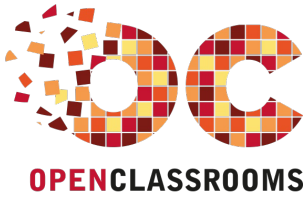
Création de paquets avec ArchLinux

Par Dadouchi A.
et Perkele



www.openclassrooms.com

Dernière mise à jour le 29/05/2010

Sommaire

Création de paquets avec ArchLinux

Par  Perkele et Dadouchi A.

Mise à jour : 29/05/2010
Difficulté : Facile Durée d'étude : 2 heures

Vous avez envie de créer le paquet ArchLinux pour votre super jeu en 3D ou encore pour votre éditeur de texte révolutionnaire mais vous ne savez pas par où commencer ?

Pour suivre ce tutoriel, il vous faut évidemment une distribution ArchLinux fonctionnelle. Si vous voulez installer ce système, vous pouvez vous référer à la page « Installation de base » du wiki français. Il faut aussi une base en bash (variables, tableaux, conditions et fonctions sont utilisés dans ce tutoriel) ainsi qu'une connaissance des commandes courantes.

Si c'est bon, suivez le guide...
Sommaire du tutoriel :

PKGBUILD ou la base du paquet

Tout d'abord, sachez que PKGBUILD est un simple fichier texte qui va expliquer comment se déroule la création du paquet. Nous allons donc voir sa syntaxe et faire une approche de paquet.

Pour comprendre la syntaxe de ce fichier, je vous propose un petit PKGBUILD classique pour un programme fictif nommé Palne :

Code : Bash

```
# Maintainer: Adrien Perkele <perkele@domain.tld>
# Contributor: Adrien Perkele <perkele@domain.tld>
pkgname=palne
pkgver=0.99_10
pkgrel=1
pkgdesc="Palne, le jeu super cool de Perkele"
arch=(any)
url="http://www.palne.net"
license=('BSD')
groups=
provides=
depends=('scengine>=1.2')
optdepends=('unelib: Pour rajouter une fonctionnalité au programme
!')
makedepends=
conflicts=('kipalne')
replaces=('palneuh')
backup=('etc/palne/palne.conf')
install=
source=(http://www.palne.net/download/$pkgname-$pkgver.tar.gz)
md5sums=('40972633aea6a8d38ff26dade3eea51e')
```

```
md5sums=('4097265...')

build() {
  cd $srcdir/$pkgname-$pkgver
  ./configure --prefix=/usr
  make || return 1
  make prefix=$pkgdir/usr install
}
```

Évidemment, toutes ces variables ne vous disent peut-être pas grand-chose, je vous propose donc ce petit tableau :

Nom	Description
# Maintainer	La personne qui se charge officiellement de la maintenance du paquet (format Nom <mail@domain.tld>)
# Contributor	La personne qui est à l'origine du paquet (même format)
pkgname	Nom du paquet
pkgver	Version du programme (chiffres et lettres uniquement, pour séparer on utilise des underscores (_)).
pkgrel	Valeur spécifique à ArchLinux, elle correspond à la version du paquet (pour chaque version d'un PKGBUILD d'un même programme de la même version, on met une valeur différente, en commençant par 1). Elle gère entre autres les paquets périmés.
pkgdesc	Courte description du contenu du paquet qui sera affichée dans les détails.
arch	L'architecture où la compilation du programme est compatible (**i686** ou **x86_64** ou **any** pour les deux)
url	Site du programme
license	Licence du programme (lire la partie « Le problème des licences » pour les valeurs)
groups	Si le paquet appartient à un groupe de paquets, on le précise (exemple kdebase appartient au groupe kde)
provides	Si notre paquet fournit un autre logiciel, on le précise
depends	Les dépendances du programme (lire la partie « Les dépendances » pour savoir comment ça marche)
optdepends	Paquet optionnel qui rajoute en général des fonctionnalités au programme (exemple, pacman-color rajoute de la couleur dans pacman).
makedepends	Ce sont aussi des dépendances, mais celles que la compilation nécessite (pareil, voir la partie « Les dépendances »)
conflicts	Les paquets qui empêchent le fonctionnement de notre programme
replaces	Précise les paquets que notre programme remplace (exemple, ancienne version du programme qui porte un autre nom)
backup	Les fichiers qui seront sauvegardés à la désinstallation
install	Précise un fichier d'installation (voir la partie « Les fichiers d'installation »)
source	Le lien de l'archive contenant les sources
md5sums	Permet de vérifier l'intégralité de l'archive en fournissant son empreinte MD5
build()	Fonction qui nous permettra de préparer le paquet (compilation ainsi que placement dans le paquet final)

