LFS : Lab 1 - Construire une chaîne d'outils de compilation croisée

Objet	construction d'une toolchain
Niveau requis	débutant, avisé
Features	
Suivi	:DONE:

Une chaîne de compilation (en anglais : « toolchain ») désigne l'ensemble des paquets utilisés dans le processus de compilation d'un programme, pour un processeur donné.

Description

Ce lab décrit la construction d'une toolchain pour un système AMD64 (c'est-à-dire x86_64) sur un CentOS 7.

L'ensemble du processus lui-même comprend les étapes suivantes:

- Installation des en-têtes du noyau dans le répertoire de la chaîne d'outils de sortie.
- Compiler des binutils croisés.
- Compiler un compilateur croisé GCC minimal avec un minimum de libgcc.
- Cross compilation de la bibliothèque standard C (dans notre cas musl) avec le minimum GCC.
- Compiler une version complète du compilateur croisé GCC avec libgcc complet.

La principale raison pour compiler deux fois GCC est l'interdépendance entre le compilateur et la bibliothèque standard.

- Tout d'abord, le système de compilation de GCC doit savoir quel type de bibliothèque standard C est utilisé et où le trouver. Non seulement le compilateur doit savoir à quoi lier les programmes, il lie également les programmes exécutables avec le code d'objet bootstrap. Fourni par la bibliothèque qui effectue la configuration de pile, appelle la fonction `main()` et appelle `exit(3)` lors du retour de `main()`. La bibliothèque fournit également l'éditeur de liens dynamique que le compilateur écrit dans le champ interpréteur ELF des programmes liés de manière dynamique.
- Deuxièmement, il y a `libgcc`: `libgcc` contient des aides spécifiques à la plate-forme de bas niveau (comme la gestion des exceptions, le code flottant, etc.) et est automatiquement lié aux programmes construits avec GCC. Le code source de Libgcc est fourni avec GCC et est compilé par le système de compilation GCC.

Certaines des implémentations de libc (comme glibc) utilisent directement des fonctions utilitaires de libgcc pour, par exemple, le déroulement de pile (`libgcc_s`).

Après avoir construit un compilateur croisé GCC, il faut compiler de manière croisée libgcc afin de pouvoir compiler d'autres éléments nécessitant `libgcc`, comme la libc, mais on a besoin d'une libc déjà compilée de manière croisée pour compiler `libgcc`. La solution consiste à:

1. Construire un GCC minimaliste **(GCC - PASS1)** avec lequel on compile une bibliothèque libgcc minimale avec de nombreuses fonctionnalités désactivées et utilisant des stubs internes pour les fonctions C standard au lieu de la liaison avec libc.
2. Ensuite on peut compiler la libc et laisser le compilateur la relier à la bibliothèque minimale libgcc.
3. Avec cela, on peut compiler le GCC complet **(GCC - PASS 2)** en le pointant sur la bibliothèque standard C du système cible et construire avec lui un libgcc couplé de manière dynamique et complet. On peut simplement l'installer par-dessus le GCC existant et libgcc dans le répertoire toolchain.

Prérequis

Le tableau ci-dessous compare certaines des différentes implémentations de bibliothèques standard disponibles pour Linux, en mettant l'accent sur les ressources utilisées.

Les packages source suivants sont requis pour la construction de la chaîne d'outils

- **Linux**: il s'agit d'un noyau très populaire fonctionnant sur notre système cible, il nous faut au moins les en-têtes pour construire la bibliothèque standard C.
- **Binutils**: contient l'assembleur GNU, l'éditeur de liens et divers outils permettant de travailler avec des fichiers exécutables.
- **GCC**: contient les compilateurs pour C et d'autres langages.
- **Musl**: Une minuscule implémentation de la bibliothèque standard C.

Le tableau ci-dessous compare les différents bibliothèques C sous l'angle des ressources matérielles (espace disque) requises :

Elément comparé	**musl**	**uClibc**	**dietlibc**	**glibc**
Complete .a set	426k	500k	120k	2.0M
Complete .so set	527k	560k	185k	7.9M
Smallest static C program	1.8k	5k	0.2k	662k
Static hello (using printf)	13k	70k	6k	662k
Dynamic overhead (min. dirty)	20k	40k	40k	48k
Static overhead (min. dirty)	8k	12k	8k	28k
Static stdio overhead (min. dirty)	8k	24k	16k	36k
Configurable featureset	no	yes	minimal	minimal

Les éléments suivants sont nécessaires à la compilation de GCC: il suffit de pointer le système de compilation GCC vers l'emplacement de la source et il s'occupe de la compilation.

