llvm-mingw

https://github.com/mstorsjo/llvm-mingw

”How to QA a toolchain on a zero budget”

Martin Storsjö, WineConf 2025

+

MinGW-w64

A complete runtime environment for GCC & LLVM for 32 and 64 bit Windows

What is llvm-mingw?

- A mingw-w64 (Windows) targeting toolchain based on LLVM components
  - Freely redistributable, properly licensed opensource
- Most LLVM components can be used as drop-in replacements in an existing toolchain
  - Clang ↔ GCC
  - LLD ↔ GNU ld
  - compiler-rt + libunwind ↔ libgcc
  - libc++ ↔ libstdc++
  - LLDB ↔ GDB
- llvm-mingw is set up standalone from scratch with all components replaced with LLVM counterparts

Origin story

- Background in multimedia, FFmpeg, VLC
  - Porting FFmpeg to odd mobile platforms since 2006
- Goal: VLC on Windows Phone in 2014
  - VLC is very heavily tied to mingw build tools, building 100 third party libraries, mainly with autoconf/automake
    - Primarily cross compiled from a Unix
  - No upstream support for Windows outside of x86 in upstream binutils/GCC
    - Stalled efforts to have a GCC maintainer work on this

VLC on Windows Phone

- Wrap MSVC cl.exe in a shell script
  - Interpret GCC style options, remap them to corresponding MSVC ones
  - Implement enough to get the build going
- VLC for Windows Phone was shipped, built like this
  - Painful and slow to build
  - Cross compilation is tricky

Better ways to get mingw for ARM

- LLVM had some support for Windows on ARM
- LLVM did support targeting mingw, on x86 - in existing mingw toolchains
- Missing:
  - Linker
  - Building compiler-rt, libunwind, libc++
  - Tying it all together
- Initial efforts by Martell Malone, 2015-2017, I got involved in 2016

New grounds: ARM64

- In 2017, there were lots of rumors around Windows coming to ARM64
- WinSDK contained a handful of EXEs for ARM64, and some libraries. (MSVC didn't include anything targeting ARM64 at this time)
- Wine got initial support for ARM64 by André Zwing
- Tested it out, could successfully run maybe a couple EXEs from WinSDK
- Tried debugging failing ones - crashing in printf
    - Need to implement a different calling convention for printf, in a compiler
    - Familiar with the LLVM codebase, easy to hack up something that seems to work, and got some more executables working

New grounds: ARM64

- Have a mostly working test environment in Wine
- LLVM/Clang got some initial commits for writing COFF-ARM64 in mid 2017, by people from Qualcomm
- Had WinSDK libraries for ARM64, but not the MSVC parts/libraries
- No MSVC compiler
- Preexisting link.exe turned out to be capable of linking ARM64 though
- Improved code generation and linker, to be able to compile small C executables
- Folded the ARM64 effort into the overall LLVM+mingw effort, added initial support for ARM64 in mingw-w64
    - Notable side quest: UCRT support in mingw-w64