Les dépendances

Pour la variable depends, on prend un tableau avec la liste des dépendances sous la forme « `'[nom du paquet archlinux]'` ». On peut aussi préciser une version avec un comparatif (qui je rappelle sont =, >, <, <= ou encore >=) suivi d'un numéro de version (par exemple « `'python>=2.6'` »).

Le problème des licences

La variable **licence** est assez particulière, en effet il faut respecter une syntaxe et ne pas balancer le nom d'une licence comme ça.
Il y a plusieurs cas selon les licences, je vais essayer de vous les détailler un maximum.
Tout d'abord, vérifiez si un dossier du nom de la licence existe dans */usr/share/licenses/common/* :

- Si votre licence (elle est « common ») s'y trouve, il suffit d'écrire le nom du répertoire dans la variable **license** (exemple: `license=('GPL')`);
- Si votre licence est MIT, BSD, zlib/libpng ou encore Python, vous devrez créer un fichier nommé généralement COPYING (le nom n'est pas important) contenant le texte de la licence dans */usr/share/licenses/**$pkgname***. Vous verrez comment faire cela dans la partie sur la fonction build(). Pour l'utiliser, il suffira de faire comme pour les licences common, un exemple se trouve dans mon fichier PKGBUILD ;
- Dans le cas d'une licence personnelle, vous allez aussi créer votre fichier COPYING. Quant à l'utilisation dans le PKGBUILD, vous allez utiliser la syntaxe « `license=('custom:Nom de la licence')` ;
- Enfin, dernier cas, si votre paquet est couvert par plusieurs licences, vous aurez remarqué que **license** est un tableau : chaque valeur est une licence (exemple : `license=('GPL' 'custom:crave' 'BSD')`).

La variable *md5sums*

Un petit mot sur le md5sums : pour le calculer, il suffit d'utiliser la commande `md5sum` ainsi :

Code : Console

```
md5sum monprogramme-1.1_1.tar.gz | cut -d ' ' -f 1
```

C'est le résultat de cette commande que vous allez mettre dans la variable md5sums.

La fonction build()

Maintenant, nous rentrons dans le vif du sujet. Jusqu'ici, nous avons fait de la théorie sur la création et nous savons récupérer les sources, vérifier qu'elles sont complètes, mais nous ne savons toujours pas créer le paquet !
C'est la fonction build() qui va se charger de cette tâche.

En général, l'ordre des commandes dans cette fonction est celui-ci :

Code : Autre

```
Fonction build
    On se place dans le répertoire contenant les sources téléchargées
    On compile les sources
    On place dans le répertoire qui va être le paquet tout ce qu'il nous faudra
Fin de la fonction
```

Il faut savoir qu'un paquet est une simple archive qui contient l'architecture du système. Le gestionnaire s'occupera donc ainsi (en partie) de décompresser l'archive à la racine.
Pendant la création de notre fonction, nous allons traiter un dossier nommé pkg, qui sera compressé à l'exécution du PKGBUILD.