- **MPFR**: une bibliothèque à virgule flottante à multiples précisions.
- **GMP**: une bibliothèque arithmétique à multiples précisions.
- **MPC**: bibliothèque de précisions multiples pour les nombres complexes.
- **ISL**: Matériel mathématique nécessaire pour optimiser les boucles.
- **CLOOG**: une bibliothèque de générateur de code.

Préparation du système hôte

Construire le dossier CLFS

```
CLFS=/var/lfs/clfs
install -dv ${CLFS}
```

Ajout de l'utilisateur CLFS

Construire un système from scratch en tant qu'utilisateur root est dangereux, car commettre une seule erreur peut endommager ou détruire le système hôte. Il est préférable de construire les packages en tant qu'utilisateur non privilégié. Pour faciliter la configuration d'un environnement de travail, créer un nouvel utilisateur appelé clfs en tant que membre d'un nouveau groupe (également appelé clfs) et utiliser cet utilisateur lors de la procédure d'installation:

```
groupadd clfs
useradd -s /bin/bash -g clfs -m -k /dev/null clfs
echo -e "clfspassword\nclfspassword" | passwd clfs
usermod -aG wheel clfs
chown -Rv clfs ${CLFS}
su - clfs
```

```
cat > ~/.bashrc << "EOF"
set +h
umask 022
CLFS=/var/lfs/clfs
LC_ALL=POSIX
PATH=${CLFS}/toolchain/bin:/bin:/usr/bin
export CLFS LC_ALL PATH
EOF
```

```
cat > ~/.bash_profile << "EOF"
exec env -i HOME=${HOME} TERM=${TERM} PS1='\u:\w\$ ' /bin/bash
EOF
```

```
source ~/.bash_profile
unset CFLAGS
echo unset CFLAGS >> ~/.bashrc
```

Définition des variables de construction

Au début, il faut définir quelques variables de shell pratiques pour stocker la configuration de la chaîne d’outils:

- Les variables **CPU** et **ARCH** contiennent l'architecture cible, la dernière étant utilisée pour le système de construction du noyau, l'ancienne pour le système de construction GNU, car elles ne peuvent pas décider d'un schéma commun pour nommer des éléments.
- La variable **TARGET** contient le triplet cible du système. Elle décrit la plate-forme cible en collant ensemble l'architecture du processeur, le noyau et l'utilisateur. Cette chaîne n'est pas arbitraire! Le système de construction GNU analyse cette chaîne pour comprendre son objectif! Pour compiler une chaîne d'outils musl, la dernière partie doit être **musl**! Sinon, le système de compilation de GCC supposera un autre fournisseur de libc et le second passage, GCC, explosera!
- La variable **HOST** contient le triplet de la machine locale sur laquelle on va construire des éléments, que l'on utilisera plus tard lors de la construction de GCC. la variable **OSTYPE** utilisée pour construite le triplet est une variable shell intégrée. Certains guides suggèrent d'utiliser une autre variable shell intégrée **MACHTYPE**, mais cela risque de donner des résultats incohérents.

```
ARCH="x86"
CPU="i686"
HOST=$(uname -m)-$OSTYPE
export TARGET=i686-linux-musl
export HOST=$MACHTYPE
export TCDIR=${CLFS}/toolchain
export CLFS="${CLFS}"
sudo chmod 777 ${CLFS}
```

Construction de l'environnement

```
mkdir -pv ${CLFS}/{src,download,toolchain,targetfs}
```

Téléchargement des paquets

Télécharger les packages dans le répertoire ${CLFS}/download.

Confection de la liste des paquets

```
cat >> ${CLFS}/download/dl-src.list << EOF
http://ftp.gnu.org/gnu/binutils/binutils-2.27.tar.bz2
https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.20.12.tar.xz
http://ftp.gnu.org/gnu/gcc/gcc-8.2.0/gcc-8.2.0.tar.xz
http://ftp.gnu.org/gnu/gmp/gmp-6.1.2.tar.xz
http://www.mpfr.org/mpfr-4.0.2/mpfr-4.0.2.tar.xz
https://ftp.gnu.org/gnu/mpc/mpc-1.1.0.tar.gz
http://isl.gforge.inria.fr/isl-0.16.1.tar.xz
ftp://gcc.gnu.org/pub/gcc/infrastructure/cloog-0.18.1.tar.gz
http://www.musl-libc.org/releases/musl-1.1.23.tar.gz
EOF
```

