Deterministic Memory Allocation for Mission-Critical Linux

Jim Huang (黃敬群) & Keng-Fu Hsu (許耕福)

Open Source Summit North America / Sep 11, 2017
Background: Phoenix CubeSat

- **2U-CubeSat** for upper atmosphere science and technology demonstration, built by a team of National Cheng Kung University

  - is a part of the QB50 constellation, which will demonstrate the possibility of launching a network of 50 CubeSats built by Universities Teams all over the world as a primary payload on a low-cost launch vehicle to perform first-class science in the largely unexplored lower thermosphere.

- carries 3 mission payloads:
  - An Ion-Neutral Mass Spectrometer (INMS). It is a standard payload of QB50 for upper atmosphere science.
  - Thermistors
  - Solar EUV sensors

Info: http://satellite.ncku.edu.tw/
Background: Rockets in Taiwan

- Advanced Rocket Research Center (ARRC) is a space transport research organization headquartered in Taiwan, with a research focus in hybrid rockets.

- Guidance, navigation and control (GNC) deals with the design of systems to control the movement of spacecraft.
  - alleviating operator work load, smoothing turbulence, fuel savings, automatic or remote control.
  - Powered by Linux PREEMPT_RT in our design!
Memory Allocation Matters
especially for Linux-based real-time systems with recent SoC
Allocator performance matters

- Web server throughput with different memory allocators

Source: Facebook
Why are people still working on?

- glibc
- Streamflow (ISMM06)
- jemalloc (BSDCan06)
- SFMalloc (PACT11)
- Scale Up

![Graph showing the speedup of different memory allocators with respect to the number of threads](image)
People always leave new problems

Kernel contention

User-level contention
Problems in modern memory allocators

- Unstable scalability
  - Critical path contention
  - Global data structure contention
  - Kernel contention
    - Might spend a great amount of time in mmap calls.

- Unstable locality
  - Kernel execution, context switch
  - Allocator data structure operation

- Unstable Latency
  - Algorithm complexity (linked list in glibc, tree structure in jemalloc)
  - Hardware details (pipeline, branch prediction, cache)
Memory problems in typical real-time

- Dynamic memory allocation is used along with
  - A lot of objects, which are referenced by different threads.
  - Their number and lifetime is unpredictable, therefore they should be allocated and deallocated dynamically.

- Heap operations are in conflict with the main demand of real-time systems
  - operations in high priority threads must be deterministic.
Recent RT Considerations

- Many-Core Era
  - Computers with tens of cores are available
    - Even valid for mobile phones:
      - e.g. MediaTek helio P10 >> 2 GHz, Octa-core 64-bit ARMv8-A CPU

- Many-Thread Application
  - Not only server programs but also desktop applications
    - Even mobile Apps, which consist of various background tasks

- Many Applications’ performance heavily relies on memory allocator, such as modern C++ framework
Execution of Memory Allocation

- Depends greatly upon operating system and architecture.
  - Some OS supply an allocator for malloc(), while others supply functions to control certain regions of data.

- Linux uses virtual memory, each process runs in its own virtual AS.
  - Linux dynamically maps the virtual memory to physical memory during runtime.
What’s wrong to default allocator(s)?

- **execution time of memory allocation is not predictable**
  - glibc tries to find an appropriate chunk. Number of available chunks in both containers is dynamic and dependent on runtime conditions.
  - Heap **fragmentation** – there is plenty of free memory available, but no contiguous block is large enough for a given request.

- **use mmap() to map physical in process’ virtual AS**
  - With virtual memory management the physical memory may be swapped on hard disk. It involves heavy IO operations
  - still, **unpredictable** execution time

(there is 6K of free memory, but a request for a block larger than 3K will fail.)
People usually “solve” by memory pool

- memory pool is a common technique
  - pre-allocates memory blocks during startup. While system is running, threads request objects from pool and return it back to pool after usage
  - Optionally use `mlockall()` to lock part or all of the calling process's virtual AS into RAM, preventing that memory from being paged to the swap area.

- single threaded env ⇒ allow allocations with constant execution time

- multi-threading env ⇒ execution time is predictable, but not constant (lock contention)
Ultimate goal: Scalable, Stable, Safe-RT
Memory Allocation at Right Time™
High system reliability by eliminating heap operations
Expected Characteristics

▶ Thread-safe (not TLSF) and deterministic allocations

▶ Track natural structure aware tree-like representation of memory dependencies with aid of low level allocator
  ▶ Freeing an alloc’ed chunk of memory would release all of its dependencies. (like talloc in Samba)

▶ Rather than unmapping chunks, extents of free chunks are managed in userspace to reduce fragmentation, system call overhead and synchronization.
  ▶ Extents of free chunks are managed via address-ordered best-fit, which greatly reduces overall fragmentation and provides logarithmic time complexity for every case where the peak virtual memory doesn't need to be increased.
Structure-aware Allocator

Each chunk of nalloc'ed memory has a header of the following form:

```
+---------+---------+---------+--------···
|  first  |  next   |  prev   | memory |
|  child  | sibling | sibling | chunk  |
+---------+---------+---------+--------···
```

For each chunk of memory allocated, 3 extra pointers are used to maintain the internal structure. The time overhead is quite small.

