configure ; make ; make install

Linux Gazette n°97 — Décembre 2003

Willy Smith

Copyright © 2003 Willy Smith

Encore et toujours, j'entends répéter autour de moi qu'il suffit d'utiliser la traditionnelle séquence « configure, make, make install » pour faire fonctionner un programme. Malheureusement, la plupart des personnes qui utilisent des ordinateurs aujourd'hui n'ont jamais fait appel à un compilateur ou écrit une ligne de code de programme. Avec l'avènement des interfaces utilisateur et des générateurs d'applications graphiques, de nombreux programmeurs de premier ordre n'ont jamais effectué ces opérations.

Il s'agit d'un processus en trois étapes dont chacune utilisera un grand nombre de programmes afin d'en faire fonctionner un nouveau. L'utilisation de la commande **configure** est relativement récente par rapport à celle de **make**. Toutefois, chaque étape a un objectif distinct. Je vais commencer par expliquer les deuxième et troisième étapes, puis je reviendrai sur **configure**.

La commande **make** fait partie intégrante de l'histoire d'Unix. Elle a été conçue afin de réduire la nécessité, pour un programmeur, de mémoriser des informations. Je suppose que c'est surtout une façon élégante de dire qu'elle réduit les contraintes de documentation du programmeur. Quoi qu'il en soit, l'idée est que si vous établissez un ensemble de règles pour créer un programme dans un format que **make** comprend, vous n'avez plus besoin de les garder en mémoire ensuite.

Pour faciliter encore les choses, **make** comporte un ensemble de règles intégrées auxquelles il ne vous reste à indiquer que les nouveaux éléments dont **make** a besoin pour compiler votre utilitaire. Par exemple, si vous saisissez **make** *`aimer`*, **make** commence par rechercher de nouvelles règles que vous auriez ajoutées. Si vous n'en avez fourni aucune, **make** effectue une recherche dans ses règles intégrées. L'une de ces dernières indique à **make** qu'il peut lancer l'éditeur de liens (ld) sur un nom de fichier se terminant par `.o` pour produire le programme exécutable.

Ainsi, la commande **make** commence par rechercher un fichier nommé *`aimer.o`*. Mais elle ne s'arrête pas là. Même si elle trouve le fichier `.o`, elle comporte d'autres règles lui demandant de s'assurer que le fichier `.o` est à jour. En d'autres termes, plus récent que le programme source. D'une façon générale, le programme source sur les systèmes Linux est écrit en langage C et son nom de fichier se termine par `.c`.

Si la commande **make** trouve le fichier `.c` (*`aimer.c`* dans notre exemple) ainsi que le fichier `.o`, elle vérifie leurs horodatages respectifs pour s'assurer que le `.o` est plus récent. Dans le cas contraire ou si `o.` n'existe pas, elle utilise une autre règle intégrée pour construire un nouveau `.o` à partir du `.c` (à l'aide du compilateur C). Le même type de situation se reproduit pour d'autres langages de programmation. Dans tous les cas, le résultat final est que lorsque **make** a terminé (partant de l'hypothèse qu'elle peut trouver tous les éléments nécessaires), l'exécutable est construit et à jour.

À propos, la traditionnelle plaisanterie UNIX sur le sujet est la réponse que les premières versions de **make** donnaient quand elles ne trouvaient pas les fichiers nécessaires. Dans l'exemple ci-dessus, s'il n'y avait aucun fichier *`aimer`*`.o`, *`aimer`*`.c` ou tout autre format source, le programme aurait répondu :

Revenons à nos préoccupations, le fichier par défaut pour enregistrer des règles complémentaires dans le `Makefile` contenu dans le répertoire actuel. Si vous avez certains fichiers sources d'un programme et s'il y a parmi eux un fichier `Makefile`, jetez-y un coup d'½il. C'est simplement du texte. Les lignes qui comportent un mot suivi d'un point-virgule sont des cibles : ce sont des mots que vous pouvez saisir à la suite de la commande **make** pour lui faire faire différentes tâches. Si vous saisissez simplement **make** sans aucune cible, c'est la première cible qui sera exécutée.

Ce que, selon toute probabilité, vous trouverez au début de la plupart des fichiers `Makefile` ressemble à des instructions d'affectation. Il s'agit de lignes comportant deux champs séparés par un signe égal. Et, ô surprise, ce sont effectivement des instructions d'affectation qui attribuent des valeurs à des variables internes de **make**. Parmi les variables qu'il est généralement nécessaire de renseigner, il y a l'emplacement du compilateur C (oui, il y a une valeur par défaut), les numéros de version du programme, etc.

Tout ceci nous ramène à **configure**. Sur d'autres systèmes, le compilateur C se trouvera à un autre endroit, vous utiliserez ZSH comme shell au lieu de BASH, le programme pourra avoir besoin de connaître votre nom d'hôte, avoir à utiliser une bibliothèque dbm et avoir besoin de savoir si le système comporte gdbm ou ndbm, et quantité d'autres choses encore. Auparavant, tout ce travail de configuration s'effectuait en éditant le fichier `Makefile`. Une tâche de plus pour le programmeur impliquant de surcroît que, à chaque installation d'un logiciel sur un nouveau système, il fallait faire un inventaire complet de tout ce qui s'y trouvait.

À mesure que de plus en plus de logiciels devenaient disponibles et que de plus en plus de plates-formes conformes à la norme POSIX apparaissaient, cette opération est devenue de plus en plus difficile. Et c'est là où la commande **configure** entre en scène. Il s'agit d'un script shell (généralement écrit par Autoconf de GNU) qui s'exécute, recherche les éléments logiciels et effectue même divers essais pour savoir ce qui fonctionne. Il prend ensuite ses instructions dans `Makefile.in` et construit le fichier `Makefile` (et éventuellement quelques autres fichiers) qui fonctionnera correctement sur ce système.

Maintenant que nous avons terminé les présentations, rassemblons toutes les éléments pour avoir une vue d'ensemble :

- Exécutez **configure** (vous devrez en général saisir **./configure** car la plupart des utilisateurs n'ont pas le répertoire en cours dans leur chemin de recherche). Cette commande construit un nouveau fichier `Makefile`.
- Saisissez ensuite **make**. Cette commande construit le programme. Autrement dit, **make** s'exécute, recherche la première cible dans le fichier `Makefile` et obéit aux instructions indiquées. Le résultat final attendu est la construction d'un programme exécutable.
- Maintenant, en tant que root, saisissez **make install**. Cette commande invoque à nouveau **make**, qui recherche la cible `install` dans le `Makefile` et suit les instructions pour installer le programme.

Il s'agit là d'une explication très simplifiée mais qui, dans la plupart des cas, représente ce que vous devez savoir. La plupart des programmes comportent un fichier nommé `INSTALL` contenant des instructions d'installation qui vous informent d'autres considérations. Par exemple, il est courant de fournir certaines options à la commande **configure** pour lui faire changer l'emplacement final du programme exécutable. D'autres cibles pour **make** existent également, telles que **clean** qui supprime les fichiers inutiles après une installation et **test** qui, dans certains cas, vous permet de tester le logiciel entre les étapes **make** et **make install**.

Copyright © 2003, Willy Smith.

Copying license http://www.linuxgazette.com/copying.html

Paru dans le n°97 de la Linux Gazette de décembre 2003.

Traduction française par Sandrine Burriel `<girlgeek CHEZ free POINT fr>`.

Relecture de la traduction française par Joëlle Cornavin `<jcornavi CHEZ club TIRET internet POINT fr>`.