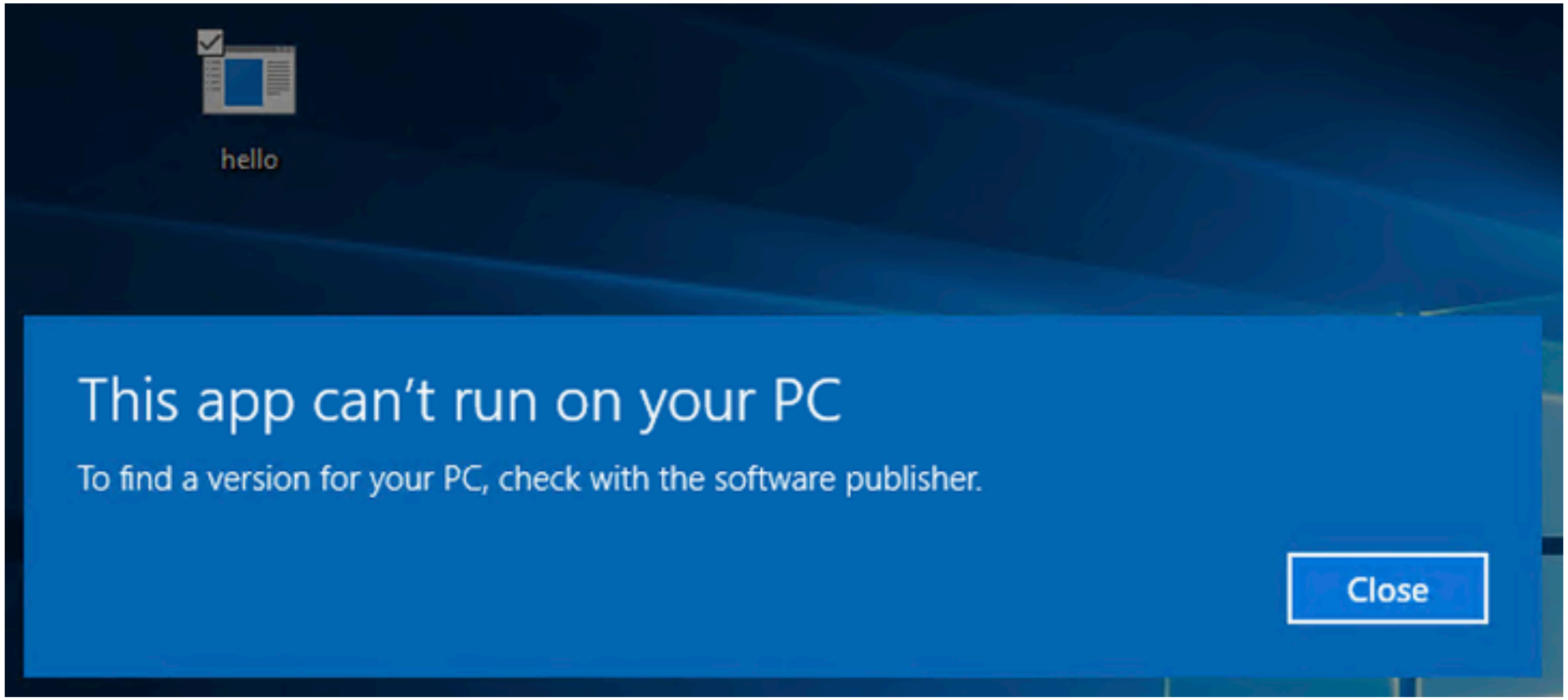
Founding llvm-mingw

- By the end of 2017, had a seemingly working toolchain targeting ARM64
- The mingw linker interface for LLD, by Martell Malone, was finally merged
    - A somewhat working LLVM+mingw environment could be bootstrapped, without any non-upstream patches
- No intent of maintaining a toolchain in itself, but building this setup was kinda complicated, with lots of hacks, so I had to share a reference example of it
    - The project graduated from a branch on VideoLAN's ”docker-buildbots” to a separate repo ”llvm-mingw” on GitHub - by the end of 2017

Testing on the real OS

- VideoLAN managed to get a prototype device from Microsoft
- Could try out my existing toolchain, developed only with Wine, with the real OS now
  - Did it work?

Testing on the real OS



This app can’t run on your PC

To find a version for your PC, check with the software publisher.

Testing on the real OS

- Did it work? Almost - three issues remained:
  - Windows on ARM(64) refuses to load binaries without the dynamicbase flag set
  - Missing stack probing; Wine on Linux doesn't (or didn't) need stack probing then. Simple executables worked, ones that had functions with a large stack crashed
  - setjmp was broken - the UCRT (and msvcrt) setjmp functions require SEH
    - Qt/FreeType does a few setjmp/longjmp
    - Near trivial to implement custom setjmp/longjmp functions
- Manage to make a working build of VLC (with a bunch of hacks) in 2 weeks

