Alpaquita Linux
Selecting a malloc implementation

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. Default malloc implementation in libc

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

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

For more information, check the following links:

- Glibc: Malloc internals
- Glibc: Memory Allocation Tunables
- Musl: Mallocng algorithm high-level overview (ML)
- Musl: documentation for mallocng (ML)

2. External general-purpose memory allocators

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 built in secure mode that adds guard pages, randomized allocation, and encrypted free lists to protect against various heap vulnerabilities. It has an around 10% performance penalty in various benchmarks compared to mimalloc.
- jemalloc: implementation that emphasizes fragmentation avoidance and scalable concurrency support.
- rpmalloc: a general purpose allocator with lock-free thread caching.

3. Enable external allocators globally

To switch to one of these allocators, such as to mimalloc, run the following command in Alpaquita Linux:

```
apk add mimalloc-global
```

`-global` packages work with LD_PRELOAD environment variable that is exported via `/etc/profile.d/` configuration scripts. Only one `-global` package can be installed at a time. If the package is installed, the new one replaces the installed package during the installation.

```
apk add jemalloc-global
...
(1/3) Installing jemalloc (5.3.0-r1)
(2/3) Installing jemalloc-global (5.3.0-r1)
Executing jemalloc-global-5.3.0-r1.post-install
*
* Please logout and login to apply the changes to your environment
* or exec 'source /etc/profile.d/jemalloc.sh'
*
(3/3) Purging mimalloc-global (1.7.7-r1)
```

Make sure it is used globally as follows:

```
ldd /bin/busybox
  ldd /bin/busybox
  /lib/ld-musl-x86_64.so.1 (0x7ff28dffc000)
  /lib/libjemalloc.so.2 => /lib/libjemalloc.so.2 (0x7ff28dc00000)
  ...
```

4. Enable external allocators individually

If you want to use external allocators only for specific packages, and even more, if you want to use different external allocators at the same time, you should not use global packages. Instead, manually provide the LD_PRELOAD environment variable only for specific applications.

- Installing several allocators:

```
apk add jemalloc mimalloc
(1/2) Installing jemalloc (5.3.0-r1)
(2/2) Installing mimalloc (1.8.1-r0)
Executing busybox-1.36.0-r7.trigger
OK: 866 MiB in 224 packages
```

- Using jemalloc only for java and mimalloc for python app:

```
jemalloc.sh java app
mimalloc.sh python -m app
```

If you do not want to use LD_PRELOAD and know exactly that the app should use certain external allocators right out of the box, build your app statically with one of these implementations. Alpaquita provides `-dev` and `-static` packages for this case.

5. Debugging glibc malloc

All the debugging features in `glibc` malloc are moved into a separate library named `libc_malloc_debug.so.0` for better security and performance. Perform the following to enable it:

```
export GLIBC_TUNABLES=glibc.malloc.check=3
LD_PRELOAD=/usr/lib/libc_malloc_debug.so.0 ./app
```

For more information, see: Securing malloc in glibc.

6. Selecting memory allocators

Memory allocations in Java

The JVM has its own heap that is maintained by the garbage collector. However, it also uses calls to `malloc()` to implement several Java objects, such as:

- Buffers for data compression routines, which are the memory space that the Java Class Libraries require to read or write compressed data like .zip or .jar files.
- Malloc allocations by application JNI code.
- Compiled code generated by the Just In Time (JIT) Compiler.
- Threads to map to Java threads.

When your app is extensively using such objects it is worth to look and check the performance with alternative allocators.

Memory allocation in Python

Default memory allocator in Python is pymalloc allocator that is optimized for small objects (smaller or equal to 512 bytes) with a short lifetime. For larger allocations, it falls back to the system allocator. Under the hood it can use `mmap()`/`munmap()` and `malloc()`/`free()`. You can disable the `pymalloc` with `PYTHONMALLOC=malloc` environment variable. It might be handy to see the various statistics of the `pymalloc` using the `PYTHONMALLOCSTATS=1` variable.

Even if pymalloc is still used by the Python interpreter by default for small allocations, you can use the techniques described above to specify a different malloc implementation for larger allocations or for allocations in parts of your Python application implemented in C.

Choosing better performance

Let's create a sample Python application. We will not disable pymalloc and only examine the libc malloc allocation there, the number of malloc calls, and the allocated lengths. This practice can show

what the application does at runtime, whether it makes sense to change or experiment with another allocator.

1. First, install the perf tool and python3:

   ```
   apk add perf python3
   ```

2. Let's create an artificial mini-benchmark as shown below that can read the same file line by line (for simplicity, `/proc/kallsyms` file) in parallel using 10 reader-threads.

   Essentially, we are comparing the time it takes to create the Python threads and the time it takes to read an entire file for all reader threads.