Comme exemple, nous allons maintenant imaginer que notre archive tar.gz téléchargée contient ceci :

Code : Autre

```
Archive tar.gz
---- bin/
---- doc/
-------- palne.6.gz
-------- index.html
---- src/
-------- main.c
-------- fonctions.c
-------- fonctions.h
---- data/
-------- COPYING
```

```
-------- images/
------------ icone.png
------------ splash.png
------------ gagne.png
------------ perdu.png
-------- conf/
------------ palne.cfg
---- Makefile
```

Nous allons maintenant créer la fonction build() qui va :

- compiler les sources ;
- mettre le man dans le répertoire qui correspond ;
- mettre la doc dans le répertoire qui correspond ;
- mettre l'exécutable dans /usr/bin ;
- mettre la configuration dans /etc ;
- placer la licence dans /usr/share/licenses/ ;
- et enfin mettre les images dans /usr/share/.

Tout d'abord, on se place dans le répertoire des sources :

Code : Bash

```
build()
{
    cd $srcdir/$pkgname
```

La variable srcdir contient le répertoire où l'archive a été décompressée.
Attention néanmoins, les fichiers décompressés sont dans un répertoire du même nom que le paquet.

Nous allons donc maintenant compiler les sources de notre programme en exécutant simplement le Makefile :

Code : Bash

```
make || return 1
```

Comme vous le constatez, ma commande de compilation est suivie d'un `|| return 1` : cela permet de dire que si la commande a échoué, on arrête la création du paquet.
Il faudra placer ceci après chaque commande importante pour le paquet.

Passons maintenant à la création de l'architecture du paquet :

Code : Bash

```
mkdir -p $pkgdir/usr/bin/
mkdir -p $pkgdir/usr/share/palne/
mkdir -p $pkgdir/usr/share/doc/palne/
mkdir -p $pkgdir/usr/man/man6/
mkdir -p $pkgdir/etc/palne/
mkdir -p $pkgdir/usr/share/licenses/palne/
```

À noter que la variable **pkgdir** correspond au dossier **pkg** dont je vous avais parlé plus tôt.
On va maintenant pouvoir passer à la copie des fichiers :

Code : Bash

```
cp bin/palne $pkgdir/usr/bin/
chmod +x $pkgdir/usr/bin
cp doc/palne.6.gz $pkgdir/usr/man/man6/
```

```
    cp doc/index.html $pkgdir/usr/share/doc/palne/
    cp -r data/images/ $pkgdir/usr/share/palne/images/
    cp data/conf/palne.cfg $pkgdir/etc/palne/
    cp data/COPYING $pkgdir/usr/share/licenses/palne/
```

À noter que le chmod est obligatoire si l'on veut pouvoir exécuter le programme.

Et bonne nouvelle, nous avons fini notre fonction build() !
Néanmoins, il ne faut pas oublier de la refermer :

Code : Bash

```
}
```

Voici donc un petit récapitulatif de la fonction qui correspond aux besoins de mon programme Palne :

Code : Bash

```
build()
{
    cd $srcdir/$pkgname
    make || return 1

    mkdir -p $pkgdir/usr/bin/
    mkdir -p $pkgdir/usr/share/palne/
    mkdir -p $pkgdir/usr/share/doc/palne/
    mkdir -p $pkgdir/usr/man/man6/
    mkdir -p $pkgdir/etc/palne/
    mkdir -p $pkgdir/usr/share/licenses/palne/

    cp bin/palne $pkgdir/usr/bin/
    chmod +x $pkgdir/usr/bin
    cp doc/palne.6.gz $pkgdir/usr/man/man6/
    cp doc/index.html $pkgdir/usr/share/doc/palne/
    cp -r data/images/ $pkgdir/usr/share/palne/images/
    cp data/conf/palne.cfg $pkgdir/etc/palne/
    cp data/COPYING $pkgdir/usr/share/licenses/palne/
}
```

Créer le paquet et le redistribuer

Création

Bon, nous savons créer un PKGBUILD basique, mais nous ne savons toujours pas comment obtenir le paquet !
En fait, vous allez rire mais ~~le PKGBUILD, c'était une blague de mon invention et il ne sert à rien~~ la création d'un paquet se limite à une commande.
Tout d'abord je vous recommande de mettre le PKGBUILD dans un répertoire créé à cet effet.

