Construire un système Linux minimum à partir du code source

Greg O'Keefe

<gcokeefe@postoffice.utas.edu.au>

Dominique van den Broeck – Traduction française

Jean–Philippe Guérard – Relecture de la version française

v0.9, novembre 2000

Voici les instructions pour construire un système Linux minimum à partir du code source. Ce document faisait partie du guide pratique « De la mise sous tension à l'invite de commande de Bash », mais j'ai choisi d'en faire un document indépendant, afin que chacun de ces 2 documents restent courts et concentrés. Le système que nous construisons ici est réduit au minimum et n'est pas apte à réaliser un vrai travail. Si vous voulez monter un vrai système, lisez plutôt le Comment faire un système Linux à partir de zéro (*Linux from scratch − LFS*).

Table des matières

1. Ce qu'il vous faut

Nous installerons une distribution de Linux telle que Red Hat sur une partition, et l'utiliserons pour construire un nouveau système Linux sur une autre partition. Je nommerai par la suite « cible » le système que nous construisons, et « source » le système que nous utilisons pour construire le système cible (à ne pas confondre avec *code source* que nous utiliserons aussi).

Vous allez donc avoir besoin d'une machine avec deux partitions libres. Si vous le pouvez, utilisez une machine qui ne contienne rien d'important. Vous pouvez utiliser un système Linux déjà existant comme système source, mais je le déconseille. Si vous oubliez un des paramètres des commandes que nous allons saisir, vous pourriez accidentellement réinstaller des choses sur votre système source. Cela peut mener à des incompatibilités, et des conflits.

Les BIOS des PC dont l'architecture est ancienne, pour la plupart des 486 et des machines antérieures, ont une limitation ennuyeuse. Ils ne peuvent lire les disques durs au−delà des 512 premiers méga−octets. Ce n'est pas vraiment un problème pour Linux, qui gère lui−même les disques une fois lancé. Mais pour que Linux soit chargé sur ces vieilles machines, le noyau doit résider quelque part en−dessous de 512 méga−octets. Si vous utilisez une de ces machines, vous devrez créer une partition distincte en−dessous de 512 Mo, à monter sur `/boot` pour chaque système dont la partition racine se situe au−dessus de la limite des 512 Mo.

La dernière fois que je l'ai fait, j'ai utilisé Red Hat 6.1 comme système source. J'ai installé le système de base plus :

- cpp
- egcs
- egcs−c++
- patch
- make
- dev86
- ncurses−devel
- glibc−devel
- kernel−headers

J'ai aussi installé X−Window et Mozilla pour pouvoir lire la documentation facilement, mais ce n'est pas vraiment nécessaire. À la fin de mon travail, celui−ci avait pris environ 350 Mo d'espace disque (cela semble un peu élevé, je me demande pourquoi).

Le système cible achevé prenait 650 Mo, mais comprenait tout le code source et les fichiers intermédiaires. Si l'espace est limité, je vous conseille de faire un **make clean** après la construction de chaque paquet. Cela dit, cette taille surprenante est un peu inquiétante.

Enfin, vous allez avoir besoin du code source du système que vous allez construire. Ce sont les paquets dont nous avons parlé dans le guide pratique « De la mise sous tension à l'invite de commande de Bash ». On peut les obtenir depuis un CD, ou par l'Internet. Je donnerai les url de leurs sites américains et des miroirs français.