Téléchargement des paquets et des correctifs

```
wget -i ${CLFS}/download/dl-src.list -P ${CLFS}/download
```

Construction toolchain transitoire

Compilation de cross binutils

Le paquet Binutils contient un éditeur de liens, un assembleur et d'autres outils pour gérer des fichiers objets.

```
TIMEFORMAT='(TOOLCHAIN) Compilation de cross binutils en %R seconds ...'
time {
tar xjf ${CLFS}/download/binutils-2.27.tar.bz2 -C ${CLFS}/src &&
mkdir -p "${CLFS}/build/binutils" &&
cd "${CLFS}/build/binutils" &&
srcdir="${CLFS}/src/binutils-2.27" &&
$srcdir/configure --prefix="$TCDIR" --target="$TARGET" \
                  --with-sysroot="$TCDIR/$TARGET" \
                  --disable-nls --disable-multilib &&
make &&
make install &&
cd "${CLFS}"
}
```

Compilation de cross binutils en 239.663 seconds ...

Extraction des en-têtes du noyau

Les Linux API Headers (en-têtes API de Linux, incluses dans linux-4.20.12.tar.xz) montrent l'API du noyau pour qu'il soit utilisé par Glibc.

Comme GCC il faut construire le noyau en dehors de son arborescence source, mais cela fonctionne un peu différemment par rapport aux outils basés sur autotools.

Selon le Makefile de la source Linux, on peut spécifier une variable d’environnement appelée KBUILD_OUTPUT ou définir une variable de Makefile appelée O, où celle-ci remplace la variable d’environnement.

La cible `headers_check` exécute quelques contrôles de cohérence simples sur les en-têtes que l'on va installer: elle vérifie si un en-tête contient quelque chose d’existant, si les déclarations à l’intérieur des en-têtes sont saines et si des interna du noyau sont perdus dans l’espace utilisateur.

Enfin (avant de revenir au répertoire racine), on installe les en-têtes du noyau, par exemple, dans `toolchain/i686-linux-musl/include`, où la libc les attend ensuite.

peu importe que la version du noyau corresponde exactement à celle qui s'exécute sur le système cible. L'appel système du noyau ABI est stable, on peut donc utiliser un noyau plus ancien. Au contraire, si on utilise un noyau beaucoup plus récent, la libc pourrait finir par exposer ou utiliser des fonctionnalités que le noyau ne prend pas encore en charge.

Pour localiser tous les fichiers de sortie dans un répertoire séparé, les deux syntaxes supportées doivent être utilisées: définir la variable d'environnement `KBUILD_OUTPUT` pour qu'elle pointe vers le répertoire où les fichiers de sortie doivent être placés, `export KBUILD_OUTPUT=dir/to/store/output/files/`, **ET** utiliser l'option `O=` dans l'intruction make, `make O=$KBUILD_OUTPUT` (le répertoire de travail doit être la racine src du noyau).

```
TIMEFORMAT='(TOOLCHAIN) Compilation des en têtes du noyau en %R seconds ...'
time {
export KBUILD_OUTPUT="$CLFS/build/linux" &&
mkdir -pv "$KBUILD_OUTPUT" &&
tar xf ${CLFS}/download/linux-4.20.12.tar.xz -C ${CLFS}/src &&
cd ${CLFS}/src/linux-4.20.12 &&
make mrproper &&
make O="$KBUILD_OUTPUT" ARCH="$ARCH" headers_check &&
make O="$KBUILD_OUTPUT" ARCH="$ARCH" INSTALL_HDR_PATH="$TCDIR/$TARGET"
headers_install &&
cd "${CLFS}"
}
```

Compilation des en têtes du noyau en 54.210 seconds ...

Installation GCC - PASS 1

Le paquet GCC contient la collection de compilateurs GNU, qui inclut les compilateurs C.

Le premier pass construit uniquement les outils nécessaires à la compilation du second pass.

Dans un système de génération d'autotools, trois triplets de système différents sont à l'œuvre:

- L'option **–build** spécifie le système sur lequel son construit les paquets.
- L'option **–host** spécifie le système sur lequel les fichiers binaires seront exécutés.
- L'option **–target** est spécifique aux outils de compilation et spécifie le système pour lequel générer une sortie.

Seule l'option **–target** pour indiquer au système de construction la cible pour laquelle l'assembleur, l'éditeur de liens et d'autres outils doivent générer une sortie, car le système de construction binutils est un peu plus robuste que celui de GCC et Peut comprendre qu'il est construit pour la machine locale.