MS Build Conference 2018

- Youtube, “Windows 10 on ARM for developers : Build 2018”
  https://youtu.be/vdYIaUeZnqc at 22:17
- "So we shipped him a couple of devices, we loaded them up with Windows, we gave them preview tools and in shockingly short order of time they had this up and running.”
- "When I asked Jean-Baptiste how much code did he have to change, his answer: Zero. He changed zero lines of code. His biggest problem was, again, VLC has its own homebrew way of building things, so ingesting the latest Visual Studio preview compiler, into his build system, that was his biggest challenge. Changing code, zero. He didn't have to change a single line of code, everything just worked.”
  - The preliminary stack of patches was somewhere around 60-70 patches
- Alternative conclusion: It’s more attractive to build a new compiler backend and assemble a new toolchain, than to integrate autotools based projects with the MSVC compiler.

Old hacks

- In 2018, LLVM didn’t have an objcopy/strip tool for PE-COFF
  - Built objcopy/strip from GNU binutils - which doesn’t recognize ARM/ARM64 executables
  - Had a small wrapper, which changed the architecture of the objects/executables to x86, ran the GNU tool, and changed it back to the original
  - Added support for PE-COFF in llvm-objcopy in early 2019
- LLVM’s resource compiler didn’t have integrated preprocessing or a GNU windres compatible interface
  - Had a shellscript wrapper, doing option handling and coordinating preprocessing
  - Integrated preprocessing and added a windres interface in 2021

Maturing

- The llvm-mingw project has matured a lot since then
  - Support for ”auto-import” and runtime pseudo relocations - probably the biggest issue for compatibility with packages that work with mingw
  - Most hacks in the build process removed now
    - (but ARM64EC adds new ones)
  - Providing not only cross compilers, but also toolchains that run on Windows, even on ARM and ARM64, since late 2018
  - MSYS2 now provides ”clang64” environments, which similarly use Clang, LLD, libc++ etc as default toolchain
    - 3447 packages out of 3531 built in this environment

Testing

Avoiding regressions

- LLVM evolves at an absolutely furious pace
  - 100-150 commits per day, on average
  - ~19000 commits per major release (2 times per year)
- Any commit can introduce breakage, and it can be subtle and affect unexpected cases
  - In code generation, the butterfly effect is very likely
    - Any innocent change to one step can cause another step to make a different decision, uncovering preexisting bugs

Avoiding regressions

- Stick to stable releases, and hope someone else has caught and fixed the issues?
  - But what if nobody else tests my cases?
- Breakages do happen occasionally, but are soon fixed/reverted - most of the time, a random git snapshot works just fine
- Continuously test latest git
  - Test one’s own personal use cases primarily
  - Anything that has worked at some point, keep testing it regularly!
- Chromium updates their toolchain from latest LLVM git around every 4 weeks; as long as it passes all their tests, any git snapshot is good enough

LLVM’s own tests

- LLVM/Clang/LLD’s testing is based on narrow unit tests
  - Every functional change must have a test that covers the change
  - Most tests consist of a couple of shell commands together with a small input file
  - No execution of the generated code involved
    - Test assembling a small snippet
    - Test generating code for a small function
    - Check for specific details in the output
  - Any build of LLVM on any machine, with the right targets enabled, will include those tests when running all the tests
- LLVM has got a test suite with larger real-world code examples, but it doesn’t support Windows (yet)
  - Unsure if the coverage it would provide is significantly better than what you get from e.g. ffmpeg

Some potential forms of breakage

- I can no longer build LLVM in the environment where I could before
    - “The GCC in Ubuntu 20.04 no longer can build the project due to tricky C++ corner cases”
- I can no longer build my favourite project, Clang now considers something an error
    - Not a bug: Fix your code
- My code builds fine, but no longer does what it used to
    - Does it have UB? -> Your fault
    - No UB - actual optimizer or code generation bug. The sooner it is caught, the better
        - Step 1: Bisect and find the offending LLVM commit
    - Pure computational tests, like FFmpeg’s FATE tests, can easily run in Wine on Unix
- The code does Windows API/runtime specific things incorrectly
    - Requires actually testing things on Windows
    - Breaks much more rarely, much less by accident

