Enterprise Linux 4.6 et versions ultérieures ? - un article de la base de connaissances

3.4.3. Inspecter les états d'arrêt d'une application à l'aide de core dumps

Conditions préalables

- Vous devez disposer d'un fichier core dump et d'un rapport sos du système où la panne s'est produite.
- GDB et elfutils doivent être installés sur votre système.

Procédure

1. Pour identifier le fichier exécutable où le crash s'est produit, exécutez la commande **eu-unstrip** avec le fichier core dump :

   ```
   $ 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
   ```

   La sortie contient les détails de chaque module sur une ligne, séparés par des espaces. Les informations sont listées dans cet ordre :

   1. L'adresse mémoire où le module a été mappé
   2. L'identifiant de construction du module et l'endroit où il a été trouvé dans la mémoire
   3. Le nom du fichier exécutable du module, affiché sous la forme **-** s'il est inconnu, ou sous la forme **.** si le module n'a pas été chargé à partir d'un fichier
   4. La source des informations de débogage, affichée sous la forme d'un nom de fichier lorsqu'elle est disponible, sous la forme de **.** lorsqu'elle est contenue dans le fichier exécutable lui-même, ou sous la forme de **-** lorsqu'elle n'est pas présente du tout
   5. Le nom de la bibliothèque partagée (*soname*) ou **[exe]** pour le module principal

   Dans cet exemple, les détails importants sont le nom du fichier **/usr/bin/sleep** et le build-id **2818b2009547f780a5639c904cded443e564973e** sur la ligne contenant le texte **[exe]**. Grâce à ces informations, vous pouvez identifier le fichier exécutable nécessaire à l'analyse du core dump.

2. Obtenir le fichier exécutable qui s'est écrasé.

   - Si possible, copiez-le à partir du système où la panne s'est produite. Utilisez le nom du fichier extrait du fichier principal.
   - Vous pouvez également utiliser un fichier exécutable identique sur votre système. Chaque fichier exécutable construit sur Red Hat Enterprise Linux contient une note avec une valeur build-id unique. Déterminez le build-id des fichiers exécutables pertinents disponibles localement :

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

     Utilisez ces informations pour faire correspondre le fichier exécutable du système distant avec votre copie locale. Le build-id du fichier local et le build-id listé dans le core dump doivent correspondre.
   - Enfin, si l'application est installée à partir d'un paquetage RPM, vous pouvez obtenir le fichier exécutable à partir du paquetage. Utilisez la sortie de **sosreport** pour trouver la version exacte du paquetage requis.

3. Obtenez les bibliothèques partagées utilisées par le fichier exécutable. Utilisez les mêmes étapes que pour le fichier exécutable.

4. Si l'application est distribuée sous forme de paquetage, chargez le fichier exécutable dans GDB, afin d'afficher des indications sur les paquets debuginfo manquants. Pour plus de détails, voir Section 3.1.4, « Obtenir les paquets d'informations de débogage pour une application ou une bibliothèque à l'aide de GDB ».

5. Pour examiner le fichier core en détail, chargez le fichier exécutable et le fichier core dump avec GDB :

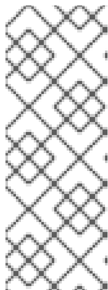
```
$ gdb -e executable_file -c core_file
```

D'autres messages concernant les fichiers manquants et les informations de débogage vous aident à identifier ce qui manque pour la session de débogage. Revenez à l'étape précédente si nécessaire.

Si les informations de débogage de l'application sont disponibles sous la forme d'un fichier et non d'un paquet, chargez ce fichier dans GDB à l'aide de la commande **symbol-file**:

```
(gdb) fichier-symbole program.debug
```

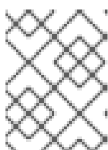
Remplacez *program.debug* par le nom réel du fichier.



NOTE

Il n'est peut-être pas nécessaire d'installer les informations de débogage pour tous les fichiers exécutables contenus dans le core dump. La plupart de ces fichiers exécutables sont des bibliothèques utilisées par le code de l'application. Ces bibliothèques peuvent ne pas contribuer directement au problème que vous analysez et vous n'avez pas besoin d'inclure les informations de débogage les concernant.

6. Utilisez les commandes GDB pour inspecter l'état de l'application au moment où elle s'est arrêtée. Voir Inspection de l'état interne de l'application avec GDB .



NOTE

Lors de l'analyse d'un fichier core, GDB n'est pas attaché à un processus en cours d'exécution. Les commandes de contrôle de l'exécution n'ont aucun effet.

Ressources supplémentaires

- Débogage avec GDB - 2.1.1 Choix des fichiers
- Débogage avec GDB - 18.1 Commandes pour spécifier des fichiers
- Débogage avec GDB - 18.3 Informations de débogage dans des fichiers séparés

3.4.4. Créer et accéder à un core dump avec coredumpctl

L'outil **coredumpctl** de **systemd** peut considérablement rationaliser le travail avec les core dumps sur la machine où le crash s'est produit. Cette procédure explique comment capturer un core dump d'un processus qui ne répond pas.