- MAKEDEV : ftp://sunsite.unc.edu/pub/Linux/system/admin/ (USA), ftp://ftp.lip6.fr/pub/linux/sunsite/system/admin/ (France).
- Lilo : ftp://lrcftp.epfl.ch/pub/linux/local/lilo/ (Suisse), ftp://ftp.lip6.fr/pub/linux/sunsite/system/boot/lilo/ (France).
- Noyau Linux : utilisez un des miroirs listés sur http://www.kernel.org plutôt que ftp://ftp.kernel.org/pub/linux/kernel/ (USA) car ils sont toujours en surcharge ; ftp://ftp.fr.kernel.org/pub/linux/kernel/ (France).
- GNU libc : la bibliothèque elle−même, ainsi que les extensions linuxthreads sont sur ftp://ftp.gnu.org/pub/gnu/glibc/ (USA), ftp://ftp.lip6.fr/pub/gnu/glibc/ (France).
- Extensions de la libc GNU : vous aurez aussi besoin des linuxthreads et des extensions libcrypt. Si libcrypt est absente du fait des lois américaines sur l'exportation, vous pouvez la récupérer sur ftp://ftp.gwdg.de/pub/gnu/glibc les extensions linuxthreads sont au même endroit que la libc proprement dite.
- GNU ncurses : ftp://ftp.gnu.org/gnu/ncurses (USA), ftp://ftp.lip6.fr/pub/gnu/ncurses (France).

- SysVinit : ftp://sunsite.unc.edu/pub/Linux/system/daemons/init/ (USA), ftp://ftp.lip6.fr/pub/linux/sunsite/system/daemons/init/ (France).
- GNU Bash : ftp://ftp.gnu.org/gnu/bash/ (USA), ftp://ftp.lip6.fr/pub/gnu/bash/ (France).
- GNU sh−utils : ftp://ftp.gnu.org/gnu/sh−utils/ (USA), ftp://ftp.lip6.fr/pub/gnu/sh−utils/ (France).
- util−linux : ftp://ftp.win.tue.nl/pub/linux/utils/util−linux/ (Pays−Bas), ftp://ftp.lip6.fr/pub/linux/sunsite/system/misc/ (France). Ce paquet contient agetty et login.

Pour résumer, il vous faut :

- Une machine avec deux partitions distinctes d'environ 400 Mo et 700 Mo respectivement, bien que vous puissiez sûrement vous en tirer avec un espace plus restreint.
- Une distribution de Linux (par exemple, un CD Red Hat), et de quoi l'installer (par exemple, un lecteur de CD).
- Les archives tar de code source listées ci−dessus. (Le format tar permet de regrouper plusieurs fichiers en un seul. Un fichier tar peut être compressé.)

Je pars du principe que vous pouvez installer le système source vous−même, sans aide de ma part. À partir de maintenant, je considère que c'est fait.

Les premiers pas de ce projet consistent à faire démarrer le noyau, et à le laisser « paniquer » (panic) car il ne trouve pas le programme init. Cela signifie que nous allons devoir installer un noyau, et installer Lilo. Pour que Lilo s'installe facilement, nous aurons besoin des fichiers spéciaux du répertoire `/dev` du système cible. Lilo en a besoin pour effectuer les accès bas niveau au disque, nécessaire pour écrire le secteur d'amorçage. **MAKEDEV** est le script qui crée ces fichiers spéciaux (vous pourriez bien sûr les recopier depuis le système source, mais ce serait tricher !). Mais d'abord, il nous faut un système de fichiers dans lequel les mettre.

2. Le système de fichier

Notre nouveau système a besoin d'un système de fichiers pour vivre. Donc, il nous faut tout d'abord créer ce système de fichiers en utilisant **mke2fs**. Ensuite il faut le monter quelque part. Je vous suggère `/mnt/cible`. Dans ce qui va suivre, je considère que votre système cible se trouve à cet endroit. Vous pouvez gagner un peu de temps en ajoutant une entrée dans `/etc/fstab` de façon à ce que le montage de votre système de destination se fasse automatique lors du démarrage de votre système source.

Lorsque nous démarrerons le système cible, ce qui se trouve dans `/mnt/cible` se trouvera alors dans `/` (la racine).

