Consistent and Portable Linux Builds with TuxMake

Antonio Terceiro

Linaro Virtual Connect, March 2021

About TuxMake

- Command line tool and Python library for building Linux
- Supports different architectures, toolchains, kernel configurations, and make targets.
- Free Software, using the MIT/Expat license.
- Created and maintained by Linaro, as part of TuxSuite, a suite of tools and services to help with Linux kernel development.

About this presentation

- This presentation covers tuxmake 0.17.0. Older or newer versions might differ.
- TuxMake has two interfaces:
  - **Command line interface**, focus of this presentation.
  - Python API, documented on tuxmake.org
- Almost everything is identical between the CLI and the Python API. Exceptions will be noted explicitly.

The 3 facets of TuxMake

1. **Automates common kernel build scenarios**, so developers can replace, or at least greatly simplify, their homegrown kernel build scripts.
2. **Provides curated container images containing toolchains**, so teams can use the exact same build environments when discussing/reproducing issues.
3. **Enables consistent, portable, and reproducible builds**, by combining the previous two points.

Getting TuxMake

You can install TuxMake in a few different ways:

- With pip from PyPI.
- Debian packages from tuxmake.org.
- RPM packages from tuxmake.org.
- Not install at all: TuxMake can be run from the source tree easily, and it depends only on the Python 3 core.

See tuxmake.org for instructions.

TuxMake Terminology

- **target**: part of the kernel tree that TuxMake knows how to build. Examples: `kernel`, `modules`, `dtbs`, ... (=~ `make` targets).
- **toolchain**: a specific toolchain version. Examples: `gcc-10`, `clang-12`.
- **runtime**: how TuxMake will run the build.
  - `null` (default): On the host system, assuming locally installed toolchain.
  - `docker`: On Docker containers, by default with toolchain images provided by the TuxMake project.
  - `podman`: same as `docker` but using Podman.

Basics

```
$ tuxmake
# to reproduce this build locally: tuxmake [... options ...]
[...]
I: config: PASS in 0:00:01.925466
I: default: PASS in 0:00:55.943743
I: kernel: PASS in 0:00:02.507238
I: xipkernel: SKIP in 0:00:00.001982
I: modules: PASS in 0:00:00.658807
I: dtbs: SKIP in 0:00:00.001642
I: dtbs-legacy: SKIP in 0:00:00.001369
I: debugkernel: PASS in 0:00:08.656499
I: headers: PASS in 0:00:03.330007
I: build output in /home/antonio.terceiro/.cache/tuxmake/builds/1
```

The output directory

- Files in the output directory:

```
$ ls /home/antonio.terceiro/.cache/tuxmake/builds/1/
build.log     zImage          config      headers.tar.xz
metadata.json modules.tar.xz  System.map  vmlinux.xz
```

- Always present:
  - `build.log`
  - `metadata.json`
- Others: build artifacts from the built targets.

Beyond the basics

```
$ tuxmake kernel
$ tuxmake kselftest
$ tuxmake --verbose V=1
$ tuxmake --kconfig=allmodconfig --kconfig-add=CONFIG_FOOBAR=y
$ tuxmake --kconfig=/path/to/custom-config
$ tuxmake --target-arch=arm64
$ tuxmake --target-arch=arm64 --runtime=podman
$ tuxmake --target-arch=arm64 --runtime=docker
```

DRY: use config files

- `@name` in the command line references `~/.config/tuxmake/name`.

```
$ cat ~/.config/tuxmake/arm64
--target-arch=arm64
--runtime=podman
$ tuxmake @arm64
```

- Options in `~/.config/tuxmake/default` are used *implicitly* on all builds:

```
$ cat ~/.config/tuxmake/default
--wrapper=ccache
--post-build-hook="git clean -dxf"
```

Note: config files are a CLI-only feature.

Integrating TuxMake in your workflow: hooks

- `--pre-build-hook COMMAND`

  `COMMAND` is run before the build, from the source tree.
- `--post-build-hook COMMAND`

  `COMMAND` is run after a succesfull build, from the source tree.
- `--results-hook COMMAND`

  `COMMAND` is run after a succesfull build, from the output (artifacts) directory.

Note: hooks are a CLI-only feature.

Workflow example 1 - bisecting a build failure

```
$ git bisect start BAD GOOD
$ git bisect run tuxmake [more options]
```

To speed things up, you might want to pass one or all of the following:

- `--wrapper=ccache`
- `--build-dir=/tmp/build`

Workflow example 2 - bisecting a runtime bug

```
$ git bisect start BAD GOOD
$ git bisect run tuxmake \
    --results-hook=/path/to/my-script.sh \
    [more options]
```

`my-script.sh` will be run in the results directory, where it can use the files there (kernel, `modules.tar.xz` etc) to:

- boot in QEMU, run tests.
- copy kernel and modules into a VM, reboot, run tests.
- submit kernel to some online testing service and wait for results (e.g. LAVA).

More workflow integrations – future research

- Two lines of investigation to be followed *Soon*™:
  - `ktest`: use `tuxmake` instead of `make` in a custom `MAKE_CMD`.
  - `tuxrun` is being extracted from TuxSuite's TuxTest service. It boots kernels from multiple architectes on QEMU, and runs tests against them.
- Please send your ideas in.
- Patches welcome.

Thanks

Contact:

- Gitlab repository: gitlab.com/Linaro/tuxmake
- IRC: `#tuxmake` channel on Freenode

Learn more:

- tuxmake.org - website and documentation