Conditions préalables

- Le système doit être configuré pour utiliser **systemd-coredump** pour la gestion du core dump. Pour vérifier que c'est bien le cas :

  ```
  $ sysctl kernel.core_pattern
  ```

  La configuration est correcte si la sortie commence par ce qui suit :

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

Procédure

1. Trouver le PID du processus suspendu, sur la base d'une partie connue du nom du fichier exécutable :

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

   Cette commande produira une ligne sous la forme

   ```
   PID command-line
   ```

   Utilisez la valeur *command-line* pour vérifier que le site *PID* appartient au processus prévu.

   Par exemple :

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

2. Envoyer un signal d'abandon au processus :

   ```
   # kill -ABRT PID
   ```

3. Vérifiez que le noyau a été capturé par **coredumpctl**:

   ```
   $ coredumpctl list PID
   ```

   Par exemple :

   ```
   $ 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. Examiner plus avant ou utiliser le fichier de base selon les besoins.
   Vous pouvez spécifier le core dump par PID et d'autres valeurs. Voir la page de manuel *coredumpctl(1)* pour plus de détails.

   - Pour afficher les détails du fichier principal :

     ```
     $ coredumpctl info PID
     ```

   - Pour charger le fichier core dans le débogueur GDB :

```
$ coredumpctl debug PID
```

En fonction de la disponibilité des informations de débogage, GDB suggérera des commandes à exécuter, telles que :

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

Pour plus de détails sur ce processus, voir Obtenir des paquets d'informations de débogage pour une application ou une bibliothèque à l'aide de GDB.

- Pour exporter le fichier de base en vue d'un traitement ultérieur ailleurs :

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

Remplacez */path/to/file_for_export* par le fichier dans lequel vous souhaitez placer le core dump.

3.4.5. Vider la mémoire du processus avec gcore

Le processus de débogage par vidage du noyau permet d'analyser l'état du programme hors ligne. Dans certains cas, vous pouvez utiliser ce processus avec un programme toujours en cours d'exécution, par exemple lorsqu'il est difficile d'accéder à l'environnement avec le processus. Vous pouvez utiliser la commande **gcore** pour vidanger la mémoire de n'importe quel processus en cours d'exécution.

Conditions préalables

- Vous devez comprendre ce que sont les core dumps et comment ils sont créés.
- GDB doit être installé sur le système.

Procédure

1. Trouvez l'identifiant du processus (*pid*). Utilisez des outils tels que **ps**, **pgrep**, et **top**:

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

2. Vider la mémoire de ce processus :

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

Cette opération crée un fichier ***filename*** et y déverse la mémoire du processus. Pendant que la mémoire est vidée, l'exécution du processus est interrompue.

3. Une fois le vidage du noyau terminé, le processus reprend son cours normal.

4. Créer un rapport SOS pour fournir des informations supplémentaires sur le système :