Nous avons besoin d'une structure de sous−répertoires sur la cible. Jetez un Sil au standard de hiérarchie des fichiers (*File Hierarchy Standard − FHS*, voir la section liens) pour trouver vous−même ce qu'elle devrait être, ou faites simplement un **cd** vers l'endroit où la cible est montée et tapez aveuglément :

```
mkdir bin boot dev etc home lib mnt root sbin tmp usr var
cd var; mkdir lock log run spool
cd ../usr; mkdir bin include lib local sbin share src
cd share/; mkdir man; cd man
mkdir man1 man2 man3 man4 man5 man6 man7 man8 man9
```

Comme le standard de hiérarchie des fichiers et la plupart des paquets se contredisent en ce qui concerne l'endroit où les pages de manuel doivent se trouver, nous avons besoin d'un lien symbolique :

```
cd ..; ln −s share/man man
```

3. MAKEDEV

Nous mettrons le code source dans le répertoire `/usr/src` cible. Ainsi si votre système de fichiers cible est monté sur `/mnt/cible`, et que vos archives tar sont dans `/root`, il faudra faire :

```
cd /mnt/cible/usr/src
tar -xzvf /root/MAKEDEV-2.5.tar.gz
```

Ne vous comportez pas en amateur fini en copiant les archives à l'endroit où vous allez les décompresser ;-)

En principe, lorsque vous installez un logiciel, vous l'installez sur le système en fonctionnement. En l'occurrence, ce n'est pas notre intention, nous souhaitons l'installer comme si /mnt/cible était la racine du système de fichiers. Les différents paquets ont différentes manières de vous laisser faire cela. Pour MAKEDEV, vous devez faire :

```
ROOT=/mnt/cible make install
```

Vous devez rechercher ces options dans les fichiers README et INSTALL ou faire un **./configure --help**.

Explorez le `Makefile` de MAKEDEV pour voir l'usage qu'il fait de la variable *ROOT*, que nous avons définie dans cette commande. Ensuite jetez un Sil à la page de manuel en faisant un **man ./MAKEDEV.man** pour voir comment il fonctionne. Vous découvrirez que la méthode que nous devons utiliser pour créer ces fichiers spéciaux consiste à faire un **cd /mnt/cible/dev** puis un **./MAKEDEV generic**. Faites un **ls** pour découvrir tous les merveilleux fichiers spéciaux qu'il a créé pour vous !

4. Le noyau

Ensuite, nous devons fabriquer un noyau. Je considère que vous l'avez déjà fait, aussi serai-je bref. Il est plus facile d'installer Lilo si le noyau censé être monté est déjà là. Retournez dans le répertoire `usr/src` de la cible, et extrayez-y les sources du noyau Linux. Entrez dans l'arborescence des sources (**cd linux**) et configurez le noyau, en utilisant votre méthode préférée, par exemple **make menuconfig**. Vous vous faciliterez grandement la vie si vous configurez un noyau sans module. Si vous choisissez d'avoir des modules, vous devrez éditer le fichier `Makefile`, trouver *INSTALL_MOD_PATH*, et lui affecter la valeur `/mnt/cible`.

Vous pouvez maintenant taper **make dep**, **make bzImage**, et si vous avez configuré des modules : **make modules**, **make modules_install**. Copiez le noyau `arch/i386/boot/bzImage` et le plan système `System.map` vers le répertoire d'amorçage de la cible `/mnt/cible/boot`, et nous seront prêts à installer Lilo.

5. Lilo

Lilo est livré avec un script très bien conçu nommé **QuickInst**. Décompressez les sources de Lilo dans le répertoire des sources du système cible, lancez ce script par la commande **ROOT=/mnt/cible ./QuickInst**. Il vous posera plusieurs questions concernant la manière dont vous souhaitez que Lilo soit installé.

Puisque nous avons affecté à la variable *ROOT* la partition cible, les noms des fichiers que nous lui indiquons doivent être relatifs à cette partition. Donc, à la question du nom du noyau à lancer par défaut, répondez `/boot/bzImage`, et *non* `/mnt/cible/boot/bzImage`.

J'ai trouvé une erreur mineure dans le script, qui lui fait dire :

```
./QuickInst: /boot/bzImage: no such file
```

Mais si vous vous contentez de l'ignorer, cela passe quand même.