Some potential forms of breakage

Actual code generation/optimization bugs

- Most of these can be caught immediately at compilation if LLVM is built with asserts enabled
  - Much, much easier to deal with a failed compile command, than to trace through code why the code no longer does the right thing at runtime
  - Allows automatically reducing the bug to a minimal example
  - Much easier to argue that there is a bug
  - Push a revert for the offending commit
  - If using an unproven git snapshot of LLVM, you want to run with asserts enabled
    - Clang with asserts is at least 30% slower

Some potential forms of breakage

Actual code generation/optimization bugs

- Reducing a testcase
  - Take the preprocessed output for the failing .c file, replace `-c -o foobar.o` with `-E -o foobar-preproc.c`
  - Check that the same issue reproduces with `foobar-preproc.c` - the issue can now be reproduced standalone outside of the original build setting
  - Run `cvise` (or `creduce`) to minimize the reproducer
    - Provide the preprocessed source and an “interestingness test”
    - Cvise/creduce repeats a set of code simplification/removal operations as long as the interestingness test passes
    - Usually takes you from a 300 KB - 3 MB monster down to a 10 line reproducer, in a matter of minutes

Some potential forms of breakage

Actual code generation/optimization bugs

- Things that doesn’t trigger asserts
  - Find which object file is getting miscompiled
  - Manually pinpoint the incorrect behaviour in the output generated code of that file
    - `clang-good src.c -S -o out-good.s`
    - `clang-bad src.c -S -o out-bad.s`
    - `diff -u out-good.s out-bad.s`
    - Hopefully the issue is easily visible
  - Just ping the author of the change, they can often easily identify what odd cases in their e.g. transformation is triggered by this code
    - “Your commit broke my code, can you have a look?”

Testing latest LLVM regularly

- Compiling LLVM takes a long time. A very long time.
  - Roughly 4-6x as long as building Wine
  - 15-20 minutes on a fast modern machine, >2h on old hardware or Github Actions
- Many times per day, enough files change in LLVM to force rebuilding essentially everything

Local test setup

Original testing setup

- All testing done on Linux, running test executables with Wine
- Old 2011 desktop x86 Linux machine at my office
    - Even a slow old machine can test a lot during night hours!
- Old 2015 ARM64 devboard (Dragonboard 410c) for testing arm/arm64

Updated testing setup

- The old 2011 desktop Linux machine finally died in 2024. A new modern machine is, not surprisingly, very much faster.
- Got a free 4 core ARM64 server from Oracle Cloud in 2021
    - Kinda slow by modern standards
    - Much faster than the ARM devboard, avoids having to build on x86 and cross-test on a different machine over the network

Original testing setup

- Nightly build job
  - Build nightly llvm-mingw, with latest LLVM and mingw-w64
    - Small set of smoke tests to make sure the new build isn’t entirely broken
      - If successful, installed as the latest nightly for use
    - Took over 2 hours with old HW, 20 minutes with new
  - Build and test many open source projects with this toolchain

Local testing setup

Nightly build

- Build latest VLC with nightly llvm-mingw
  - Testing latest VLC, instead of a specific known-good snapshot, gives more valuable test coverage
  - Took up to 2,5 hours with old HW, 15 minutes with new
  - Could only afford building one architecture per nightly run
    - Rotate between building for i686, x86_64, armv7, aarch64
    - Any architecture independent breakage is caught the next day, architecture specific breakage is caught within 4 days
  - Build only, no testing
  - Store built packages locally, to allow testing when wanted
    - Can pinpoint runtime failures to a 4 day window if some breakage is noticed later

Local testing setup

Nightly build

- Build FFmpeg and run its testsuite (FATE)
  - Test all 4 architectures, running in Wine
    - i686 and x86_64 can be tested locally
    - Originally tested armv7 and aarch64 by compiling on x86 and running tests on an ARM devboard (like Raspberry Pi) in Wine
      - These days: Got a free ARM server from Oracle Cloud, running the ARM tests entirely there (both compile and test)
  - This catches many miscompilations in LLVM
- Build a couple of other projects (and run tests for some of them); Qt, x264, dav1d, openh264, libvpx, libaom, libyuv