```
# sosreport
```

Cette opération crée une archive tar contenant des informations sur votre système, telles que des copies des fichiers de configuration.

5. Transférez le fichier exécutable du programme, le core dump et le rapport SOS sur l'ordinateur où le débogage aura lieu.

6. Facultatif : Supprimez le core dump et le rapport SOS après les avoir transférés, afin de libérer de l'espace disque.

Ressources supplémentaires

- Comment obtenir un fichier core sans redémarrer une application ? - Article de la base de connaissances

3.4.6. Vider la mémoire d'un processus protégé avec GDB

Vous pouvez marquer la mémoire des processus comme ne devant pas être vidée. Cela permet d'économiser des ressources et d'assurer une sécurité supplémentaire lorsque la mémoire du processus contient des données sensibles : par exemple, dans les applications bancaires ou comptables ou sur des machines virtuelles entières. Les vidanges du noyau (**kdump**) et les vidanges manuelles du noyau (**gcore**, GDB) ne vidangent pas la mémoire marquée de cette manière.

Dans certains cas, vous devez extraire tout le contenu de la mémoire du processus sans tenir compte de ces protections. Cette procédure montre comment procéder à l'aide du débogueur GDB.

Conditions préalables

- Vous devez comprendre ce que sont les core dumps.
- GDB doit être installé sur le système.
- GDB doit déjà être attaché au processus avec la mémoire protégée.

Procédure

1. Configurer GDB pour qu'il ignore les paramètres du fichier **/proc/PID/coredump_filter**:

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

2. Configure GDB pour qu'il ignore le drapeau de page mémoire **VM_DONTDUMP**:

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

3. Vider la mémoire :

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

Remplacez *core-file* par le nom du fichier dans lequel vous voulez vider la mémoire.

Ressources supplémentaires

- Débogage avec GDB - Comment produire un fichier Core à partir de votre programme

3.5. CHANGEMENTS DE COMPATIBILITÉ DANS GDB

La version de GDB fournie dans Red Hat Enterprise Linux 9 contient un certain nombre de changements qui rompent la compatibilité. Les sections suivantes fournissent plus de détails sur ces changements.

Commandes

- La commande **gdb -P python-script.py** n'est plus prise en charge.
  Utilisez plutôt la commande **gdb -ex 'source python-script.py'**.
- La commande **gdb *COREFILE*** n'est plus prise en charge.
  Utilisez plutôt la commande **gdb *EXECUTABLE* --core *COREFILE*** pour charger l'exécutable spécifié dans le fichier core.
- GDB stylise désormais la sortie par défaut.
  Cette nouvelle modification peut perturber les scripts qui essaient d'analyser la sortie de GDB. Utilisez la commande **gdb -ex 'set style enabled off'** pour désactiver le style dans les scripts.
- Les commandes définissent désormais la syntaxe des symboles en fonction de la langue.
  Les commandes **info functions**, **info types**, **info variables** et **rbreak** définissent désormais la syntaxe des entités en fonction de la langue choisie par la commande **set language**. En choisissant **set language auto**, GDB choisira automatiquement la langue des entités affichées.
- Les commandes **set print raw frame-arguments** et **show print raw frame-arguments** sont obsolètes.
  Ces commandes sont remplacées par les commandes **set print raw-frame-arguments** et **show print raw-frame-arguments**. Les anciennes commandes pourront être supprimées dans les versions futures.
- Les commandes suivantes de l'interface utilisateur sont désormais sensibles à la casse :
  - **focus**
  - **winheight**
  -
  - **-**
  - **>**
  - **<**
- Les commandes **help** et **apropos** n'affichent plus qu'une seule fois les informations relatives à la commande.
  Ces commandes n'affichent plus qu'une seule fois la documentation d'une commande, même si cette commande a un ou plusieurs alias. Ces commandes affichent désormais le nom de la commande, puis tous ses alias, et enfin la description de la commande.

L'interprète MI

- La version par défaut de l'interpréteur MI est désormais la version 3.
  L'édition d'informations sur les points d'arrêt multi-locaux (syntaxiquement incorrecte dans MI 2) a été modifiée dans MI 3, ce qui affecte les commandes et événements suivants :
  - **-break-insert**
  - **-break-info**

- **=breakpoint-created**
- **=breakpoint-modified**

Utilisez la commande **-fix-multi-location-breakpoint-output** pour activer ce comportement avec les versions précédentes de MI.

API Python

- Les symboles suivants sont désormais obsolètes :
  - **gdb.SYMBOL_VARIABLES_DOMAIN**
  - **gdb.SYMBOL_FUNCTIONS_DOMAIN**
  - **gdb.SYMBOL_TYPES_DOMAIN**
- Le type **gdb.Value** dispose d'un nouveau constructeur, qui est utilisé pour construire un **gdb.Value** à partir d'un objet tampon Python et d'un **gdb.Type**.
- Les informations sur les trames imprimées par le code Python de filtrage des trames sont désormais cohérentes avec ce que la commande **backtrace** imprime lorsqu'il n'y a pas de filtres, ou lorsque l'option **-no-filters** de la commande **backtrace** est utilisée.

CHAPITRE 4. OUTILS SUPPLÉMENTAIRES POUR LE DÉVELOPPEMENT

4.1. UTILISATION DE LA BOÎTE À OUTILS GCC

4.1.1. Qu'est-ce que la boîte à outils GCC ?

Red Hat Enterprise Linux 9 poursuit la prise en charge de GCC Toolset, un flux d'applications contenant des versions plus récentes d'outils de développement et d'analyse des performances. GCC Toolset est similaire à Red Hat Developer Toolset pour RHEL 7.

GCC Toolset est disponible en tant que flux d'application sous la forme d'une collection de logiciels dans le référentiel **AppStream**. GCC Toolset est entièrement pris en charge dans le cadre des accords de niveau d'abonnement à Red Hat Enterprise Linux, est fonctionnellement complet et est destiné à une utilisation en production. Les applications et les bibliothèques fournies par GCC Toolset ne remplacent pas les versions du système Red Hat Enterprise Linux, ne les annulent pas et ne deviennent pas automatiquement des choix par défaut ou préférés. En utilisant un cadre appelé collections de logiciels, un ensemble supplémentaire d'outils de développement est installé dans le répertoire /**opt**/ et est explicitement activé par l'utilisateur à la demande à l'aide de l'utilitaire **scl**. Sauf indication contraire pour des outils ou des fonctionnalités spécifiques, le jeu d'outils GCC est disponible pour toutes les architectures prises en charge par Red Hat Enterprise Linux.

4.1.2. Installation du jeu d'outils GCC

L'installation de GCC Toolset sur un système permet d'installer les principaux outils et toutes les dépendances nécessaires. Notez que certaines parties du jeu d'outils ne sont pas installées par défaut et doivent être installées séparément.

Procédure

- Pour installer la version du jeu d'outils GCC *N*:

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

4.1.3. Installation de paquets individuels à partir du GCC Toolset

Pour n'installer que certains outils du GCC Toolset au lieu de l'ensemble du jeu d'outils, listez les paquets disponibles et installez ceux qui sont sélectionnés avec l'outil de gestion des paquets **dnf**. Cette procédure est également utile pour les paquets qui ne sont pas installés par défaut avec le jeu d'outils.

Procédure

1. Liste des paquets disponibles dans la version du jeu d'outils GCC *N*:

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

2. Pour installer l'un de ces paquets :

   ```
   # dnf install package_name
   ```

   Remplacez *package_name* par une liste de paquets à installer, séparés par des espaces. Par exemple, pour installer les paquets **gcc-toolset-9-gdb-gdbserver** et **gcc-toolset-9-gdb-doc**:

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

4.1.4. Désinstallation de GCC Toolset

Pour supprimer GCC Toolset de votre système, désinstallez-le à l'aide de l'outil de gestion des paquets **dnf**.

Procédure

- Pour désinstaller GCC Toolset version *N*:

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

4.1.5. Exécution d'un outil à partir de la boîte à outils GCC

Pour exécuter un outil du GCC Toolset, utilisez l'utilitaire **scl**.

Procédure

- Pour exécuter un outil de la version du jeu d'outils GCC *N*:

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

4.1.6. Lancer une session shell avec GCC Toolset