Comment doit−on s'y prendre pour expliquer à **QuickInst** où installer le secteur d'amorçage ? Quand nous redémarrerons, nous voulons avoir le choix de démarrer le système source ou le système cible, ou encore n'importe quel autre système présent sur la machine. Et nous souhaitons que l'instance de Lilo que nous mettons en place maintenant lance le noyau de notre nouveau système. Comment réaliser ces deux choses ? Écartons−nous un moment du sujet et étudions la façon dont Lilo démarre DOS sur un système Linux en double−amorçage. Le fichier `lilo.conf` d'un tel système doit sûrement ressembler à ça.

```
prompt
timeout = 50
default = linux

image = /boot/bzImage
        label  = linux
        root   = /dev/hda1
        read−only

other = /dev/hda2
        label = dos
```

Si la machine est configurée de cette façon, alors le bloc de démarrage (*Master Boot Record − MBR*) est lu et chargé par le Bios, et lance le chargeur d'amorçage de Lilo (*bootloader*), qui affiche une invite de commande. Si vous tapez *dos* à cette invite, Lilo chargera le secteur d'amorçage de hda2, qui lancera DOS.

Ce que nous allons faire est exactement la même chose, à une différence près : le secteur d'amorçage de hda2 sera un autre secteur d'amorçage Lilo − celui−là même que **QuickInst** va installer. Donc le Lilo de la distribution Linux chargera le Lilo que nous avons construit, qui chargera le noyau que nous avons bâti. Vous verrez alors deux invites Lilo au redémarrage.

Pour raccourcir une longue histoire, lorsque **QuickInst** vous demande où placer le secteur de boot, indiquez−lui l'endroit où se trouve votre système de fichiers cible, par exemple `/dev/hda2`.

Maintenant modifiez le fichier `lilo.conf` de votre système source, de façon à ce qu'il comprenne une ligne ressemblant à :

```
other = /dev/hda2
        label = cible
```

Lancez **lilo**, et nous devrions être capables de faire notre premier démarrage sur le système cible.

6. Glibc

L'étape suivante consiste à installer init, mais comme la plupart des programmes qui tournent sous Linux, init utilise des fonctions issues de la bibliothèque standard C GNU, glibc. Aussi l'installerons−nous en premier.

Glibc est un paquet très gros et très complexe. Il faut 90 heures pour le construire sur mon vieux 386sx / 16 avec 8 Mo de mémoire. Mais cela ne prend que 33 minutes sur mon Celeron 433 avec 64 Mo de mémoire. Je pense que la quantité de mémoire est le principal critère dans notre cas. Si vous n'avez que 8 Mo de mémoire (ou − j'en tremble − encore moins !), préparez vous à une très longue compilation.

La documentation d'installation de glibc recommande une construction dans un répertoire distinct. Cela vous permettra de recommencer facilement, en supprimant simplement ce répertoire. Cela vous permet aussi d'économiser 265 Mo d'espace disque.

Comme d'habitude, décompressez l'archive `glibc−2.1.3.tar.gz` (ou n'importe quelle autre version) dans `/mnt/cible/usr/src`. À présent, nous devons décompresser les extensions dans le répertoire de glibc. Donc, faites un **cd glibc−2.1.3**, puis décompressez à cet endroit les archives `glibc−crypt−2.1.3.tar.gz` et `glibc−linuxthreads−2.1.3.tar.gz`.

Maintenant, nous pouvons créer le répertoire de construction, configurer, construire et installer glibc. Voici les commandes que j'ai utilisées, mais relisez vous−même la documentation et assurez−vous de faire ce qui est le plus approprié dans votre environnement. Toutefois, avant de faire tout cela, vous voudrez sans doute connaître l'espace disque qu'il vous reste en faisant un **df**. Vous pourrez en faire un autre après avoir construit et installé glibc pour en déduire son volume.

```
cd ..
mkdir glibc−build
../glibc−2.1.3/configure −−enable−add−ons −−prefix=/usr
make
make install_root=/mnt/cible install
```

Remarquez que nous avons ici encore une autre façon de dire au paquet l'endroit où s'installer.

