musl libc

Szabolcs Nagy
2016

Introduction

- C runtime for Linux
- Pronounced as “mussel” or “muscle”
- First release in 2011 by Rich Felker
- Community project (not driven by a company)
- http://www.musl-libc.org/

Software Stack

Application

libc API & ABI

libc

syscall ABI, ISA

Kernel, HW

APIs – topics

- memcpy, printf, malloc, math, fenv
- pthreads, dynamic linker, crt startup code
- regex, locales, date, time
- networking, sockets, DNS resolver
- syscalls, unistd
- ...

APIs – standards

- ISO C standard library
- POSIX System interfaces
  - C superset
  - About 1500 APIs
- Extensions (BSD, GNU)
- Linux specific APIs

Linux libcs

- glibc – general, kitchen sink
- musl – general, new
- uClibc(-ng) – embedded, configurable
- bionic – android, BSD
- dietlibc – small size, broken
- newlib – baremetal, incomplete
- klibc – initramfs, incomplete

glibc issues – design

- Size (install, code, unshareable data)
- No safe update, unstable internal ABI
- ABI mess (unwind, libgcc_s, IFUNC)
- dlopen nss, gcov, limited static linking
- Hurd
- >= gcc-4.7, >= linux v3.2
- Messy headers
- LGPL, FSF copyright assignment

glibc issues – bw compat

- GNU regex (int instead of size_t)
- Number of signals
- Thread cancel cleanup handling
- malloc hooks (align, as-safety, dlsym, locks)
- Sun RPC
- C++ dtor ABI
- sched_* thread vs process
- Broken APIs (printf hooks, gnu scanf, utmp, ...)

glibc issues – bugs

- Asm overuse (x86 pthread_cond_timedwait)
- Dynlinker races, pthread races
- UTF-8 performance, correctness
- Malloc overuse, OOM handling
- Alloca overuse, unbounded stack usage
- Lot of nasty legacy code

musl vs glibc

- Install size: 4.5M vs 122M
- In memory size: 0.6M vs 1.2M (4.4M, 9.2M)
- Min. dirty: 8K vs 28K (static), 20K vs 48K (dyn)
- Min. static code: 3K vs 510K (vs newlib: 34K)
- Code (c+asm SLOC): 72K+6K vs 935K+262K
- UTF-8 decode performance: 4x glibc
- http://www.etalabs.net/compare_libcs.html

musl – project health

- 2-3 monthly release
- Tracing current standards
  - POSIX.1-2008 support (no TYM, TPP/TPI, dbm)
  - ISO C11 (no Annex K)
- Fast bug fixes, responsive ML, IRC
- Stable ABI
- ~50 contributors, ~5 active at a time

musl design – principles

- Simplicity (minimize bloat, layers, deps)
  - Maintainability (code organization, build system)
  - Hackability (runtime changes, understandability)
  - Security (CVE, hardening)
- Conformance (portable C, POSIX bugs)
- Fail-safety (OOM, corner cases)
- Problematic features are avoided (dlclose, conf)
- Well supported features: UTF-8, static linking

musl design – fail-safety

- No unnecessary failure cases, APIs that cannot report errors cannot fail
- After main is entered all failures are reported
- No lazy binding, no lazy TLS allocation
- Avoid OOM killer (posix_spawn, stack size)
- Avoid races, double free (pthread_cancel)
- Reject >SIZE_MAX/2 allocation

musl design – examples

- Single file libc.so/libpthread.so/ld.so:
  - atomic update, easy deployment
  - no internal ABI
  - always available APIs (no -lrt -lpthread)
- No linux header deps: easy build, clean/correct
- Time conversion: uses long long (no overflows)
- Stack usage: < 16K (worst case is 9K, no alloca)
- Thread and dynlinker is uniform across targets
- Builtin iconv (128K) for most relevant encodings

target support

- Supported target ABIs (13):

  aarch64, arm, microblaze, mips, mips64, mipsn32, or1k, powerpc, powerpc64, sh, i386, x86_64, x32

- work in progress (3):

  riscv32, riscv64, s390

- Only glibc (9):

  alpha, hppa, ia64, m68k, nios2, sparc, sparc64, tile, s390

- Only musl (1):

  or1k

musl users

- Sabotage
- OpenWrt (routers)
- Alpine Linux (Docker)
- Void Linux
- Gentoo (overlay)
- Android NDK
- J-Core toolchain
- Emscripten
- WebAssembly
- Midipix
- OSv
- seL4
- Fuchsia
- (codeplay, chrome)

musl TODO

- Symbol versioning
- IDN support in DNS resolver
- LC_COLLATE
- libc message translations
- Stateful encoding support
- 128bit long double math
- C++11 non-POD TLS dtor
- Priority inheritance/protected mutex
- ARM thumb