GCC Toolset permet d'exécuter une session shell dans laquelle les versions des outils GCC Toolset sont utilisées à la place des versions système de ces outils, sans utiliser explicitement la commande **scl**. Cette fonction est utile lorsque vous devez lancer les outils de manière interactive à plusieurs reprises, par exemple lors de la configuration ou du test d'une installation de développement.

Procédure

- Pour lancer une session shell dans laquelle les versions des outils du Toolset GCC *N* remplacent les versions système de ces outils :

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

4.1.7. Ressources supplémentaires

- Guide de l'utilisateur de Red Hat Developer Toolset

4.2. JEU D'OUTILS GCC 12

Ce chapitre fournit des informations spécifiques à la version 12 du Toolset GCC et aux outils contenus dans cette version.

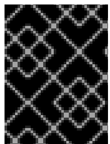
4.2.1. Outils et versions fournis par le GCC Toolset 12

Le jeu d'outils GCC 12 fournit les outils et versions suivants :

Tableau 4.1. Versions d'outils dans le Toolset 12 de GCC

Nom	Version	Description
CCG	12.1.1	Une suite de compilateurs portables prenant en charge le C, le C , et le Fortran.
GDB	11.2	Un débogueur en ligne de commande pour les programmes écrits en C, C , et Fortran.
binutils	2.38	Une collection d'outils binaires et d'autres utilitaires pour inspecter et manipuler les fichiers objets et les binaires.
dwz	0.14	Outil permettant d'optimiser la taille des informations de débogage DWARF contenues dans les bibliothèques partagées ELF et les exécutables ELF.
annobin	10.76	Un outil de vérification de la sécurité de la construction.

4.2.2. Compatibilité avec le langage C dans le jeu d'outils GCC 12



IMPORTANT

Les informations de compatibilité présentées ici ne s'appliquent qu'au GCC du GCC Toolset 12.

Le compilateur GCC du GCC Toolset peut utiliser les normes C suivantes :

C 14

Cette norme linguistique est disponible dans le jeu d'outils GCC 12.
L'utilisation de la version du langage C 14 est possible lorsque tous les objets C compilés avec le drapeau correspondant ont été construits à l'aide de la version 6 ou ultérieure de GCC.

C 11

Cette norme linguistique est disponible dans le jeu d'outils GCC 12.
L'utilisation de la version du langage C 11 est possible lorsque tous les objets C compilés avec le drapeau correspondant ont été construits à l'aide de GCC version 5 ou ultérieure.

C 98

Cette norme de langage est disponible dans GCC Toolset 12. Les binaires, les bibliothèques partagées et les objets construits à l'aide de cette norme peuvent être librement mélangés, qu'ils soient construits avec GCC à partir de GCC Toolset, Red Hat Developer Toolset, et RHEL 5, 6, 7 et 8.

C 17

Cette norme linguistique est disponible dans le jeu d'outils GCC 12.
Il s'agit de la norme de langage par défaut pour GCC Toolset 12, avec les extensions GNU, ce qui équivaut à l'utilisation explicite de l'option **-std=gnu 17**.

L'utilisation de la version du langage C 17 est possible lorsque tous les objets C compilés avec le drapeau correspondant ont été construits à l'aide de la version 10 ou ultérieure de GCC.

C 20 et C 23

Ce standard de langage n'est disponible dans GCC Toolset 12 qu'en tant que capacité expérimentale, instable et non supportée. En outre, la compatibilité des objets, des fichiers binaires et des bibliothèques construits à l'aide de ce standard ne peut être garantie.
Pour activer la prise en charge du C 20, ajoutez l'option de ligne de commande **-std=c 20** à votre ligne de commande g.

Pour activer la prise en charge du C 23, ajoutez l'option de ligne de commande **-std=c 23** à votre ligne de commande g.

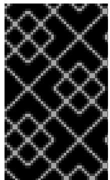
Toutes les normes de langage sont disponibles à la fois dans la variante conforme à la norme et avec les extensions GNU.

Lorsque l'on mélange des objets construits avec GCC Toolset et d'autres construits avec la chaîne d'outils RHEL (en particulier les fichiers **.o** ou **.a**), la chaîne d'outils GCC Toolset doit être utilisée pour toute liaison. Cela permet de s'assurer que toutes les nouvelles fonctionnalités des bibliothèques fournies uniquement par GCC Toolset sont résolues au moment de l'établissement des liens.

4.2.3. Spécificités de GCC dans GCC Toolset 12

Liaison statique des bibliothèques

Certaines fonctionnalités plus récentes de la bibliothèque sont liées statiquement dans les applications construites avec GCC Toolset pour prendre en charge l'exécution sur plusieurs versions de Red Hat Enterprise Linux. Cela crée un risque de sécurité mineur supplémentaire car les errata standard de Red Hat Enterprise Linux ne modifient pas ce code. Si les développeurs doivent reconstruire leurs applications en raison de ce risque, Red Hat le communiquera par le biais d'un erratum de sécurité.



IMPORTANT

En raison de ce risque de sécurité supplémentaire, il est vivement conseillé aux développeurs de ne pas lier statiquement l'ensemble de leur application pour les mêmes raisons.

Spécifier les bibliothèques après les fichiers objets lors de l'établissement des liens

Dans GCC Toolset, les bibliothèques sont liées à l'aide de scripts d'édition de liens qui peuvent spécifier certains symboles par le biais d'archives statiques. Ceci est nécessaire pour assurer la compatibilité avec plusieurs versions de Red Hat Enterprise Linux. Cependant, les scripts de l'éditeur de liens utilisent les noms des fichiers d'objets partagés respectifs. Par conséquent, l'éditeur de liens utilise des règles de gestion des symboles différentes de celles prévues et ne reconnaît pas les symboles requis par les fichiers objets lorsque l'option ajoutant la bibliothèque est spécifiée avant les options spécifiant les fichiers objets :

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

L'utilisation d'une bibliothèque du GCC Toolset de cette manière entraîne le message d'erreur de l'éditeur de liens **undefined reference to symbol**. Pour éviter ce problème, suivez la pratique standard d'édition de liens et spécifiez l'option ajoutant la bibliothèque après les options spécifiant les fichiers objets :

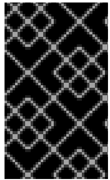
```
$ scl enable gcc-toolset-12 'gcc objfile.o -lsomelib'
```

Notez que cette recommandation s'applique également à l'utilisation de la version de base de Red Hat Enterprise Linux de **GCC**.

4.2.4. Spécificités de binutils dans GCC Toolset 12

Liaison statique des bibliothèques

Certaines fonctionnalités plus récentes de la bibliothèque sont liées statiquement dans les applications construites avec GCC Toolset pour prendre en charge l'exécution sur plusieurs versions de Red Hat Enterprise Linux. Cela crée un risque de sécurité mineur supplémentaire car les errata standard de Red Hat Enterprise Linux ne modifient pas ce code. Si les développeurs doivent reconstruire leurs applications en raison de ce risque, Red Hat le communiquera par le biais d'un erratum de sécurité.



IMPORTANT

En raison de ce risque de sécurité supplémentaire, il est vivement conseillé aux développeurs de ne pas lier statiquement l'ensemble de leur application pour les mêmes raisons.

Spécifier les bibliothèques après les fichiers objets lors de l'établissement des liens

Dans GCC Toolset, les bibliothèques sont liées à l'aide de scripts d'édition de liens qui peuvent spécifier certains symboles par le biais d'archives statiques. Ceci est nécessaire pour assurer la compatibilité avec plusieurs versions de Red Hat Enterprise Linux. Cependant, les scripts de l'éditeur de liens utilisent les noms des fichiers d'objets partagés respectifs. Par conséquent, l'éditeur de liens utilise des règles de gestion des symboles différentes de celles prévues et ne reconnaît pas les symboles requis par les fichiers objets lorsque l'option ajoutant la bibliothèque est spécifiée avant les options spécifiant les fichiers objets :

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

L'utilisation d'une bibliothèque du GCC Toolset de cette manière entraîne le message d'erreur de l'éditeur de liens **undefined reference to symbol**. Pour éviter ce problème, suivez la pratique d'édition de liens standard et spécifiez l'option ajoutant la bibliothèque après les options spécifiant les fichiers objets :

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

Notez que cette recommandation s'applique également à l'utilisation de la version de base de Red Hat Enterprise Linux de **binutils**.

4.2.5. Spécificités de annobin dans GCC Toolset 12

Dans certaines circonstances, en raison d'un problème de synchronisation entre **annobin** et **gcc** dans GCC Toolset 12, votre compilation peut échouer avec un message d'erreur qui ressemble à ce qui suit :

