AUR Helper

An AUR helper is a tool that automates the process of installing packages from the Arch User Repository.

It does this by automating the following tasks:

- search the AUR for published packages
- resolve dependencies for AUR packages
- retrieval and build of AUR packages
- show user comments
- submission of AUR packages

AUR packages are distributed in the form of `PKGBUILD`s that contain information on how the package needs to be built, what dependencies is needs and all the usual metadata associated with every other Arch Linux package.

Arch Wiki has a list of AUR helpers with comparison tables

> **WARNING:** The AUR is entirely user-maintained with no formal security audit by the Arch Linux maintainers. If you install ***anything*** from the AUR, thoroughly inspect the `PKGBUILD` files and verify they contain no malicious instructions!

Installation

The installation procedure for any AUR helper is largely the same, as they are all published on the AUR itself.

Building packages from the AUR manually will at minimum require the `base-devel` and `git` packages:

```
pacman -S base-devel git
```

> **ATTENTION:** If you'rere currently logged in as the `root` user, you need to switch to a regular user profile with `su username`, as `makepkg` will not allow you to run it as `root`.

Change to a temporary directory, clone the AUR helper of your choice with `git`, change into the newly created directory and call `makepkg` to build and install it, e.g. `paru`:

```
cd /tmp
git clone https://aur.archlinux.org/paru
cd paru
makepkg -si
```

`makepkg -si` will prompt you to install any missing dependencies for your chosen AUR helper and call `pacman` to install the helper for you after the build has finished.

Configuration

`makepkg` can be configured to make better use of available system resources, improving build times and efficiency.

One of these optimizations is instructing `makepkg` to pass specific options to compilers. You can either edit the main configuration file of `makepkg` at `/etc/makepkg.conf` or supply a drop-in config file in `/etc/makepkg.conf.d/*.conf` — the latter is recommended in case building starts to act strangely and you want to quickly be able to revert changes by deleting drop-in config files.

Optimizing builds

By default, `makepkg` is configured to produce generic builds of software packages. Since `makepkg` will mostly be used to build packages for your own personal machine, compiler options can be tweaked to produce optimized builds for the machine they're getting built on.

For example, create a drop-in config file `/etc/makepkg.conf.d/cflags.conf` with the following contents:

```
CFLAGS="-march=native -O2 -pipe -fno-plt -fexceptions \
        -Wp,-D_FORTIFY_SOURCE=3 -Wformat -Werror=format-security \
        -fstack-clash-protection -fcf-protection \
        -fno-omit-frame-pointer -mno-omit-leaf-frame-pointer"
```

This will cause GCC to automatically detect and enable safe architecture-specific optimizations.

The same thing can be applied to the Rust compiler. There is already a drop-in config file at `/etc/makepkg.conf.d/rust.conf` that can be edited:

```
RUSTFLAGS="-C opt-level=2 -C target-cpu=native"
```

The `opt-level` parameter can be set to different values ranging in different levels of optimizations that will have an impact on build time. See the Rust docs for details.

Additionally, the `make` build system can also be optimized with the `MAKEFLAGS` variable. One such optimization is to increase the number of jobs that can run simultaneously.

Create a drop-in config file `/etc/makepkg.conf.d/make.conf` with the following contents:

```
MAKEFLAGS="--jobs=$(nproc)"
```

This will prompt `make` to utilize the maximum number of CPU cores to run build jobs.

> **ATTENTION:** Some `PKGBUILD`s specifically override this with `-j1`, because of race conditions in certain versions or simply because it is not supported in the first place. If a package fails to build you should report this to the package maintainer.

Prevent build of `-debug` packages

By default, `makepkg` is configured to also generate debug symbol packages. This affects all AUR helpers. To turn this behavior off, modify the `OPTIONS` array with a drop-in config file `/etc/makepkg.conf.d/options.conf` and either remove the `debug` option entirely or disable it with a `!` in front of it:

```
OPTIONS=(strip docs !libtool !staticlibs emptydirs zipman purge !debug lto)
```

Using the mold linker

`mold` is a drop-in replacement for `ld`/`lld` linkers, which claims to be significantly faster.

Install `mold` from the repositories:

```
pacman -S mold
```

To use `mold`, append `-fuse-ld=mold` to `LDFLAGS` in `/etc/makepkg.conf.d/cflags.conf`:

```
LDFLAGS="-Wl,-O1 -Wl,--sort-common -Wl,--as-needed -Wl,-z,relro -Wl,-z,now \
         -Wl,-z,pack-relative-relocs -fuse-ld=mold"
```

This also needs to be passed to `RUSTFLAGS`:

```
RUSTFLAGS="-C opt-level=2 -C target-cpu=native -C link-arg=-fuse-ld=mold"
```

Compression options

By default, `makepkg` will compress built packages with zstd. This is controlled by the `PKGEXT` variable. The compression algorithm used is inferred from the archive extension. To speed up the packaging process, you might consider turning off the compression at the expense of increased storage usage in the package cache:

```
PKGEXT='.pkg.tar'
```

If you need to conserve space, consider keeping compression enabled, but increasing the number of utilized cores by telling `zstd` to count logical cores instead of physical ones with `--auto-threads=logical`:

```
COMPRESSZST=(zstd -c -T0 --auto-threads=logical -)
```

You can also increase the level of compression applied at the expense of longer packaging time, ranging from 1 (weakest) to 19 (strongest), default is 3:

```
COMPRESSZST=(zstd -c -T0 -19 --auto-threads=logical -)
```

Or use the LZ4 algorithm, which is optimized for speed:

```
PKGEXT='.pkg.tar.lz4'
```

Build entirely in RAM

You can pass `makepkg` a different directory for building packages. Since building causes a lot of rapid small file access, performance could be improved by moving this process to a `tmpfs` location that is held entirely in RAM. The variable `BUILDDIR` can be used to instruct `makepkg` to build packages in another location:

```
BUILDDIR=/tmp/makepkg
```

Since `/tmp` is such a `tmpfs` files in this directory are held in RAM. Building packages completely in RAM can therefore speed up data access and help preserve the durability of flash-based storage mediums like SSDs.

Revision #10
Created 2022-02-12 00:41:20 UTC by Sebin
Updated 2026-07-14 19:37:04 UTC by Sebin