7. SysVinit

Fabriquer et installer les binaires de SysVinit est assez simple. Il y a juste une petit manipulation à faire dans le fichier `Makefile`, situé dans le sous−répertoire `src/`. Dans les 4 dernières lignes, vous devez placer *$(ROOT)* juste devant */dev/initctl*. Par exemple :

```
@ if [ ! −p /dev/initctl ]; then \
```

devient :

```
@ if [ ! −p $(ROOT)/dev/initctl ]; then \
```

Le fichier spécial `initctl` est un moyen de communication avec init. Par exemple, la page de manuel d'init indique que ce fichier doit être utilisé de préférence au signal *SIGPWR* pour demander à init d'arrêter le système lorsque l'alimentation électrique à basculé sur batterie suite à une panne de courant. Cette manipulation permet de s'assurer que ce fichier se trouvera dans le système cible, et non dans le système source.

Une fois que c'est fait, placez−vous dans le sous−répertoire `src`, et entrez :

```
make
ROOT=/mnt/cible make install
```

Il existe aussi beaucoup de scripts associés à init. Il y a des scripts d'exemple fournis dans le paquet SysVinit, qui fonctionnent bien. Mais vous devez les installer manuellement. Ils sont rangés hiérarchiquement sous `debian/etc` dans l'arborescence du code source. Vous pouvez recopier toute cette hiérarchie dans le répertoire `etc` du système cible, avec une commande du style **cd ../debian/etc; cp −r * /mnt/cible/etc**. Évidement, vous voudrez les examiner avant de les recopier.

Tout est désormais en place pour permettre au noyau cible de lancer init au redémarrage. Le problème, cette fois, viendra des scripts qui ne pourront être exécutés car bash ne sera pas là pour les interpréter. init tentera également de lancer des getty, qui sont eux aussi inexistants. Redémarrez le système, et assurez−vous que tout le reste fonctionne correctement.

8. Ncurses

L'étape suivante consiste à mettre Bash en place, mais bash a besoin de ncurses, aussi devons−nous installer celui−ci en premier. Ncurses remplace termcap dans la manière de gérer les écrans texte, mais apporte également une compatibilité descendante en prenant en charge les appels termcap. Dans l'objectif d'avoir un système moderne, simple et propre, je pense que le mieux est de désactiver l'ancienne méthode termcap. Vous pourriez par la suite rencontrer des problèmes avec des applications utilisant termcap, mais au moins vous connaîtrez les éléments qui l'utilisent. Si vous en avez besoin, vous pourrez recompiler ncurses avec prise en charge de termcap.

Les commandes que j'ai utilisées sont :

```
./configure --prefix=/usr --with-install-prefix=/mnt/cible --with-shared --disable-termcap
make
make install
```

---

9. Bash

Il m'a fallu beaucoup de lecture, de réflexion, de tests, et d'erreurs pour que Bash s'installe là où je pensais qu'il devait aller. Les options de configuration que j'ai utilisées sont :

```
./configure --prefix=/mnt/cible/usr/local --exec-prefix=/mnt/cible \
            --with-curses
```

Une fois que vous aurez construit et installé Bash, vous devrez créer un lien symbolique comme ceci : **cd /mnt/cible/bin; ln −s bash sh**. Cela est dû au fait que les scripts débutent généralement par une ligne comme celle−ci :

```
#!/bin/sh
```

Si vous n'avez ce lien symbolique, les scripts ne fonctionneront pas, car ils chercheront */bin/sh* et non */bin/bash*.

Arrivé à ce point, vous pouvez redémarrer si vous le souhaitez. Lors du redémarrage, vous devriez remarquer que, maintenant, les scripts s'exécutent. Cependant, vous ne pourrez pas vous connecter, car il n'y pas encore de programmes getty ou login.