```
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
```

Pour contourner le problème, créez un lien symbolique dans le répertoire du plugin du fichier **annobin.so** vers le fichier **gcc-annobin.so**:

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

Remplacez *architecture* par l'architecture que vous utilisez dans votre système :

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

4.3. UTILISATION DE L'IMAGE CONTENEUR DU JEU D'OUTILS GCC

Seule l'image conteneur de GCC Toolset 12 est prise en charge. Les images conteneur des versions antérieures de GCC Toolset sont obsolètes.

Les composants du GCC Toolset 12 sont disponibles dans l'image du conteneur **GCC Toolset 12 Toolchain**.

L'image du conteneur GCC Toolset est basée sur l'image de base **rhel9** et est disponible pour toutes les architectures prises en charge par RHEL 9 :

- Architectures AMD et Intel 64 bits
- L'architecture ARM 64 bits
- IBM Power Systems, Little Endian
- iBM Z 64 bits

4.3.1. Contenu de l'image du conteneur du jeu d'outils GCC

Les versions des outils fournies dans l'image du conteneur GCC Toolset 12 correspondent aux versions des composants de GCC Toolset 12.

Contenu de la chaîne d'outils du jeu d'outils GCC 12

L'image **rhel8/gcc-toolset-12-toolchain** fournit le compilateur GCC, le débogueur GDB et d'autres outils de développement. L'image du conteneur se compose des éléments suivants :

Composant	Paquet
gcc	gcc-toolset-12-gcc
g	gcc-toolset-12-gcc-c
gfortran	gcc-toolset-12-gcc-gfortran
gdb	gcc-toolset-12-gdb

Ressources supplémentaires

- Pour utiliser les composants du GCC Toolset sur RHEL 7, utilisez Red Hat Developer Toolset qui fournit des outils de développement similaires pour les utilisateurs de RHEL 7 - Guide d'utilisation de Red Hat Developer Toolset.
- Instructions sur l'utilisation de l'image de conteneur Red Hat Developer Toolset sur RHEL 7 - Red Hat Developer Toolset images .

4.3.2. Accéder à l'image conteneur du GCC Toolset et l'exécuter

La section suivante décrit comment accéder à l'image conteneur du jeu d'outils GCC et l'exécuter.

Conditions préalables

- Podman est installé.

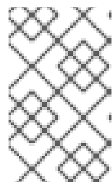
Procédure

1. Accédez au Red Hat Container Registry à l'aide de vos identifiants du portail client :

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

2. Extrayez l'image du conteneur dont vous avez besoin en exécutant la commande correspondante en tant que root :

