TCMalloc : Thread-Caching Malloc

Sanjay Ghemawat, Paul Menage <opensource@google.com>

Motivation

TCMalloc is faster than the glibc 2.3 malloc (available as a separate library called ptmalloc2) and other mallocs that I have tested. ptmalloc2 takes approximately 300 nanoseconds to execute a malloc/free pair on a 2.8 GHz P4 (for small objects). The TCMalloc implementation takes approximately 50 nanoseconds for the same operation pair. Speed is important for a malloc implementation because if malloc is not fast enough, application writers are inclined to write their own custom free lists on top of malloc. This can lead to extra complexity, and more memory usage unless the application writer is very careful to appropriately size the free lists and scavenge idle objects out of the free list

TCMalloc also reduces lock contention for multi-threaded programs. For small objects, there is virtually zero contention. For large objects, TCMalloc tries to use fine grained and efficient spinlocks. ptmalloc2 also reduces lock contention by using per-thread arenas but there is a big problem with ptmalloc2's use of per-thread arenas. In ptmalloc2 memory can never move from one arena to another. This can lead to huge amounts of wasted space. For example, in one Google application, the first phase would allocate approximately 300MB of memory for its data structures. When the first phase finished, a second phase would be started in the same address space. If this second phase was assigned a different arena than the one used by the first phase, this phase would not reuse any of the memory left after the first phase and would add another 300MB to the address space. Similar memory blowup problems were also noticed in other applications.

Another benefit of TCMalloc is space-efficient representation of small objects. For example, N 8-byte objects can be allocated while using space approximately `8N * 1.01` bytes. I.e., a one-percent space overhead. ptmalloc2 uses a four-byte header for each object and (I think) rounds up the size to a multiple of 8 bytes and ends up using `16N` bytes.

Usage

To use TCmalloc, just link tcmalloc into your application via the "-ltcmalloc" linker flag.

You can use tcmalloc in applications you didn't compile yourself, by using LD_PRELOAD:

```
$ LD_PRELOAD="/usr/lib/libtcmalloc.so"
```

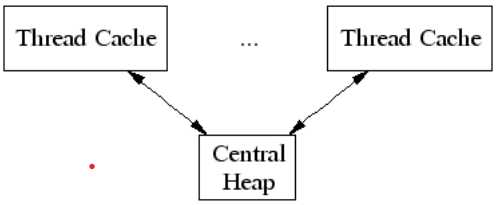
LD_PRELOAD is tricky, and we don't necessarily recommend this mode of usage.

TCMalloc includes a heap checker and heap profiler as well.

If you'd rather link in a version of TCMalloc that does not include the heap profiler and checker (perhaps to reduce binary size for a static binary), you can link in `libtcmalloc_minimal` instead.

Overview

TCMalloc assigns each thread a thread-local cache. Small allocations are satisfied from the thread-local cache. Objects are moved from central data structures into a thread-local cache as needed, and periodic garbage collections are used to migrate memory back from a thread-local cache into the central data structures.



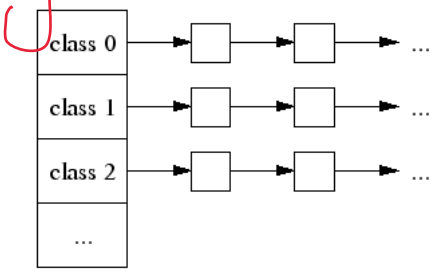
TCMalloc treates objects with size <= 32K ("small" objects) differently from larger objects. Large objects are allocated directly from the central heap using a page-level allocator (a page is a 4K aligned region of memory). I.e., a large object is always page-aligned and occupies an integral number of pages.

A run of pages can be carved up into a sequence of small objects, each equally sized. For example a run of one page (4K) can be carved up into 32 objects of size 128 bytes each.