Pour construire une toolchain pour un autre système, il faut définir l'option **–host** sur le triplet de la chaîne d'outils croisés existante afin de construire des binutils qui fonctionnent sur une machine différente et génèrent une sortie pour une autre.

L'option **–prefix** spécifie où installer les fichiers, en même temps que la variable make DESTDIR. Lib, en-têtes vers xy/prefix/include, etc. Le suffixe spécifique au type de fichier peut bien entendu être configuré, mais cela n'a pas vraiment d'intérêt à l'heure actuelle.

Le préfixe par défaut est `/usr/local/`. On le place dans le répertoire de niveau supérieur de la chaîne d'outils (TCDIR=$CLFS/toolchain).

L'option **–with-sysroot** indique au système de construction que le répertoire racine du système n'est pas '/' mais bien '$TCDIR/$TARGET' (par exemple, "toolchain/i686-linux-musl") et qu'il devrait rechercher des bibliothèques et des en-têtes. Là-bas.

Les fonctionnalités nls (prise en charge de la langue maternelle, c'est-à-dire i18n) sont désactivées parce qu'on en a pas besoin.

Certaines architectures prennent en charge l'exécution de code pour d'autres architectures apparentées (par exemple, le code x86 peut exécuter x86_64). Sur les distributions GNU/Linux qui le supportent, on dispose généralement de versions différentes des mêmes bibliothèques (par exemple, dans les répertoires lib/ et lib32/) avec des programmes pour Différentes architectures étant reliées aux bibliothèques appropriées. Parceque'on ne veut intéressés qu'une architecture unique et n'en avons pas besoin, on a donc défini **–disable-multilib**.

```
TIMEFORMAT='(TOOLCHAIN) Compilation du compilateur Cross GCC - PASS 1 en %R
seconds ...'
time {
tar -xf ${CLFS}/download/gcc-8.2.0.tar.xz -C ${CLFS}/src &&
tar -xf ${CLFS}/download/mpfr-4.0.2.tar.xz -C ${CLFS}/src &&
tar -xf ${CLFS}/download/gmp-6.1.2.tar.xz -C ${CLFS}/src &&
tar -xzf ${CLFS}/download/mpc-1.1.0.tar.gz -C ${CLFS}/src &&
tar -xf ${CLFS}/download/isl-0.16.1.tar.xz -C ${CLFS}/src &&
tar -xzf ${CLFS}/download/cloog-0.18.1.tar.gz -C ${CLFS}/src &&
cd ${CLFS}/src/gcc-8.2.0 &&
ln -s "${CLFS}/src/gmp-6.1.2" "gmp" &&
ln -s "${CLFS}/src/mpc-1.1.0" "mpc" &&
ln -s "${CLFS}/src/mpfr-4.0.2" "mpfr" &&
ln -s "${CLFS}/src/cloog-0.18.1" "cloog" &&
ln -s "${CLFS}/src/isl-0.16.1" "isl" &&
mkdir -p "${CLFS}/build/gcc-1" &&
cd "${CLFS}/build/gcc-1" &&
srcdir="${CLFS}/src/gcc-8.2.0" &&
$srcdir/configure --prefix="$TCDIR" --target="$TARGET" \
                  --build="$HOST" --host="$HOST" \
                  --with-sysroot="$TCDIR/$TARGET" \
```

```
                  --disable-nls --disable-shared --without-headers \
                  --disable-multilib --disable-decimal-float \
                  --disable-libgomp --disable-libmudflap \
                  --disable-libssp --disable-libatomic \
                  --disable-libquadmath --disable-threads \
                  --enable-languages=c --with-newlib --with-arch="$CPU" &&
make all-gcc all-target-libgcc &&
make install-gcc install-target-libgcc &&
cd "${CLFS}"
}
```

(TOOLCHAIN) Compilation du compilateur Cross GCC - PASS 1 en 2383.507 seconds ...

Installation de MUSL

```
TIMEFORMAT='(TOOLCHAIN) Compilation de la librairie standard C en %R seconds
...'
time {
tar xzf ${CLFS}/download/musl-1.1.23.tar.gz -C ${CLFS}/src &&
mkdir -p "${CLFS}/build/musl" &&
cd "${CLFS}/build/musl"
srcdir="${CLFS}/src/musl-1.1.23"
CC="${TARGET}-gcc" $srcdir/configure --prefix=/ --target="$TARGET" &&
CC="${TARGET}-gcc" make &&
make DESTDIR="$TCDIR/$TARGET" instal l&&
cd "${CLFS}"
}
```

(TOOLCHAIN) Compilation de la librairie standard C en 132.522 seconds ...