```
# podman pull registry.redhat.io/rhel9/gcc-toolset-12-toolchain
```



NOTE

Vous pouvez également configurer votre système pour travailler avec des conteneurs en tant qu'utilisateur non rooté. Pour plus de détails, voir Configuration de conteneurs sans racine .

3. Facultatif : Vérifiez que l'extraction a réussi en exécutant une commande qui répertorie toutes les images de conteneurs sur votre système local :

```
# podman images
```

4. Exécuter un conteneur en lançant un shell bash à l'intérieur d'un conteneur :

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

L'option **-i** crée une session interactive ; sans cette option, l'interpréteur de commandes s'ouvre et se ferme instantanément.

L'option **-t** ouvre une session de terminal ; sans cette option, vous ne pouvez rien saisir dans l'interpréteur de commandes.

Ressources supplémentaires

- Construire, exécuter et gérer des conteneurs Linux sur RHEL 9
- Un article du blog de Red Hat - Understanding root inside and outside a container (Comprendre la racine à l'intérieur et à l'extérieur d'un conteneur)
- Entrées dans le Red Hat Container Registry - Images du conteneur GCC Toolset

4.3.3. Exemple : Utilisation de l'image conteneur de la chaîne d'outils GCC Toolset 12

Cet exemple montre comment extraire et commencer à utiliser l'image conteneur GCC Toolset 12 Toolchain.

Conditions préalables

- Podman est installé.

Procédure

1. Accédez au Red Hat Container Registry à l'aide de vos identifiants du portail client :

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

2. Tirer l'image du conteneur en tant que root :

```
# podman pull registry.redhat.io/rhel9/gcc-toolset-12-toolchain
```

3. Lancer l'image du conteneur avec un shell interactif en tant que root :

```
# podman run -it registry.redhat.io/rhel9/gcc-toolset-12-toolchain /bin/bash
```

4. Exécutez les outils de la boîte à outils GCC comme prévu. Par exemple, pour vérifier la version du compilateur **gcc**, exécutez :