---

10. Util−linux (getty et login)

Le paquet util−linux contient agetty et login. Nous avons besoin des deux pour pouvoir nous connecter et obtenir l'invite de commande de bash. Après l'avoir installé, faites un lien symbolique depuis `agetty` vers `getty` dans le répertoire `/sbin` du système cible. getty est un des programmes censés se trouver sur tous les systèmes de type Unix, donc faire un lien est une meilleure idée que de modifier `inittab` pour qu'il lance agetty.

Il me reste un problème avec la compilation d'util−linux. Le paquet contient également le programme *more*, et, pour ce programme, je n'ai pas été capable de persuader *make* de réaliser l'édition de liens avec le ncurses 5 du système cible au lieu du ncurses 4 du système source.

Vous aurez aussi besoin d'un fichier `/etc/passwd` sur le système cible. C'est l'endroit où le programme login ira vérifier si vous avez le droit de vous connecter. Comme il ne s'agit que d'un système jouet, vous pouvez vous permettre à ce niveau des choses scandaleuses, comme de ne définir que l'utilisateur root, sans mot de passe ! Mettez le simplement dans le fichier `/etc/passwd` du système cible :

```
root::0:0:root:/root:/bin/bash
```

Les champs sont séparés par des deux−points, et correspondent, de gauche à droite, à l'identifiant de l'utilisateur, à son mot de passe (chiffré), à son numéro d'utilisateur, à son numéro de groupe, à son nom complet, à son répertoire personnel, et à son interpréteur de commandes par défaut.

11. Sh−utils

Le dernier paquet dont nous ayons besoin est le sh−utils GNU. Le seul programme nécessaire à ce niveau est *stty*, qui est utilisé dans `/etc/init.d/rc`, lui−même utilisé pour changer de niveau d'exécution et entrer dans le niveau initial. En fait, je possède et ai utilisé un paquet qui ne contient que *stty* mais je ne peux me souvenir d'où il vient. Il vaut mieux utiliser le paquet GNU, car il contient d'autres choses dont vous aurez besoin si vous voulez les ajouter au système pour le rendre vraiment utilisable.

Bien, ça y est. Vous devriez maintenant avoir un système qui va démarrer et vous donner l'invite de connexion. Saisissez−y *root*, et vous devriez accéder à l'interpréteur de commandes. Vous ne pourrez pas faire grand chose avec, il n'y a même pas la commande **ls** pour voir votre travail. Tapez deux fois sur la touche **Tab** pour voir les commandes disponibles. C'est la chose la plus intéressante que j'ai trouvée à faire avec.

12. Rendre le système plus utilisable

Il semblerait que nous ayons là un système plutôt inutilisable. Mais en réalité, nous ne sommes pas très loin de pouvoir commencer à travailler avec. L'une des premières choses à faire est de rendre le système de fichiers racine accessible en lecture et en écriture. Il y a un script issu du paquet SysVinit, /etc/init.d/mountall.sh qui s'occupe de cela, et effectue un **mount −a** pour monter automatiquement tout ce qui est spécifié dans le fichier `/etc/fstab`. Mettez un lien symbolique du genre `S05mountall` vers lui dans le répertoire `etc/rc2.d` du système cible.

Il se peut que ce script utilise des commandes que vous n'avez pas encore installées. Si c'est le cas, trouvez le paquet qui contient ces commandes et installez−le. Voyez la section astuces diverses pour avoir des indications sur la marche à suivre pour trouver ces paquets.

Regardez les autres scripts de `/etc/init.d`. La plupart d'entre−eux doit être incluse dans tout système sérieux. Ajoutez−les un à un, et assurez−vous que tout se lance en douceur avant d'en ajouter d'autres.

Lisez le standard de hiérarchie des fichiers (voir section liens). Il contient une liste des commandes qui devraient être dans `/bin` et `/sbin`. Assurez−vous que toutes ces commandes sont installées sur votre système. Mieux encore, trouvez la documentation Posix qui spécifie tout cela.

