Alpaquita Linux
Libc implementations

Alpaquita Linux
Revision 1.0
June 2024

bellsoft

BellSoft software contains open source software. Additional information about third party code is available at https://bell-sw.com/third_party_licenses. You can also get more information on how to get a copy of source code by contacting info@bell-sw.com.

The use of any software product referenced in this document is governed by the applicable license agreement, which is not modified in any way by the terms of this notice.

Contents

1. Overview

Alpaquita Linux has two libc variants based on two different libc implementations, namely `glibc` and `musl`. `musl` is further represented by two variants, such as `musl-default` and `musl-perf`, and can be easily changed on the installed system using the `apk` package manager.

In general, the choice is between `glibc` and `musl-perf/default`. When `musl` is mentioned in the text below, both musl variants are implied.

This document provides a general comparative analysis of the most important differences in libc implementations. For more information and more details, see the following documents:

- Functional differences from glibc
- Comparison of C/POSIX standard library implementations for Linux

2. Locales

Both libc variants use the "C.UTF-8" locale by default. In `musl` it is built in, but in `glibc` it is generated on the package level. Changing the locale in `musl` is difficult and there is still limited support via the `MUSL_LOCPATH` environment variable.

Usually, features that are not in the `musl` library itself, are in the third-party projects including the locales. See the musl-locales project for details.

On the contrary, glibc has an impressive set of locales and it is already available in the `glibc-locales` package.

3. DNS

Unlike `glibc`, `musl` makes parallel requests to the nameservers found in /etc/resolv.conf and only returns the first accepted answer. In glibc, the requests are sequential, that is the next server is requested only after the previous one times out. Both versions are limited to 3 nameservers.

Parallel queries provide better performance in `musl`, but in some setups it may be unacceptable, in which case caching local DNS server can come to the rescue. Caching reduces the CPU and network load and can even speed-up name resolution in `glibc` to overcome a sequential nature of the requests.

Consider a simple setup of a caching server using a flexible tool like dnsmasq as follows.

Setting up a caching DNS server

```
apk add dnsmasq

cat > /etc/resolv.conf << EOF
127.0.0.1
EOF

cat >> /etc/dnsmasq.conf << EOF
port=53
listen-address=127.0.0.1
strict-order
no-resolv
no-poll
server=IP_address_1
server=IP_address_2
EOF

rc-update add dnsmasq default
rc-service dnsmasq start
```

DNS TCP

`musl` did not support TCP prior to version 1.2.3-r15, but now when a UDP response is received with

the `Truncated` flag, `musl` sends a new request over TCP.

DNS directives in `musl` /etc/resolv.conf

`search` directive is available in `musl` since version 1.1.13, hence Alpaquita supports it from the start. `domain` directive (obsolete) behaves identically to the `search` one, even though it was only supposed to have a single entry.

Exposed `options` in `musl`:

- `ndots:n`: default is 1, meaning that if there are any dots in a name, a search list is not used.
- `timeout:n`: default is 5 seconds. The amount of time the resolver waits for the response. Retry interval is calculated as `timeout / attempts`.
- `attempts:n`: default is 2.

The default values for the above options are quite similar to `glibc`.

4. Performance

`musl` is a lightweight library, which is much smaller than glibc. There are no overloaded or tricky functions, versioning, and overuse of malloc.

stream-glibc docker image	stream-musl with musl-perf
8.4 MB	3.33MB

The resource consumption is small too, for example, the default stack size in `musl` is only 128K, unlike the default in `glibc`. This can provide additional benefits for applications with a large number of threads. If necessary, `musl` supports changing the stack size with `pthread_attr_setstacksize` or when linking with the `-wl,-z,stack-size=N` option. The size can be increased or decreased.

memcpy, memmove, memset and others

Speaking of performance, we must mention `musl-perf`. Some consider `glibc` to be faster than `musl` due to optimized string functions for various CPU features, such as avx2, avx512, etc. Now such optimizations are available in the `musl-perf` package that distinguishes it from a regular `musl` implementation. The full list of optimized string functions can be found in the Collection of glibc optimized asm string implementations document.

The best implementation of a string function is chosen at runtime depending on the capabilities of the CPU. The same as in `glibc`. Although, there is a cost in terms of size to having multiple implementations of the functions.

Let's look at the results of Phoronix Stream-Copy benchmarks on an VX2-capable machine. This is just an example of a benchmark that uses only `memcpy()`.

Distro	MB/s
Alpaquita (glibc)	21601
Alapquita (musl-default)	19023

Distro	MB/s
Alpaquita (musl-perf)	21663
Alpine 3.17	19184
Centos 9	21558
Red-Hat 8	20409
Debian 11	20509

The performance of `musl-perf` is pretty much the same as in the Alpaquita `glibc` variant or in other glibc-based distributions.