```
bash-4.4$ gcc -v
...
gcc version 12.1.1 20220628 (Red Hat 12.1.1-3) (GCC)
```

5. Pour lister tous les paquets fournis dans le conteneur, exécutez :

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

4.4. OUTILS DE COMPILATION

RHEL 9 fournit les ensembles d'outils de compilation suivants en tant que flux d'applications :

- LLVM Toolset fournit le cadre d'infrastructure du compilateur LLVM, le compilateur Clang pour les langages C et C, le débogueur LLDB et les outils connexes pour l'analyse du code.
- Rust Toolset fournit le compilateur du langage de programmation Rust **rustc**, l'outil de construction et le gestionnaire de dépendances **cargo**, le plugin **cargo-vendor** et les bibliothèques nécessaires.

- Go Toolset fournit les outils et les bibliothèques du langage de programmation Go. Go est également connu sous le nom de **golang**.

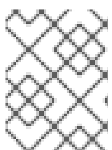
Pour plus de détails et d'informations sur l'utilisation, consultez les guides d'utilisation des jeux d'outils du compilateur sur la page Red Hat Developer Tools .

4.5. LE PROJET ANNOBIN

Le projet Annobin est une implémentation du projet de spécification Watermark. Le projet de spécification Watermark vise à ajouter des marqueurs aux objets ELF (Executable and Linkable Format) afin de déterminer leurs propriétés. Le projet Annobin se compose du plugin **annobin** et du programme **annockeck**.

Le plugin **annobin** analyse la ligne de commande de GNU Compiler Collection (GCC), l'état de la compilation et le processus de compilation, et génère les notes ELF. Les notes ELF enregistrent la manière dont le binaire a été construit et fournissent des informations permettant au programme **annocheck** d'effectuer des vérifications de renforcement de la sécurité.

Le vérificateur de renforcement de la sécurité fait partie du programme **annocheck** et est activé par défaut. Il vérifie les fichiers binaires pour déterminer si le programme a été construit avec les options de renforcement de la sécurité nécessaires et s'il a été compilé correctement. **annocheck** est capable d'analyser de manière récursive les répertoires, les archives et les paquets RPM à la recherche de fichiers objets ELF.



NOTE

Les fichiers doivent être au format ELF. **annocheck** ne gère aucun autre type de fichier binaire.

La section suivante décrit comment

- Utiliser le plugin **annobin**
- Utiliser le programme **annocheck**
- Supprimer les notes redondantes de **annobin**

4.5.1. Utiliser le plugin annobin

La section suivante décrit comment

- Activer le plugin **annobin**
- Transmettre des options au plugin **annobin**

4.5.1.1. Activation du plugin annobin

La section suivante décrit comment activer le plugin **annobin** via **gcc** et via **clang**.

Procédure

- Pour activer le plugin **annobin** avec **gcc**, utilisez :

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

  - Si **gcc** ne trouve pas le plugin **annobin**, utilisez :

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

    Remplacez */path/to/directory/containing/annobin/* par le chemin absolu vers le répertoire qui contient **annobin**.

  - Pour trouver le répertoire contenant le plugin **annobin**, utilisez :

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

- Pour activer le plugin **annobin** avec **clang**, utilisez :

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

  Remplacez */path/to/directory/containing/annobin/* par le chemin absolu vers le répertoire qui contient **annobin**.

4.5.1.2. Passer des options au plugin annobin

La section suivante décrit comment transmettre des options au plugin **annobin** via **gcc** et via **clang**.

Procédure

- Pour passer des options au plugin **annobin** avec **gcc**, utilisez :

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

  Remplacez *option* par les arguments de la ligne de commande **annobin** et remplacez *file-name* par le nom du fichier.

  Exemple :

  - Pour afficher des détails supplémentaires sur ce que **annobin** est en train de faire, utilisez :

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

    Remplacez *file-name* par le nom du fichier.

- Pour passer des options au plugin **annobin** avec **clang**, utilisez :

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

  Remplacez *option* par les arguments de la ligne de commande **annobin** et remplacez */path/to/directory/containing/annobin/* par le chemin absolu du répertoire contenant **annobin**.

  Exemple :

  - Pour afficher des détails supplémentaires sur ce que **annobin** est en train de faire, utilisez :

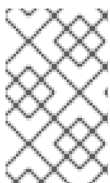
    ```
    $ clang -fplugin=/usr/lib64/clang/10/lib/annobin.so -Xclang -plugin-arg-annobin -Xclang
    verbose file-name
    ```

Remplacez *file-name* par le nom du fichier.

4.5.2. Utilisation du programme annocheck

La section suivante décrit comment utiliser **annocheck** pour examiner :

- Dossiers
- Annuaires
- Paquets RPM
- **annocheck** outils supplémentaires



NOTE

annocheck analyse récursivement les répertoires, les archives et les paquets RPM à la recherche de fichiers objets ELF. Les fichiers doivent être au format ELF. **annocheck** ne gère aucun autre type de fichier binaire.

4.5.2.1. Utiliser annocheck pour examiner les fichiers

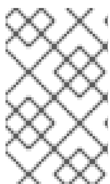
La section suivante décrit comment examiner les fichiers ELF à l'aide de **annocheck**.

Procédure

- Pour examiner un fichier, utilisez :

```
$ annocheck file-name
```

  Remplacez *file-name* par le nom d'un fichier.



NOTE

Les fichiers doivent être au format ELF. **annocheck** ne gère aucun autre type de fichier binaire. **annocheck** traite les bibliothèques statiques qui contiennent des fichiers objets ELF.

Informations complémentaires

- Pour plus d'informations sur **annocheck** et les options de ligne de commande possibles, voir la page de manuel **annocheck**.

4.5.2.2. Utiliser annocheck pour examiner les répertoires

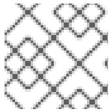
La section suivante décrit comment examiner les fichiers ELF dans un répertoire à l'aide de **annocheck**.

Procédure

- Pour scanner un répertoire, utilisez :