Maintenant, l'heure de vérité, lancez cette commande :

Code : Console

```
makepkg
```

Si tout se passe bien, il vous posera quelques questions, et vous retournerez dans votre prompt.
Mais mais, que vois-je ? Notre répertoire a été rempli !

Code : Console

```
$ ls
```

```
src/ pkg/ PKGBUILD nom-version.tar.gz
```

Eh bien oui, ce fameux tar.gz est votre paquet. Vous ne me croyez pas ? Alors lancez l'installation avec `yaourt -U nom-version.tar.gz` puis testez votre programme avec `palne`.

Redistribution

Vous pouvez vous contenter de donner votre tar.gz au public via votre site internet ou alors la mettre dans les dépôts AUR d'ArchLinux.
C'est cette méthode que je vais vous présenter.

Tout d'abord, il faut vous inscrire sur le site officiel de AUR : http://aur.archlinux.org/account.php
Une fois inscrit, connectez-vous et choisissez « submit » (ou « soumettre » en français) dans le menu en haut.
Vous aurez deux champs : Package Category et Upload package file.
Pour la liste déroulante, vous mettrez la catégorie de votre paquet (dans le cas de mon exemple, ce serait un jeu).
Pour le fichier à sélectionner, nous allons le créer tout de suite en exécutant la commande `makepkg --source` dans le répertoire du PKGBUILD.
Validez. Si tout s'est bien passé, en exécutant la commande `yaourt -Sy monpaquet`, votre paquet s'installera !

Les fichiers d'installation

Quand votre programme nécessite une configuration spécifique, vous devriez l'expliquer à l'installation.
C'est grâce au fameux fichier d'installation précisé par la variable install que vous pourrez le faire.

Il est composé de trois simples fonctions, chacune exécutée à un moment précis de l'utilisation du paquet.

La première fonction, post_install(), est exécutée après l'installation du paquet (vous pourrez préciser où trouver la configuration de base, par exemple. Comme argument, cette fonction prend le numéro de version.

La seconde fonction, post_upgrade(), est exécutée après la mise à jour du paquet, on l'utilise souvent pour préciser que la configuration est incompatible, si c'est le cas. En arguments, cette fonction prend le numéro de la nouvelle version puis celui de l'ancienne.

La troisième fonction, pre_remove(), est exécutée avant la suppression. Je n'ai malheureusement pas d'exemple d'utilisation, mais si quelqu'un en trouve un, je serai heureux de l'ajouter dans ce tutoriel. Cette fonction prend le numéro de version.

Voici un exemple de fichier tiré du paquet du Window Manager Awesome :

Code : Bash

```
post_install() {
cat << _EOF

==> awesome installation notes:
---------------------------------------
During some updates of awesome, the config file syntax changes
heavily.
This means that your configuration file (~/.config/awesome/rc.lua)
for
any other than this particular version of awesome may or may not
work.

_EOF

}

post_upgrade() {
        post_install
}
```

Des PKGBUILD plus compliqués

Ceux que nous avons vu jusqu'à maintenant sont assez basiques, nous allons donc en voir quelques-uns plus complexes, par exemple avec gestion des patchs ou d'un système de versionnage.
C'est parti !

Les gestionnaires de versions

Comme je n'ai pas une imagination infinie, je vais simplement réutiliser mon ancien PKGBUILD et le transformer pour utiliser le gestionnaire de versionnage **git**.