Small Object Allocation

Each small object size maps to one of approximately 170 allocatable size-classes. For example, all allocations in the range 961 to 1024 bytes are rounded up to 1024. The size-classes are spaced so that small sizes are separated by 8 bytes, larger sizes by 16 bytes, even larger sizes by 32 bytes, and so forth. The maximal spacing (for sizes >= ~2K) is 256 bytes.

A thread cache contains a singly linked list of free objects per size-class.



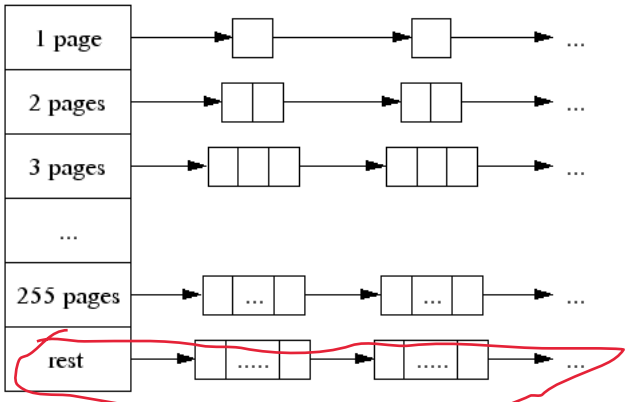
When allocating a small object: (1) We map its size to the corresponding size-class. (2) Look in the corresponding free list in the thread cache for the current thread. (3) If the free list is not empty, we remove the first object from the list and return it. When following this fast path, TCMalloc acquires no locks at all. This helps speed-up allocation significantly because a lock/unlock pair takes approximately 100 nanoseconds on a 2.8 GHz Xeon.

If the free list is empty: (1) We fetch a bunch of objects from a central free list for this size-class (the central free list is shared by all threads). (2) Place them in the thread-local free list. (3) Return one of the newly fetched objects to the applications.

If the central free list is also empty: (1) We allocate a run of pages from the central page allocator. (2) Split the run into a set of objects of this size-class. (3) Place the new objects on the central free list. (4) As before, move some of these objects to the thread-local free list.

Large Object Allocation

A large object size (> 32K) is rounded up to a page size (4K) and is handled by a central page heap. The central page heap is again an array of free lists. For `i < 256`, the `k`th entry is a free list of runs that consist of `k` pages. The `256`th entry is a free list of runs that have length `>= 256` pages:



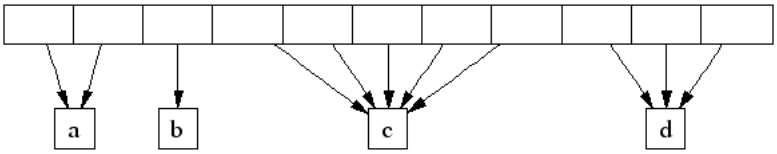
An allocation for `k` pages is satisfied by looking in the `k`th free list. If that free list is empty, we look in the next free list, and so forth. Eventually, we look in the last free list if necessary. If that fails, we fetch memory from the system (using sbrk, mmap, or by mapping in portions of /dev/mem).

If an allocation for `k` pages is satisfied by a run of pages of length > `k`, the remainder of the run is re-inserted back into the appropriate free list in the page heap.

Spans

The heap managed by TCMalloc consists of a set of pages. A run of contiguous pages is represented by a `Span` object. A span can either be allocated, or free. If free, the span is one of the entries in a page heap linked-list. If allocated, it is either a large object that has been handed off to the application, or a run of pages that have been split up into a sequence of small objects. If split into small objects, the size-class of the objects is recorded in the span.

A central array indexed by page number can be used to find the span to which a page belongs. For example, span a below occupies 2 pages, span b occupies 1 page, span c occupies 5 pages and span d occupies 3 pages.



A 32-bit address space can fit 2^20 4K pages, so this central array takes 4MB of space, which seems acceptable. On 64-bit machines, we use a 3-level radix tree instead of an array to map from a page number to the corresponding span pointer.