Local testing setup

Nightly build

- Build latest Wine, on all 4 architectures
  - Build most configurations (ELF, PE, or new-wow64, where applicable) each day
  - Rotate between compilers (nightly Clang or host GCC, mingw or msvc mode for PE)
  - If build is successful and seems to work, install it for use
    - Rotate between configurations (ELF, PE, new-wow64)
- Use nightly Wine for running MSVC, testing building FFmpeg with it

Breakage stats

A period of 173 days; April - September 2023

- LLVM progressed by 18943 commits
- Build of llvm (in our config) + mingw-w64 + runtimes: broken 11 times
- Code generation bug caught by asserts: 9 times
- Silent code generation bug caught at runtime: 2 times
- LLVM bug leading to crash, not caught by asserts: 1 times

- Conclusion: A random git snapshot of LLVM is good enough 87% of the time

Manual testing on Windows

- Some tests run on actual Windows, not in Wine, before a release (usually run in a VM)
  - Only doing a few smoke tests
    - Test Address Sanitizer catching an out of bounds write
    - Test LLDB
      - Catch a crash and look at the backtrace
      - Test both DWARF and PDB debug info formats
      - Stop at a breakpoint, continue from it
  - Others probably test these things regularly, but mostly in MSVC setups. Should test the mingw setup we ship.
- Manual testing is boring, easily gets deferred/skipped/forgotten.

More testing

libc++ testing

- libc++ took a hard stance; platforms must be set up in their CI in order to be considered “supported”
  - libc++ had had regularly run tests on Windows (in MSVC/clang-cl setups) at some point, but it had been out of service for a couple years - the bitrot was quite significant; hundreds of tests were failing on Windows
  - Google had set up Windows based CI machines for other LLVM things; took them in use for libc++ CI
    - Lots of work to clean up the tests to either fix, or mark as XFAIL with an explanation
    - Added llvm-mingw in the CI runner images, added libc++ CI configurations with a mingw context
      - Now considered formally supported! \o/
  - libc++ tests have uncovered a couple of bugs in mingw-w64 CRT routines as well, and code generation bugs

GitHub Actions

- GitHub actions provides free CI runners
    - Slow, but free for public repos
        - Each job is limited to 6 hours of run time, but a CI pipeline can have many jobs (and many in parallel - up to 20 running at a time)
    - They do provide macOS and Windows based runners, in addition to Linux
        - Allows doing scripted testing on real Windows, without fiddling with a VM
        - Since 2025, they also have ARM Linux and Windows runners