   ```
   import threading
   import datetime

   OUTER_THREADS_N = 2
   INNER_THREADS_N = 5

   def read_file():
       with open('/proc/kallsyms') as file:
           for line in file:
               pass

   def spawn_threads(n, fn_name, args):
       threads = []
       for i in range(n):
           t = threading.Thread(target=fn_name, args=args)
           t.start()
           threads.append(t)
       for t in threads:
           t.join()

   dt0 = datetime.datetime.now()
   spawn_threads(OUTER_THREADS_N, spawn_threads, (INNER_THREADS_N, read_file,
   ()))

   diff = datetime.datetime.now() - dt0
   print(f"{diff.total_seconds()}")
   ```

3. Add a dynamic tracepoint for our system allocations in `musl`:

   ```
   perf probe -x /usr/lib/ld-musl-x86_64.so.1 'default_malloc n:u32'
   perf probe -x /usr/lib/ld-musl-x86_64.so.1 'realloc n:u32'
   ```

4. Run memory.py:

```
perf record -z -e probe_ld:default_malloc -e probe_ld:realloc python
memory.py
```

Once the application finished running, you can analyze its output.

- Created tasks overview:

```
perf report --tasks
#      pid      tid     ppid  comm
         0        0       -1 |swapper
      3565     3565       -1 |python
      3565     3567     3565 | python
      3565     3568     3567 |  python
      3565     3569     3567 |  python
      3565     3572     3567 |  python
      3565     3573     3567 |  python
      3565     3574     3567 |  python
      3565     3570     3565 | python
      3565     3571     3570 |  python
      3565     3575     3570 |  python
      3565     3576     3570 |  python
      3565     3577     3570 |  python
      3565     3578     3570 |  python
```

- Various statisitcs:

```
perf report --stats
Aggregated stats:
           TOTAL events:      39845
...
probe_ld:default_malloc stats:
          SAMPLE events:      26821
probe_ld:realloc stats:
          SAMPLE events:      12832
```

- You can see the overhead and the hot malloc lengths:

```
perf report --stdio -n
# Samples: 26K of event 'probe_ld:default_malloc'
# Event count (approx.): 26929
#
# Overhead       Samples  Trace output
# ........  ............  ........................
#
    47.05%         12670  (7fddb81be420) n_u32=8225
     1.67%           450  (7fddb81be420) n_u32=4140
```

```
      1.41%           380  (7fddb81be420) n_u32=4127
      1.37%           370  (7fddb81be420) n_u32=4116
      ...
```

In our example, the length is 8225 bytes. The allocation comes from reading the file. Have a look at the allocations for one of our reader threads (remember, we had 10 such threads):

```
perf report -F pid,overhead,sample,trace --stdio -n
      ...
      3578:python      4.72%          1266  (7fb52688a420) n_u32=8225
      3578:python      0.15%            40  (7fb52688a420) n_u32=4140
      3578:python      0.14%            37  (7fb52688a420) n_u32=4116
      ...
```

From the above example we can see that our simple application calls malloc() approximately 26 thousand times and realloc() 13 thousand times during runtime, and not just at startup with `pymalloc` enabled. Experimenting with different allocators is a good idea here. Continue trying all of them as follows:

```
for i in dummy rpmallocwrap.so jemalloc.so.2 mimalloc.so.1 mimalloc-
secure.so.1; do
    echo "$i:"
    LD_PRELOAD=/lib/lib$i python memory.py
done
```

The table below shows that replacing musl's malloc with even mimalloc-secure improves performance for this workload by ~30%.

	musl	glibc	mimalloc	mi-secure	jemalloc	rpmalloc
musl	0.685	-	0.451	0.472	0.438	0.44
glibc	-	0.439	0.473	0.485	0.447	0.449

Choosing better security

If security is more important than performance, it is better to stick with the default malloc implementation, or use external `mimalloc-secure`. `musl` already has some security mitigations that prevent exploiting bugs in the calling application.

`musl` security features:

- Heap metadata protected by the guard pages.
- Detect and trap any attempt to free a slot that is already free, or an address that is not part of an allocation obtained by malloc.
- Write-after-free detection in the next call in case the metadata becomes inconsistent.
- Detection and trapping of single-byte overflows with arbitrary non-zero values on realloc/free time.

`mimalloc-secure` features:

- All internal mimalloc pages are surrounded by guard pages, and the heap metadata is also behind a guard page, therefore, a buffer overflow exploit cannot reach the metadata.
- All free list pointers are encoded with per-page keys which are used both to prevent overwriting with a known pointer, and to detect heap corruption.
- Double free's are detected (and ignored).
- The free lists are initialized in a random order, and allocation randomly chooses between an extension and reuse within a page to mitigate against attacks that rely on a predicable allocation order. Similarly, the larger heap blocks allocated by mimalloc from the OS have randomized addresses.

Alpaquita Linux
Selecting a malloc implementation

bellsoft