Alloc hierarchy tree would look like this:

```
NULL <-- chunk --> NULL
  ^
 +--> chunk <-- chunk <-- chunk --> NULL
    |         |         |
    v         v         v
    NULL      NULL      NULL
```

17
Use structure-aware allocator (high-level view)

```c
struct matrix { size_t rows, cols; int **data; }

struct matrix *matrix_new(size_t rows, size_t cols) {
    struct matrix *m = ncalloc(sizeof(*m), NULL);
    m->rows = rows; m->cols = cols; m->data = ncalloc(rowssizeof(*m->data), m);
    for (size_t i = 0; i < rows; i++)
        m->data[i] = nalloc(colssizeof(**m->data), m->data);
    return m;
}

void matrix_delete(struct matrix *m) { nfree(m); /* Simply call nfree() */ }
```

- Straightforward implementation: [https://github.com/jserv/nalloc](https://github.com/jserv/nalloc)
Combine memory pool and custom alloc [RT-Alloc]

- Memory pool pre-allocates memory for user data and its reference counter (refcnt)
- Process acquires memory from memory pool as shared pointer, which uses preallocated refcnt at runtime
- Execution time is therefore predictable
Combine memory pool and custom allocator

- Memory pool pre-allocates memory for user data and its reference counter (refcnt)
- When nobody is using the memory block any longer, shared pointer automatically returns the memory to memory pool.
- Once a memory block is allocated, it will be immediately turned over to a shared pointer.
- Next step is to customize allocator(s)
Review: ptmalloc2
(glibc default)
Mechanisms for multi-threading allocation

- In main thread, heap memory is created by `brk`
  - grow when arena runs out of free space
  - arena can also shrink when there is lots of free space on top chunk.
- Disadvantage: if top chunk keeps allocated, total heap cannot shrink even if other chunks are freed ⇒ memory waste.
Mechanisms for multi-threading allocation

- Thread arena
  - mmap

(Thread arena with multiple heaps)
Mechanism for free-list of memory

- heap organizes all available chunks in two containers
  - Fast-bins (16B-80B) contains small sized chunks for fast allocation, no coalescing, external fragmentation
  - Bins contains normal sized chunks.

- Unsorted bin
  - when small or large chunk gets freed, its gets added into unsorted bin.

- Small bin (16B-512B)

- Large bin (512B-256KB)
RT Analysis: ptmalloc2\textsuperscript{(a)}

- When free() is called, a chunk will be released, which means the chunk is marked as available and dependent on its size, goes to Bins or Fastbins for further allocations.
  - Available chunks in Bins and Fastbins are dynamic and dependent on runtime.
- The allocation time is therefore not predictable.
When malloc() is called, proceed the following steps:

1) If memorysize ≤ maxfast, try to find a chunk in Fastbins.
2) If step 1 failed or size > max fast && size <= DEFAULT MMAP THRESHOLD, try to find a chunk in Bins.
3) If step 2 failed, try to increase the heap size by calling sbrk().
4) If step 3 failed or size > DEFAULT MMAP THRESHOLD, call mmap() to map a physical memory in the process virtual address space.
5) Finally return the allocated chunk, or NULL if steps 1 – 4 all failed.

Number of available chunks in both containers is dynamic and dependent on runtime conditions. ⇒ allocation time is therefore not predictable.
Review: jemalloc
used by Android libc, Facebook, Mozilla Firefox
jemalloc Features

- use thread cache to avoid thread blocking problems, similar to Google’s TCMalloc
- jemalloc maintains a *cache per thread*
  - *Faster* if threads are static
- TCMalloc maintains a *pool of cache*
  - *Faster* when threads are created/destructed frequently
Data structure of jemalloc

- **Arena**
  - Chunk list
  - Chunk1
  - Chunk2
  - Chunk3
  - Chunk4

- **tcache**
  - Arena

**Run_1**
- Page
  - Region
  - Region
- Page
  - Region
  - Region

**Run_2**

- run is a set of one or more contiguous pages
- aligned to multiple of the size
Comparison between TCMalloc and jemalloc

jemalloc is faster for object size > 64KB

TCMalloc is faster in general (< 64KB), but it varies a lot.
Review: Daniel Micay’s allocator
Improvements from Daniel Micay
https://github.com/thestinger/allocator