Pour ceux qui l'auraient oublié, en voici le code :

Secret (cliquez pour afficher)

Code : Bash

```
# Maintainer: Adrien Perkele <perkele@domain.tld>
# Contributor: Adrien Perkele <perkele@domain.tld>
pkgname=palne
pkgver=1
pkgrel=1
pkgdesc="Palne, le jeu super cool de Perkele"
arch=(any)
url="http://www.palne.net"
license=('BSD')
groups=
provides=
depends=('scengine>=1.2')
optdepends=('unelib: Pour rajouter une fonctionnalité au programme
!')
makedepends=
conflicts=('kipalne')
replaces=('palneuh')
backup=('etc/palne/palne.conf')
install=
source=(http://www.palne.net/download/$pkgname-$pkgver.tar.gz)
md5sums=('40972633cea6c8d38ff26dade3eec51e')

build()
{
    cd $srcdir/$pkgname
    make || return 1

    mkdir -p $pkgdir/usr/bin/
    mkdir -p $pkgdir/usr/share/palne/
    mkdir -p $pkgdir/usr/share/doc/palne/
    mkdir -p $pkgdir/usr/man/man6/
    mkdir -p $pkgdir/etc/palne/

    cp bin/palne $pkgdir/usr/bin/
    chmod +x $pkgdir/usr/bin
    cp doc/palne.6.gz $pkgdir/usr/man/man6/
    cp doc/index.html $pkgdir/usr/share/doc/palne/
    cp -r data/images/ $pkgdir/usr/share/palne/images/
    cp data/conf/palne.cfg $pkgdir/etc/palne/
}
```

On va donc commencer par renommer le paquet pour préciser qu'il utilise les versions de développement, par exemple palne-git.

Maintenant, il va falloir préciser que git est nécessaire à la compilation :

Code : Bash

```
makedepends=('git')
```

Étant donné que nous ne téléchargeons plus les sources par une archive sur le net, il va falloir supprimer le contenu de **source** et de **md5sums** :

Code : Bash

```
source=()
md5sums=()
```

Maintenant, pour que le code soit plus clair, nous allons créer deux variables personnalisées. Néanmoins, pour éviter d'éventuels conflits, nous allons mettre un underscore (_) avant le nom de la variable. Nous allons donc créer nos deux variables, une pour la racine de notre *repository* et une pour le nom du répertoire à créer, par exemple :

Code : Bash

```
_gitroot="git://git.palne.net/palne"
_gitname="palne"
```

Pour la suite, nous allons modifier la fonction build().
Remplacez la première ligne `cd $srcdir/$pkgname` par un simple `cd $srcdir` .

Ensuite nous allons vérifier si le *repository* a déjà été récupéré ou s'il faut le faire :

Code : Bash

```
if [[ -d $_gitname ]] ; then
    # Le repository a déjà été récupéré, on le met a jour
else
    # on doit le récupérer
fi
```

Pour mettre à jour notre *repository*, nous allons faire comme cela (et laisser un petit message) :

Code : Bash

```
git-pull origin || return 1
msg "Les fichiers locaux ont été mis à jour."
```

À noter que je n'ai pas oublié de finir par ligne par un `|| return 1` pour préciser que si la fonction échoue, la suite ne fonctionnera pas.

Maintenant, pour récupérer le *repository*, nous allons faire ainsi :

Code : Bash

```
git clone $_gitroot || return 1
msg "Les fichiers ont été téléchargés."
```

Nous avons fini, c'était si compliqué ?

Pour ceux qui n'auraient pas bien compris où placer le code, voici un récapitulatif du PKGBUILD :

Code : Bash

```
# Maintainer: Adrien Perkele <perkele@domain.tld>
# Contributor: Adrien Perkele <perkele@domain.tld>
pkgname=palne-git
pkgver=1
pkgrel=1
pkgdesc="Palne, le jeu super cool de Perkele"
arch=(any)
url="http://www.palne.net"
license=('BSD')
groups=
provides=
depends=('scengine>=1.2')
optdepends=('unelib: Pour rajouter une fonctionnalité au programme
!')
makedepends=('git')
conflicts=('kipalne')
replaces=('palneuh')
backup=('etc/palne/palne.conf')
install=
source=()
md5sums=()
_gitroot="git://git.palne.net/palne"
_gitname="palne"

build()
{
    msg "Récupération des sources sur le serveur git"
    if [[ -d $_gitname ]] ; then
git-pull origin || return 1
        msg "Les fichiers locaux ont été mis à jour."
    else
git clone $_gitroot || return 1
        msg "Les fichiers ont été téléchargés."
    fi
cd $_gitname

    msg "Récupération des sources terminée"
    msg "Début de la compilation"

    make || return 1

    mkdir -p $pkgdir/usr/bin/
    mkdir -p $pkgdir/usr/share/palne/
    mkdir -p $pkgdir/usr/share/doc/palne/
    mkdir -p $pkgdir/usr/man/man6/
    mkdir -p $pkgdir/etc/palne/

    cp bin/palne $pkgdir/usr/bin/
    chmod +x $pkgdir/usr/bin
    cp doc/palne.6.gz $pkgdir/usr/man/man6/
    cp doc/index.html $pkgdir/usr/share/doc/palne/
    cp -r data/images/ $pkgdir/usr/share/palne/images/
    cp data/conf/palne.cfg $pkgdir/etc/palne/
}
```

