TP n°9 : Environnement Utilisateur

A) Découverte des librairies partagées

1. Extraire le fichier archive /home/shared/gregoirep/COURS-TP9/dummy.tgz

```
tar xvf  /home/shared/gregoirep/COURS-TP9/dummy.tgz

cd dummy
```

2. Compiler le fichier foo pour en faire une librairie partagée :

```
gcc -c -Wall -Werror -fpic foo.c

gcc -shared -o libfoo.so foo.o
```

3. Compiler l'application ; Que se passe t-il ? Compléter la seconde commande gcc pour que l'édition de liens se fasse correctement.

```
gcc -c -Wall main.c

gcc  -o test main.o ……. -lfoo
```

4. Lancer l'application . Expliquer ce qui se passe et l'origine du problème

```
./test
```

5. Relancer l'application avec strace : Combien de répertoires sont explorés pour trouver la librairie foo ?

```
strace -e open ./test
```

6. Exporter la variable LD_LIBRARY_PATH pour permettre l'exécution correcte du programme test :

```
 LD_LIBRARY_PATH=XXXXX ./test
```

7. Relancer l'application avec la variable LD_LIBRARY_PATH et strace : Combien de répertoires sont explorés pour trouver la librairie foo ?

```
strace -E LD_LIBRARY_PATH=XXXXX -e open ./test
```

8. Compiler le programme avec l'option rpath :

```
gcc  -o test_rpath main.o -L$PWD -lfoo -Wl,-rpath=$PWD
```

9. Vérifier que le programme test_rpath s'exécute bien quelque soit le répertoire courant :

```
unset LD_LIBRARY_PATH ; ./test_rpath

( export D=$PWD ; cd /tmp ; $D/test_rpath ; )
```

10. La commande ldd montre les librairies utilisées par un prorgamme. Comparer les resultats sur les programmes test et test_rpath :

```
ldd ./test
```

```
ldd ./test_rpath
```

11. Que se passe t-il si on utilise un path relatif avec l'option rpath ?

```
gcc  -o test_rpath_rel main.o -L$PWD -lfoo -Wl,-rpath=.
```

12. Un code utilise 20 librairies différentes dont les répertoires d'installation sont spécifiés dans la variable LD_LIBRARY_PATH au moment de son lancement. En vous appuyant sur les résultats obtenus en 7/ donner une estimation du nombre maximum d'accès à un système de fichiers NFS qui contiendrait ces 20 répertoires dans le cas d'un lancement sur 5000 cœurs. Commentez.

B) Utilisation de la commande module

1. Lister les modules disponibles sur la machine hpc01
2. Charger le module **openmpi**.
3. Où est trouvée la commande **mpicc** ? Quel paquet fournit cette commande ?
4. Avec le commande module, afficher la définition du module que vous venez de charger.
5. Avec la commande rpm et une regex, trouvez tous les rpms qui installent des versions MPI.
6. Une version openMPI-2 est déjà installée. Dans que répertoire ? Vérifier si ce paquet fournit des fichiers module.
7. Quel fichier contient la définition du module openmpi existant ?
8. Créer le répertoire **$HOME/modulefiles** Utilisez le fichier trouvé précédemment comme modèle pour créer un module pour la version openmpi-2.0.3.
9. Utilisez la commande module pour ajouter le répertoire **$HOME/modulefiles** à la liste des répertoires utilisés par la commande module. Vérifier que votre module apparaît dans la liste des modules disponibles
10. Charger ce nouveau module mpi. Que se passe t-il ?
11. En une seule commande module, changer de version d'openmpi.
12. Vérifier où est trouvée la commande **mpicc** maintenant.

C) Installation et utilisation d'EasyBuild

Vous allez installer le framework EasyBuild dans votre répertoire HOME sur la machine hpc01 :

1. Récupérer le script de bootstrap d'easybuild :

   ```
   wget  https://raw.githubusercontent.com/easybuilders/easybuild-framework/develop/easybuild/scripts/bootstrap_eb.py
   ```

2. Définir les variables d'environnement pour que EasyBuild s'installe dans le répertoire $HOME/local/easybuild

   ```
   EASYBUILD_PREFIX=$HOME/local/easybuild

   python bootstrap_eb.py $EASYBUILD_PREFIX
   ```

3. Suivre les instructions données à la fin du bootstrap pour pouvoir utiliser **easybuild** qui vient d'être généré lui même comme un logiciel utilisable via module . Vous avez une seule variables d'environnement à définir.

   ```
   ...
   ```

4. Vérifier avec la commande ***module*** que le module ***easybuild*** est disponible

   ```
   ...
   ```

5. Charger le module ***easybuild*** avec la commande module et exécuter la commande **eb** (**e**asy**b**uild) avec l'option –version

   ```
   ...
   ```

6. Générer un fichier de configuration commenté :

   ```
   mkdir $HOME/.config/easybuild

   eb --confighelp  $HOME/.config/easybuild/config.cfg
   ```

7. Éditer le fichier $HOME/.config/easybuild/config.cfg et modifier le pour
   - choisir la syntaxe module Tcl,
   - le modèle de commande module qui a été détecté lors du bootstrap
   - le répertoire d'installation $HOME/local (mettre le path complet)

   ```
   vim  $HOME/.config/easybuild/config.cfg

   grep -v '^#' $HOME/.config/easybuild/config.cfg
   ```

8. Vérifier que votre fichier de configuration est bien pris en compte (**mettre en gras ce qui le prouve**) :

   ```
   eb --show-default-configfiles
   ```

9. Vérifier que votre paramétrage est bien pris en compte (**mettre en gras ce qui le prouve**) :

```
eb --show-config
```

10. Consulter le help de la commande et donner la liste des toolchains ; Quelle est la composition des toolchains dummy , ictce, iomkl et gompi? Quel est parmi ces 4 là celui qui n'utilise que des produits Intel ?

11. Donner la liste des easyblocks (les 10 premières lignes)

12. Utiliser la commande **eb** pour chercher un easyblock pour flex-2.6.4

13. Sélectionner le fichier easyblock qui correspond à la toolchain dummy et lancer un build en dry-run ; Donner les noms et versions des logiciels qui seront générés en même temps à cause des dépendances.

14. Lancer la commande eb avec la bonne option pour générer cette version du logiciel flex et ses dépendances.

15. Utilisez la commande module pour ajouter le répertoire **$HOME/local/modules/all** à la liste des répertoires utilisés par la commande module. Montrer la liste des modules disponible

16. Charger le module flex que vous venez de générer. Afficher le contenu du module flex. Donner le path de la commande flex. Executer la commande flex :

```
flex --version
```