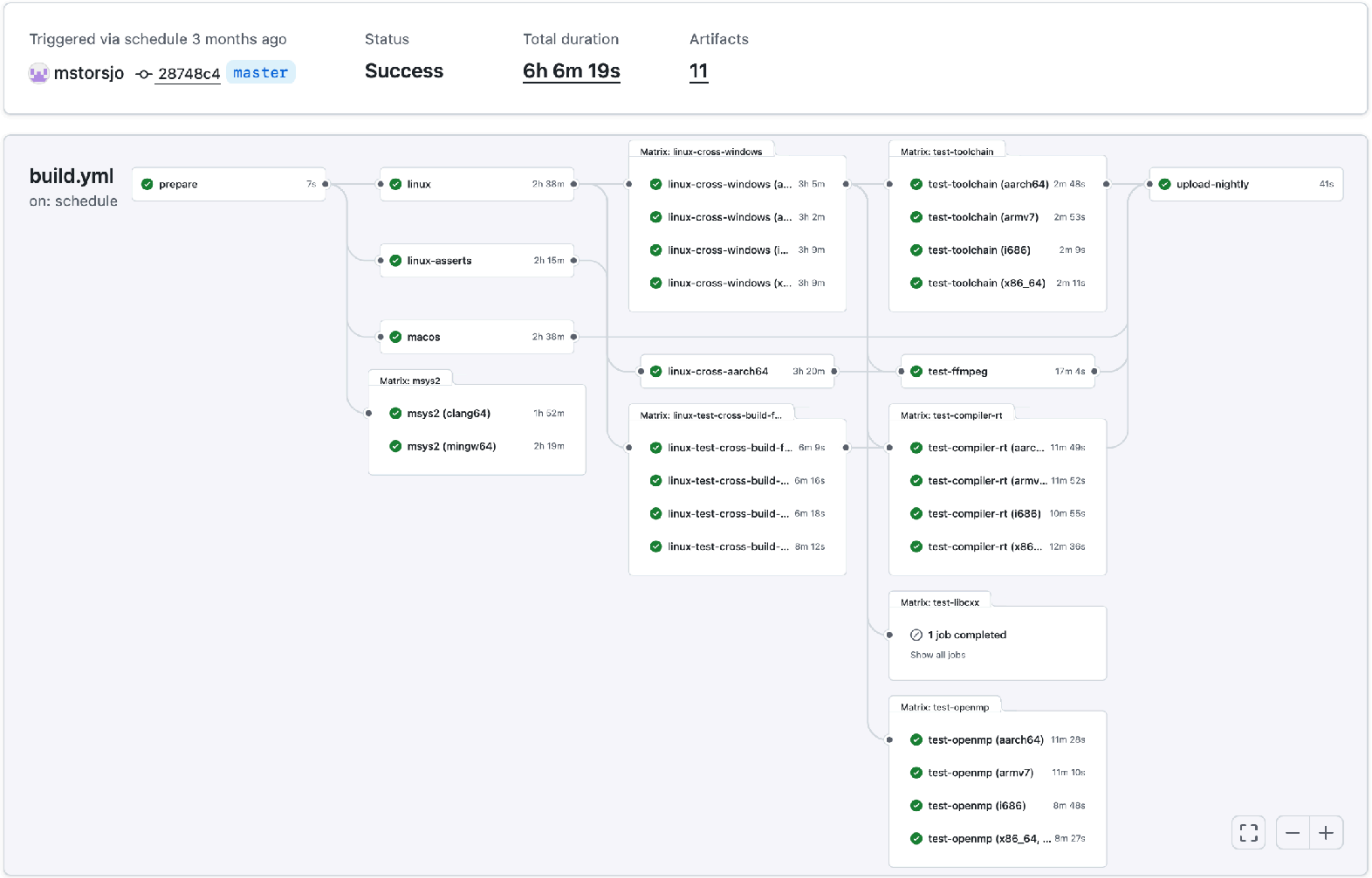
GitHub Actions

- Set up a nightly build
  - Building all the release configurations, and running some tests on it
- Using the Windows-hosted toolchains to run the LLDB and sanitizer smoke tests
  - No longer need to run them manually!
  - Later also fixed up the OpenMP and compiler-rt testsuites for running in mingw configurations, run them nightly as well
  - With Windows/ARM runners now, no more need for any manual testing
    - … but within a couple of months, the runners were updated to Windows 11 24H2, which dropped support for 32 bit ARM 😢

GitHub Actions

Current nightly setup

- Very useful for running various LLVM runtime testsuites on actual Windows
  - For test configurations that might not run properly in Wine
  - Running testsuites for OpenMP, compiler-rt, libc++, and the main LLVM tool build itself
    - Yet to be made working fully in mingw configuration: the LLDB testsuite


Triggered via schedule 3 months ago

mstorsjo -o- 28748c4 master

Status

Success

Total duration

6h 6m 19s

Artifacts

11

build.yml
on: schedule

prepare 7s

linux 2h 38m

linux-asserts 2h 15m

macos 2h 38m

Matrix: msys2
- msys2 (clang64) 1h 52m
- msys2 (mingw64) 2h 19m

Matrix: linux-cross-windows
- linux-cross-windows (a... 3h 5m
- linux-cross-windows (a... 3h 2m
- linux-cross-windows (i... 3h 9m
- linux-cross-windows (x... 3h 9m

linux-cross-aarch64 3h 20m

Matrix: linux-test-cross-build-f...
- linux-test-cross-build-f... 6m 9s
- linux-test-cross-build-... 6m 16s
- linux-test-cross-build-... 6m 18s
- linux-test-cross-build-... 8m 12s