Appliquer un patch

Maintenant, éloignons-nous un peu des PKGBUILD pour nous intéresser à deux commandes très utiles, j'ai nommé diff et patch.

diff sert à calculer la différence entre deux ensemble de fichiers tandis que **patch** sert, à partir du résultat de la commande précédente et des sources, à obtenir la version modifiée des sources.

Pour comprendre le fonctionnement de ces commandes, je vous propose donc cet exemple simple :

Nous avons deux fichiers, code.c.old et code.c :

Code : C

```c
#include <stdio.h>

int main(int a, char **b)
{
    printf("Hello world!");
    return 0;
}
```

Code : C

```c
#include <stdio.h>

int main(int a, char **b)
{
    printf("Bonjour le monde !");
    return 0;
}
```

La seule chose qui change entre ces deux codes est le texte, l'un en anglais, l'autre en français.

Pour créer un patch, on va donc exécuter la commande diff ainsi :

Code : Console

```
diff -c code.c.old code.c > code.diff
```

Si vous ouvrez le fichier code.diff, vous aurez un résultat semblable à celui-ci :

Code : Diff

```
*** code.c.old 2010-05-13 16:49:09.000000000 +0200
--- code.c 2010-05-13 16:47:58.000000000 +0200
***************
*** 2,7 ****

  int main(int a, char **b)
  {
! printf("Hello World!");
      return 0;
  }
--- 2,7 ----

  int main(int a, char **b)
  {
! printf("Bonjour le monde !");
      return 0;
  }
```

Bien sûr, en général, vous allez devoir comparer plusieurs fichiers pour un même patch.
Pour cela, vous allez créer un répertoire contenant les anciennes sources, code-old par exemple, et un autre contenant les sources à jour, code. La commande sera donc :

Code : Console

```
diff -cr code-old/ code/ > code.diff
```

Où -r est le mode récursif.

C'est bien beau mais à quoi sert ce fichier .diff si on ne le réutilise pas ?
Pour ce faire, on va utiliser la commande patch dans le répertoire code-old ainsi :

Code : Console

```
patch -p0 < code.diff
```

Où -p0 est le mode récursif.

Voilà, maintenant que nous connaissons le fonctionnement de la commande patch et que nous avons notre diff, nous allons pouvoir utiliser tout ceci dans un paquet et ce n'est pas plus compliqué que ça !

Tout d'abord, il faudra héberger votre diff quelque part, et le rajouter dans le tableau **source** ainsi que son md5sums dans le tableau correspondant :

Code : Bash

```
source=(http://palne.net/$pkgname-$pkgver.pkg.tar.gz
http://palne.net/patch.diff)
md5sums=('40972633cea6c8d38ff26dade3eec51e'
'8def8b39aceae8e27fdcf2af107991cb')
```

Juste avant la compilation, dans notre fonction build(), on ajoute un simple :

Code : Bash

```
patch < $srcdir/patch.diff
```

Et voilà, les sources du paquet sont patchées !
Et voilà, vous pouvez être fier car vous avez découvert le dur boulot de mainteneur.

Perkele