- Experiments over jemalloc in aggressive ways
- Memory Management in userspace instead of unmapping chunks
  - Reduce fragmentation
  - Reduce system call overhead
  - synchronization
- Natural alignment for chunks
  - distinguish between allocations smaller/larger than chunk size from addresses
  - Find metadata in O(1) for both small/large object
Overcommit & madvise

- Overcommit
  = getting out VM with no guarantee that physical memory for it exits

- madvise
  - syscall to advise kernel about how to handle paging I/O

- Decommit
  - Enable overcommit ⇒ MADV_FREE or MADV_DONTNEED
  - Disable overcommit ⇒ PROT_NONE
Purging

- currently only implemented at a chunk level and does not perform the work lazily.
  - beyond MADV_FREE lazily dropping pages
- Track potential free memory, with lazy purging in FIFO order.
  - The same purging strategy can be used for small/large and chunk allocation.
- Think of Kernel Same Page (KSM) in userspace
Throughput with Overcommit

- When overcommit is disabled, commit charge is dropped.
- Throughput gets worse w/o overcommit
Performance Evaluation

ptmalloc2 vs. jemalloc vs. dmalloc
sysbench - MySQL

- Use sysbench to perform synthetic stress testing of MySQL
  - No difference among all existing memory allocators!
- We need better tools and precise methods to measure!
How to measure allocators?

- Use “ACDC” to benchmark heap management systems
  - \(-s\): min. size (in \(2^x\) bytes)  \(-S\): max. size (in \(2^x\) bytes)
  - \(-F\): object number  \(-n\): number of threads

- Functionalities of ACDC
  - emulate explicit single-/multi-threaded memory allocation, sharing, access, and deallocation behavior to expose virtually any relevant allocator performance differences.
  - study multicore scalability and even false sharing
Evaluation

- **System configurations**
  - Dual 6-Core (2 GHz) Intel Xeon E5-2620 (24 cores with HT)
  - Memory: 8 GB
  - Ubuntu Linux 17.04, kernel v4.10.0

- **Allocators**
  - PTMalloc2 (taken from glibc 2.24)
  - TCMalloc (taken from google-perftools 1.7)
  - jemalloc 2.1.2
  - DMalloc (Daniel Micay’s experiments)
Memory Consumption (4 cores)

- Many allocators have similar behavior when size $\leq 4$KB

TCMalloc consumes lots of memory

PTMalloc2 is the most efficient
Allocation Time (4 cores)

- jemalloc is not best in class; close to TCMalloc

DMalloc is the fastest
Memory Consumption (24 cores)

Both TCMalloc and DMalloc pre-allocate contiguous pages.
Allocation Time (24 cores)

- TCMalloc is better when size <= 64KB
- jemalloc is better when 64KB <= size <= 256KB

DMalloc is the fastest

PTMalloc2 suffers from serious scalability problem
Memory Consumption (16B-512B size)

- jemalloc consumes less memory than TCMalloc.
Allocation Time
(size: 16B-512B)

- TCMalloc performs better than jemalloc
Memory Consumption (size: 512B-128KB)

- jemalloc consumes less memory than TCMalloc
Allocation Time (size: 512B-128KB)

- TCMalloc and jemalloc are close
Memory Consumption (size > 128KB)

- jemalloc consumes less memory than TCMalloc
TCMalloc is still better than jemalloc

**Allocation Time (size > 128KB)**
Improvements over low-level allocator

- **Scalability**
  - Synchronization primitive-free critical path
  - Local memory reuse
  - Lock-free global data structure
  - Excessive VM management calls avoidance (mmap, munmap)

- **Latency**
  - Wait-free algorithm within private heap
  - Shorten critical path

- **Locality**
  - Locality-conscious memory chunk management
  - Allocator false-sharing avoidance
Conclusion

- Analyze performance problems of existing allocators
- To meet real-time constraints, revised memory pool and custom allocator is proposed to fit threading model on modern processors
- Comprehensive performance comparison and evaluation
Reference

- “SSMalloc: A Low-latency, Locality-conscious Memory Allocator with Stable Performance Scalability”, Ran Liu, Haibo Chen
- “Dynamic Memory Allocation and Fragmentation in C & C++”, Mentor Graphics
- “ACDC: Towards a Universal Mutator for Benchmarking Heap Management Systems”, Martin Aigner Christoph M. Kirsch
- “Dynamic Memory Allocation on Real-time Linux”, Jianping Shen, Michael Hamal, and Sven Ganzenmüller