5. Memory allocators

`glibc` uses `ptmalloc2` (pthreads malloc) general-purpose memory allocator. This is one of the most proven and fastest malloc implementation for multiple threads without lock contention. It has a good speed/memory balance and tunable parameters via `GLIBC_TUNABLES` env var.

`musl` has the so called `mallocng` allocator introduced and used by default since version 1.2.1 in 2020. It is known for its strengthened protection against heap-based overflows, use-after-free, double-free errors, and better fragmentation avoidance.

In addition to the built-in allocators, there are a number of other external allocators that are available for both `glibc` and `musl`:

- `mimalloc`: outperforms other leading allocators (jemalloc, tcmalloc, etc.), and often uses less memory.
- `mimalloc-secure`: mimalloc that has a built-in secure mode, which adds guard pages, randomized allocation, and encrypted free lists to protect against various heap vulnerabilities. Though it has a performance penalty compared to mimalloc around 10% on average over various benchmarks.
- `jemalloc`: implementation that emphasizes fragmentation avoidance and scalable concurrency support.
- `rpmalloc`: a general purpose allocator with lock free thread caching.

To switch to one of these allocators, for example, to mimalloc, we can run the following command in Alpaquita Linux:

```
apk add mimalloc-global
```

Make sure it is used globally:

```
ldd /bin/busybox
  /lib/ld-musl-x86_64.so.1 (0x7f2725f02000)
  /lib/libmimalloc.so.1.7 => /lib/libmimalloc.so.1.7 (0x7f2725e04000)
  libc.musl-x86_64.so.1 => /lib/ld-musl-x86_64.so.1 (0x7f2725f02000)
```

For more information, see the following links:

- Glibc: malloc internals
- Glibc: tunables
- Musl: Mallocng algorithm high-level overview (ML)

- Musl: documentation for mallocng (ML)

6. Indirect Function (ifunc)

Both `glibc` and `musl` support ifunc. Initially, `musl-default` did not support it, but since version 1.2.3-r12 ifunc support was added to Alpaquita as well. This is because the GCC Function Multi-Versioning feature was added to the packages, for example the gzip package, which now requires ifunc support to work since `gzip-1.12-r2`.

7. DT_RELR relative relocation format

The new format can optimally encode R_*_RELATIVE relocations in shared objects and position independent executables (PIE), saving the size of the resulted binary.

Only `musl` supports this feature since version 1.2.3-r9 in Alpaquita Linux. glibc variant with the current version 2.34, does not provide such support yet.

In order to enable compact relative relocations for your builds in Alpaquita musl environment, simply pass the `-Wl,-z,pack-relative-relocs` flag to the linker. As a result, you get a new relocation section `.relr.dyn` and save on the size of the built binary. In some cases the shrinkage can be as much as 5% without additional effort.

Consider the busybox, an app that has over a thousand relative relocations:

busybox	RELASZ	RELACOUNT	RELRSZ	Total Size
RELA	25896	1057	-	833 200
with RELR	528 (-79%)	-	200	808 696 (-2.9%)

The use of RELR results in a 79% reduction in the relocation section size and approximately 2.9% smaller in total size.

For more information, see Relative relocations and RELR.

8. Security

Musl's simplicity and lightweight design makes its mechanism easy to audit, and more importantly, less likely to fail, reducing the attack surface.

Enable FORTIFY_SOURCE

The FORTIFY_SOURCE macro helps detect buffer overflows in various functions that manipulate memory and strings in libc, such as `memcpy`, `strcpy`, etc., providing an additional level of validation for such functions, which are potentially a source of buffer overflow bugs.

To enable it, set `-D_FORTIFY_SOURCE=2` (above 0) and enable the compiler optimizations by the `-O` flag.

`glibc` supports it natively, but `musl` depends on the external headers of another package. To make sure the compiled binary is built with the extra protections enabled, first install the `fortify-headers` package if it is not already installed as follows:

> **Note:**
>
> The `fortify-headers` package is automatically installed if the `build-base` package is installed on `musl`. So it might already be present on your machine.

```
apk add fortify-headers
```

You can review what is included in the package:

```
apk info -L fortify-headers
fortify-headers-1.1-r3 contains:
usr/include/fortify/fortify-headers.h
usr/include/fortify/poll.h
usr/include/fortify/stdio.h
usr/include/fortify/stdlib.h
usr/include/fortify/string.h
usr/include/fortify/strings.h
usr/include/fortify/unistd.h
usr/include/fortify/wchar.h
usr/include/fortify/sys/select.h
```

`usr/include/fortify/sys/socket.h`

Now you can compile the applications with `-D_FORTIFY_SOURCE=2` as usual. `gcc` and `clang` provide a built-in `usr/include/fortify` include on `musl`. Therefore, it is not necessary to additionally specify this path to compilers.

Unfortunately, you need to disassemble the binary to make sure FORTIFY_SOURCE is used in `musl`. With `glibc`, you can check the `*_chk` symbols in the relative functions.

Alpaquita Linux
Libc implementations

bellsoft