À partir de maintenant, il n'est plus question que d'ajouter de plus en plus de paquets, jusqu'à ce que tout ce que vous souhaitez avoir se trouve sur votre système. Installez les outils de construction comme *make* et *gcc* le plus tôt possible. Une fois que cela sera fait, vous pourrez faire construire le système cible par lui−même, ce qui est bien moins complexe.

13. Informations complémentaires

13.1. Astuces diverses

Si vous avez une commande appelée **machin** sur un système Linux avec RPM, et souhaitez avoir des indications sur l'endroit où trouver les sources, vous pouvez utiliser la commande :

```
rpm −qif `which machin`
```

Et si vous avez un CD de sources Red Hat, vous pouvez installer le code source avec

```
rpm -i /mnt/cdrom/SRPMS/ce.qu.il.vient.de.dire-1.2.srpm
```

Ceci mettra l'archive, avec les patches Redhats éventuels dans `/usr/src/redhat/SOURCES`.

13.2. Liens

- Il existe un guide pratique sur la manière de construire des logiciels à partir de leurs sources, le Software Building HOWTO.
- Il existe aussi un guide pratique sur la manière de construire un système Linux à partir de zéro. Il met l'accent sur la construction d'un système réellement utilisable, plutôt que d'être un simple exercice : Comment faire un système Linux à partir de zéro (*Linux from scratch − LFS*).
- Le standard de hiérarchie du système de fichier Unix (FHS). Ce standard décrit quels éléments doivent aller à quels endroits dans un système de fichier Unix, et pourquoi. Il indique également le contenu minimum requis des répertoires `/bin`, `/sbin`, et cætera. C'est une bonne référence si votre but est de réaliser un système minimal, mais fonctionnel. Il en existe une version française.

14. Section administrative

14.1. Copyright

14.2. Page principale

Les mises à jour de ce document évoluent sur From Powerup To Bash Prompt.

14.3. Réactions

J'aimerais recevoir vos commentaires, critiques et suggestions. Veuillez s'il vous plaît me les envoyer en anglais à Greg O'Keefe `<gcokeefe@postoffice.utas.edu.au>`

14.4. Références et remerciements

Les noms de produits cités sont des marques déposées par leurs propriétaires respectifs, et considérés par cette note comme reconnus comme tels.

Il y a quelques personnes que je voudrais remercier, pour m'avoir aidé à réaliser tout ceci.

Michael Emery
: Pour m'avoir rappelé Unios.

Tim Little
: Pour de bonnes indications concernant `/etc/passwd`

sPaKr dans #linux sur efnet
: Qui a soupçonné l'utilisation de `/etc/services` par syslog, et m'a fait connaître la phrase « rolling your own » (« fabriquez votre propre système ») pour décrire la construction d'un système à partir des sources.

Alex Aitkin
: Pour avoir porté Vico et son « verum ipsum factum » (La compréhension découle de l'expérience) à

mon attention.

Dennis Scott
: Pour avoir corrigé mon arithmétique hexadécimale.

jdd
: Pour avoir mis en évidence quelques erreurs typographiques.

14.5. Historique des changements

0.8 −> 0.9
: Ajout de la manipulation du fichier `makefile` de sysvinit. Cette information est due à Gerard Beekmans, connu pour le « Linux From Scratch ».

0.8
: Version initiale. Séparation de ce HOWTO du « From PowerUp to Bash Prompt ».

14.6. Améliorations prévues

- Conversion au format DocBook.

14.7. Adaptation française

14.7.1. Traduction

La traduction française de ce document a été réalisée par Dominique van den Broeck `<dvandenbroeck@free.fr>`, décembre 2000 (v0.9).

14.7.2. Relecture

La relecture de ce document a été réalisée par Jean−Philippe Guérard `<jean-philippe.guerard@laposte.net>`. Les version précédentes ont été relues par Guillaume Allègre et Anthony Boureux.