Matrix: test-toolchain
- test-toolchain (aarch64) 2m 48s
- test-toolchain (armv7) 2m 53s
- test-toolchain (i686) 2m 9s
- test-toolchain (x86_64) 2m 11s

test-ffmpeg 17m 4s

Matrix: test-compiler-rt
- test-compiler-rt (aarc... 11m 49s
- test-compiler-rt (armv... 11m 52s
- test-compiler-rt (i686) 10m 55s
- test-compiler-rt (x86... 12m 36s

Matrix: test-libcxx
- 1 job completed

Show all jobs

Matrix: test-openmp
- test-openmp (aarch64) 11m 28s
- test-openmp (armv7) 11m 10s
- test-openmp (i686) 8m 48s
- test-openmp (x86_64, ... 8m 27s

upload-nightly 41s

Extra complication: PGO

- Profile Guided Optimization
  - Build subject application with instrumentation
  - Run typical use cases with instrumented build, gathering a profile
  - Rebuild application, passing the profile to the compiler

Extra complication: PGO

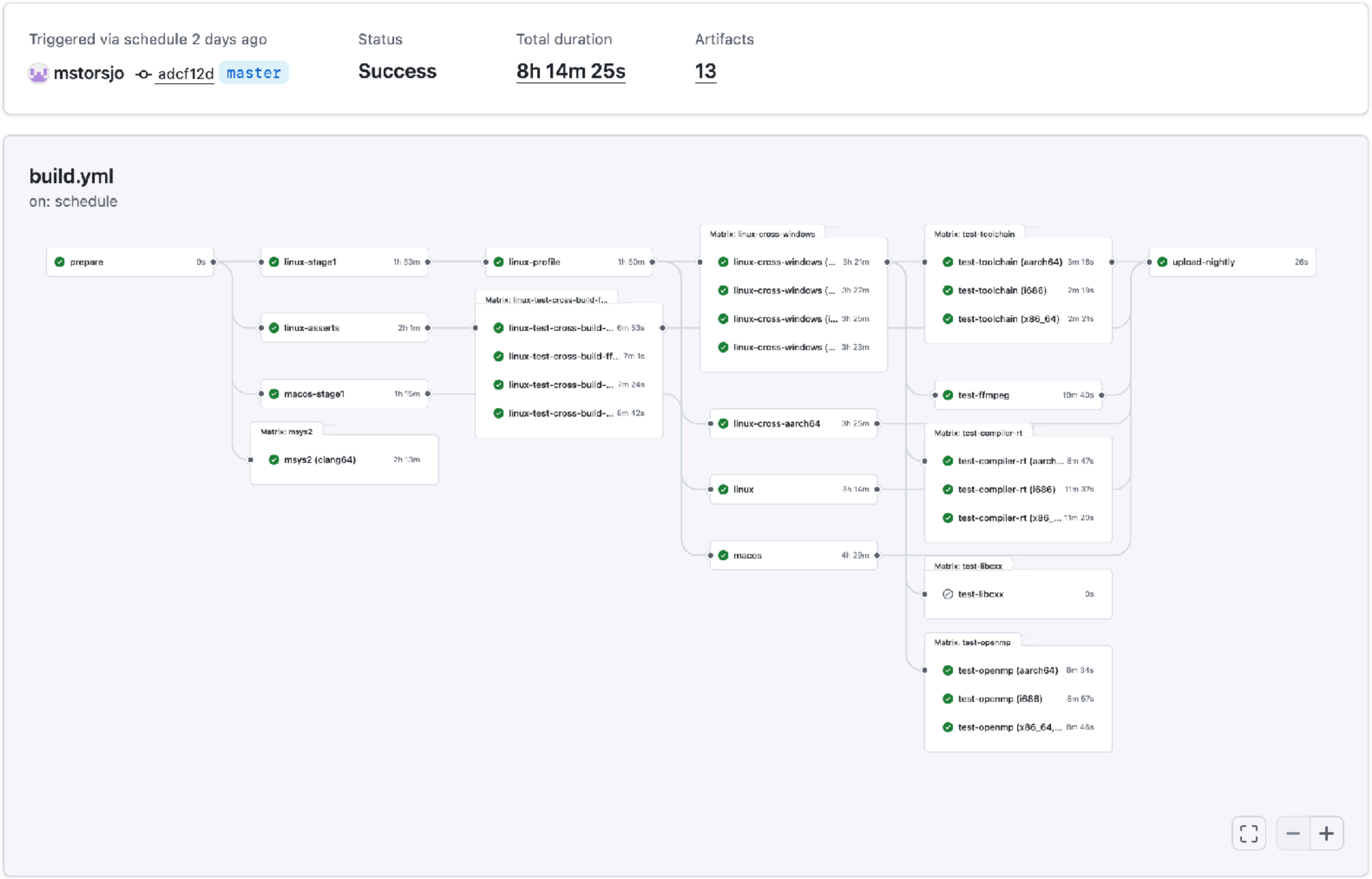
- PGO can make Clang up to 50% faster
- Stage 1: Build Clang once with existing compiler
  - Could use existing host compiler as well
- Instrument: Build Clang with stage1, with instrumentation
- Profile: Use instrumented Clang to exercise compiling code (for all 4 target architectures)
- Optimize: Build Clang with stage1, with profile from previous step, with LTO (Link Time Optimization enabled)