```
$ annocheck directory-name
```

Remplacez *directory-name* par le nom d'un répertoire. **annocheck** examine automatiquement le contenu du répertoire, de ses sous-répertoires et de toutes les archives et paquets RPM qui s'y trouvent.



NOTE

annocheck ne recherche que les fichiers ELF. Les autres types de fichiers sont ignorés.

Informations complémentaires

- Pour plus d'informations sur **annocheck** et les options de ligne de commande possibles, voir la page de manuel **annocheck**.

4.5.2.3. Utiliser annocheck pour examiner les paquets RPM

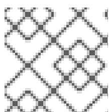
La section suivante décrit comment examiner les fichiers ELF dans un paquetage RPM à l'aide de **annocheck**.

Procédure

- Pour analyser un paquetage RPM, utilisez :

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

  Remplacez *rpm-package-name* par le nom d'un paquetage RPM. **annocheck** analyse récursivement tous les fichiers ELF contenus dans le paquetage RPM.



NOTE

annocheck ne recherche que les fichiers ELF. Les autres types de fichiers sont ignorés.

- Pour analyser un paquetage RPM avec l'info debug RPM fournie, utilisez :

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

  Remplacez *rpm-package-name* par le nom d'un paquet RPM et *debuginfo-rpm* par le nom d'un RPM d'informations de débogage associé au RPM binaire.

Informations complémentaires

- Pour plus d'informations sur **annocheck** et les options de ligne de commande possibles, voir la page de manuel **annocheck**.

4.5.2.4. Utiliser les outils supplémentaires d'annocheck

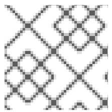
annocheck comprend plusieurs outils permettant d'examiner les fichiers binaires. Vous pouvez activer ces outils à l'aide des options de la ligne de commande.

La section suivante décrit comment activer la fonction :

- **built-by** outil
- **notes** outil

- **section-size** outil

Vous pouvez activer plusieurs outils en même temps.



NOTE

Le vérificateur de durcissement est activé par défaut.

4.5.2.4.1. Activation de l'outil **built-by**

Vous pouvez utiliser l'outil **annocheck built-by** pour trouver le nom du compilateur qui a construit le fichier binaire.

Procédure

- Pour activer l'outil **built-by**, utilisez

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

Informations complémentaires

- Pour plus d'informations sur l'outil **built-by**, voir l'option de ligne de commande **--help**.

4.5.2.4.2. Activation de l'outil **notes**

Vous pouvez utiliser l'outil **annocheck notes** pour afficher les notes stockées dans un fichier binaire créé par le plugin **annobin**.

Procédure

- Pour activer l'outil **notes**, utilisez

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

  Les notes sont affichées dans une séquence triée par plage d'adresses.

Informations complémentaires

- Pour plus d'informations sur l'outil **notes**, voir l'option de ligne de commande **--help**.

4.5.2.4.3. Activation de l'outil **section-size**

Vous pouvez utiliser l'outil **annocheck section-size** pour afficher la taille des sections nommées.

Procédure

- Pour activer l'outil **section-size**, utilisez

```
$ annocheck --section-size= (taille de la section)name
```

  Remplacez *name* par le nom de la section nommée. La sortie est limitée à des sections spécifiques. Un résultat cumulé est produit à la fin.

Informations complémentaires

- Pour plus d'informations sur l'outil **section-size**, voir l'option de ligne de commande **--help**.

4.5.2.4.4. Principes de base du vérificateur de trempe

Le vérificateur d'endurcissement est activé par défaut. Vous pouvez le désactiver à l'aide de l'option de ligne de commande **--disable-hardened**.

4.5.2.4.4.1. Options du vérificateur de durcissement

Le programme **annocheck** vérifie les options suivantes :

- Lazy binding est désactivée à l'aide de l'option **-z now** linker.
- Le programme n'a pas de pile dans une région exécutable de la mémoire.
- Les emplacements de la table GOT sont définis en lecture seule.
- Aucun segment de programme ne possède les trois bits de permission de lecture, d'écriture et d'exécution.
- Il n'y a pas de relocalisation par rapport au code exécutable.
- Les informations relatives au chemin d'exécution permettant de localiser les bibliothèques partagées au moment de l'exécution ne comprennent que les répertoires ayant pour racine /usr.
- Le programme a été compilé avec les notes **annobin** activées.
- Le programme a été compilé avec l'option **-fstack-protector-strong** activée.
- Le programme a été compilé avec **-D_FORTIFY_SOURCE=2**.
- Le programme a été compilé avec **-D_GLIBCXX_ASSERTIONS**.
- Le programme a été compilé avec **-fexceptions** activé.
- Le programme a été compilé avec **-fstack-clash-protection** activé.
- Le programme a été compilé à l'adresse **-O2** ou à une adresse supérieure.
- Le programme n'a pas de relocalisation maintenue dans un élément inscriptible.
- Les exécutables dynamiques ont un segment dynamique.
- Les bibliothèques partagées ont été compilées avec **-fPIC** ou **-fPIE**.
- Les exécutables dynamiques ont été compilés avec **-fPIE** et liés avec **-pie**.
- Si elle est disponible, l'option **-fcf-protection=full** a été utilisée.
- Si elle est disponible, l'option **-mbranch-protection** a été utilisée.
- Si elle est disponible, l'option **-mstackrealign** a été utilisée.

4.5.2.4.4.2. Désactivation du vérificateur de durcissement

La section suivante décrit comment désactiver le vérificateur de durcissement.

Procédure

- Pour scanner les notes d'un fichier sans le vérificateur de durcissement, utilisez :

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

  Remplacez *file-name* par le nom d'un fichier.

4.5.3. Suppression des notes annobines redondantes

L'utilisation de **annobin** augmente la taille des fichiers binaires. Pour réduire la taille des binaires compilés avec **annobin**, vous pouvez supprimer les notes redondantes de **annobin**. Pour supprimer les notes redondantes de **annobin**, utilisez le programme **objcopy**, qui fait partie du paquetage **binutils**.

Procédure

- Pour supprimer les notes redondantes de **annobin**, utilisez :

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

  Remplacez *file-name* par le nom du fichier.

4.5.4. Spécificités de annobin dans GCC Toolset 12

Dans certaines circonstances, en raison d'un problème de synchronisation entre **annobin** et **gcc** dans GCC Toolset 12, votre compilation peut échouer avec un message d'erreur qui ressemble à ce qui suit :

```
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
```

Pour contourner le problème, créez un lien symbolique dans le répertoire du plugin du fichier **annobin.so** vers le fichier **gcc-annobin.so**:

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

Remplacez *architecture* par l'architecture que vous utilisez dans votre système :

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