Installation GCC - PASS 2

Pour la deuxième étape, on construit également un compilateur C++, les options **–enable-c99** et **–enable-long-long** sont spécifiques à C++ Lorsque le compilateur final fonctionne en mode C++ 98, on lui permet d’exposer les fonctions C99. De la libc à une extension GNU, on lui permet également de supporter le type de données **long long** normalisé en C99.

On a pas besoin de créer `libstdc++` entre la première et la deuxième passe, comme la libc. Le code source de la `libstdc++` est fourni avec le compilateur G++ et est construit automatiquement comme libgcc. Seule une bibliothèque qui ajoute des éléments C++ au-dessus de libc et du compilateur n'en dépend pas. C++, par contre, ne dispose pas d'une ABI standard et est entièrement spécifique au compilateur et au système d'exploitation. Les constructeurs de compilateurs `Libstdc++` implémentation avec le compilateur.

Les options **–disable-libmpx** et **–disable-libssp** sont des hacks spéciaux nécessaires à la construction d'un compilateur croisé x86 sur AMD64. Ces deux bibliothèques sont utilisées dans la génération de code pour utiliser certaines fonctionnalités du jeu d'instructions 64 bits. Suffisamment intelligent pour compiler ces bibliothèques pour la cible x86 (car elle n’a tout simplement pas ces fonctionnalités de processeur), mais pour une raison quelconque tente de lier le compilateur final aux bibliothèques, générant une erreur de liaison. Désactiver ces bibliothèques empêchera cela.

Spécifier l'option **–disable-libsanitizer** lorsqu'on veut utiliser uniquement musl car celui-ci fournit “uniquement” un plugin d'analyse de code statique pour le compilateur C++.

L'option **–with-native-system-header-dir** est d'un intérêt particulier pour un compilateur croisé. Comme on indique le répertoire racine à `$TCDIR/$TARGET`, le compilateur recherchera les en-têtes dans `$TCDIR/$TARGET/usr/include`, mais on ne les a pas installé dans `/usr/include`, on les a installé dans `$TCDIR/$TARGET/include`, il faut donc indiquer au système de construction qu'il doit rechercher dans `/include` (par rapport au répertoire racine) .

```
TIMEFORMAT='(TOOLCHAIN) Compilation du compilateur Cross GCC - PASS 2 en %R
seconds ...'
time {
mkdir -p "${CLFS}/build/gcc-2" &&
cd "${CLFS}/build/gcc-2" &&
srcdir="${CLFS}/src/gcc-8.2.0" &&
$srcdir/configure --prefix="$TCDIR" --target="$TARGET" \
                  --build="$HOST" --host="$HOST" \
                  --with-sysroot="$TCDIR/$TARGET" \
                  --disable-nls --enable-languages=c,c++ \
                  --enable-c99 --enable-long-long \
                  --disable-libmudflap --disable-multilib \
                  --disable-libmpx --disable-libssp --disable-libsanitizer \
                  --with-arch="$CPU" \
                  --with-native-system-header-dir="/include" &&
make &&
make install &&
cd "${CLFS}"
}
```

(TOOLCHAIN) Compilation du compilateur Cross GCC - PASS 2 en 2984.533 seconds ...

Test de la Toolchain

Créer un programme `hello world` moyen dans un fichier appelé test.c:

```
cat > test.c << "EOF"
#include <stdio.h>

int main(void)
{
    puts("Hello, world");
    return 0;
}
EOF
```

On peut maintenant utiliser le compilateur croisé pour compiler ce fichier C:

```
${TARGET}-gcc test.c
```

L'exécution de la commande `file` sur le fichier `a.out` résultant dira qu'il a été correctement compilé et lié pour la machine cible:

La commande `file a.out` devrait retourner :

```
file a.out
a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked (uses shared libs), not stripped
```

Bien sûr, on ne peut pas exécuter le programme sur le système hôte, à l'exception peut-être de la version x86 qui fonctionnera sous x86_64 si on le compile de manière complètement statique:

```
${TARGET}-gcc -static -static-libgcc test.c
```

Nettoyage de l'arbre de construction

A l'issue des tests les dossiers `src` et `build` peuvent être détruits.

```
rm ${CLFS}/src -rf
rm ${CLFS}/build -rf
exit
```