Extra complication: PGO

- Requires 3x the compilation effort
  - We currently provide 10 release packages
- Does not fit in with cross compilation
  - We don’t assume to be able to execute the toolchains we build anywhere during the build process
- Solution: Profile once, on Linux, use the same profile for all final builds
  - Only reaches ~30% speedup, rather than 50%, but at a tolerable cost
  - Only requires building LLVM 2-3 times extra in addition to a straightforward build

Extra complication: PGO

- Extra annoying if there are bugs
- With a plain single-stage build, debugging is easier:
    - Assuming the host compiler isn't at fault - look for a functional bug in the Clang/LLVM code
- With a multi-stage build, debugging is tricky:
    - Say we have a crash in Clang in file A
    - Is the code in file A buggy?
    - Or is code elsewhere in Clang, in file B (from stage 1), buggy, miscompiling file A?
    - May need to bisect stage1 and final builds separately



Triggered via schedule 2 days ago

mstorsjo -o- adcf12d `master`

Status

Success

Total duration

8h 14m 25s

Artifacts

13

build.yml

on: schedule

prepare 0s
linux-stage1 1h 53m
linux-profile 1h 50m
linux-asserts 2h 1m
macos-stage1 1h 15m
Matrix: msys2
msys2 (clang64) 2h 13m
Matrix: linux-test-cross-build-f...
linux-test-cross-build-... 6m 53s
linux-test-cross-build-ff... 7m 1s
linux-test-cross-build-... 7m 24s
linux-test-cross-build-... 6m 42s
Matrix: linux-cross-windows
linux-cross-windows (... 3h 21m
linux-cross-windows (... 3h 27m
linux-cross-windows (i... 3h 25m
linux-cross-windows (... 3h 23m
linux-cross-aarch64 3h 25m
linux 3h 14m
macos 4h 29m
Matrix: test-toolchain
test-toolchain (aarch64) 3m 18s
test-toolchain (i686) 2m 19s
test-toolchain (x86_64) 2m 21s
test-ffmpeg 10m 40s
Matrix: test-compiler-rt
test-compiler-rt (aarch... 8m 47s
test-compiler-rt (i686) 11m 37s
test-compiler-rt (x86_... 11m 20s
Matrix: test-libcxx
test-libcxx 0s
Matrix: test-openmp
test-openmp (aarch64) 8m 34s
test-openmp (i686) 8m 57s
test-openmp (x86_64,... 8m 45s
upload-nightly 26s

That’s all!

https://github.com/mstorsjo/llvm-mingw