Deallocation

When an object is deallocated, we compute its page number and look it up in the central array to find the corresponding span object. The span tells us whether or not the object is small, and its size-class if it is small. If the object is small, we insert it into the appropriate free list in the current thread's thread cache. If the thread cache now exceeds a predetermined size (2MB by default), we run a garbage collector that moves unused objects from the thread cache into central free lists.

If the object is large, the span tells us the range of pages covered by the object. Suppose this range is $[p, q]$. We also lookup the spans for pages $p-1$ and $q+1$. If either of these neighboring spans are free, we coalesce them with the $[p, q]$ span. The resulting span is inserted into the appropriate free list in the page heap.

Central Free Lists for Small Objects

As mentioned before, we keep a central free list for each size-class. Each central free list is organized as a two-level data structure: a set of spans, and a linked list of free objects per span.

An object is allocated from a central free list by removing the first entry from the linked list of some span. (If all spans have empty linked lists, a suitably sized span is first allocated from the central page heap.)

An object is returned to a central free list by adding it to the linked list of its containing span. If the linked list length now equals the total number of small objects in the span, this span is now completely free and is returned to the page heap.

Garbage Collection of Thread Caches

A thread cache is garbage collected when the combined size of all objects in the cache exceeds 2MB. The garbage collection threshold is automatically decreased as the number of threads increases so that we don't waste an inordinate amount of memory in a program with lots of threads.

We walk over all free lists in the cache and move some number of objects from the free list to the corresponding central list.

The number of objects to be moved from a free list is determined using a per-list low-water-mark L. L records the minimum length of the list since the last garbage collection. Note that we could have shortened the list by L objects at the last garbage collection without requiring any extra accesses to the central list. We use this past history as a predictor of future accesses and move $L/2$ objects from the thread cache free list to the corresponding central free list. This algorithm has the nice property that if a thread stops using a particular size, all objects of that size will quickly move from the thread cache to the central free list where they can be used by other threads.

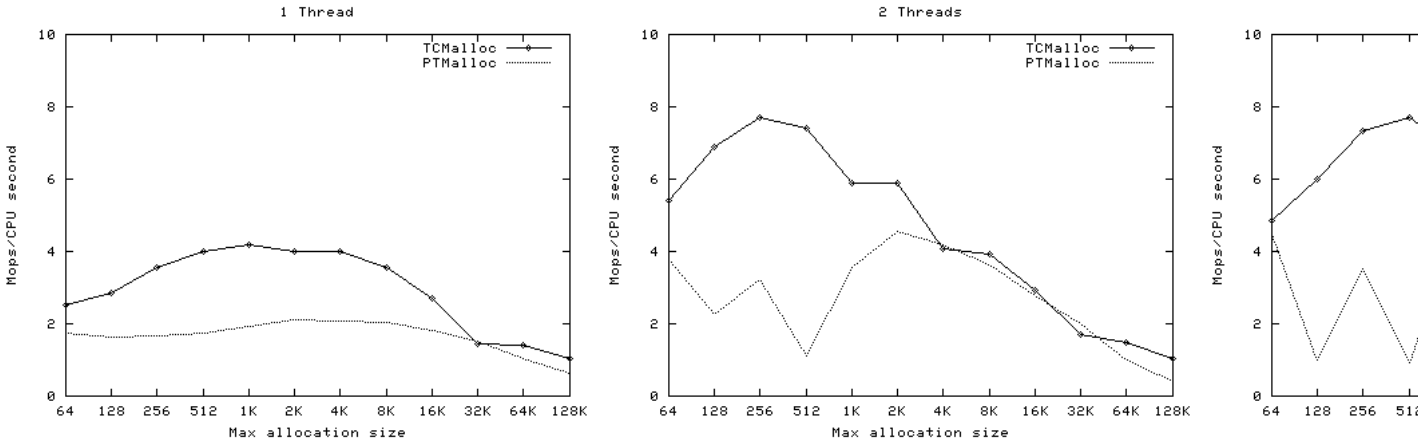
Performance Notes

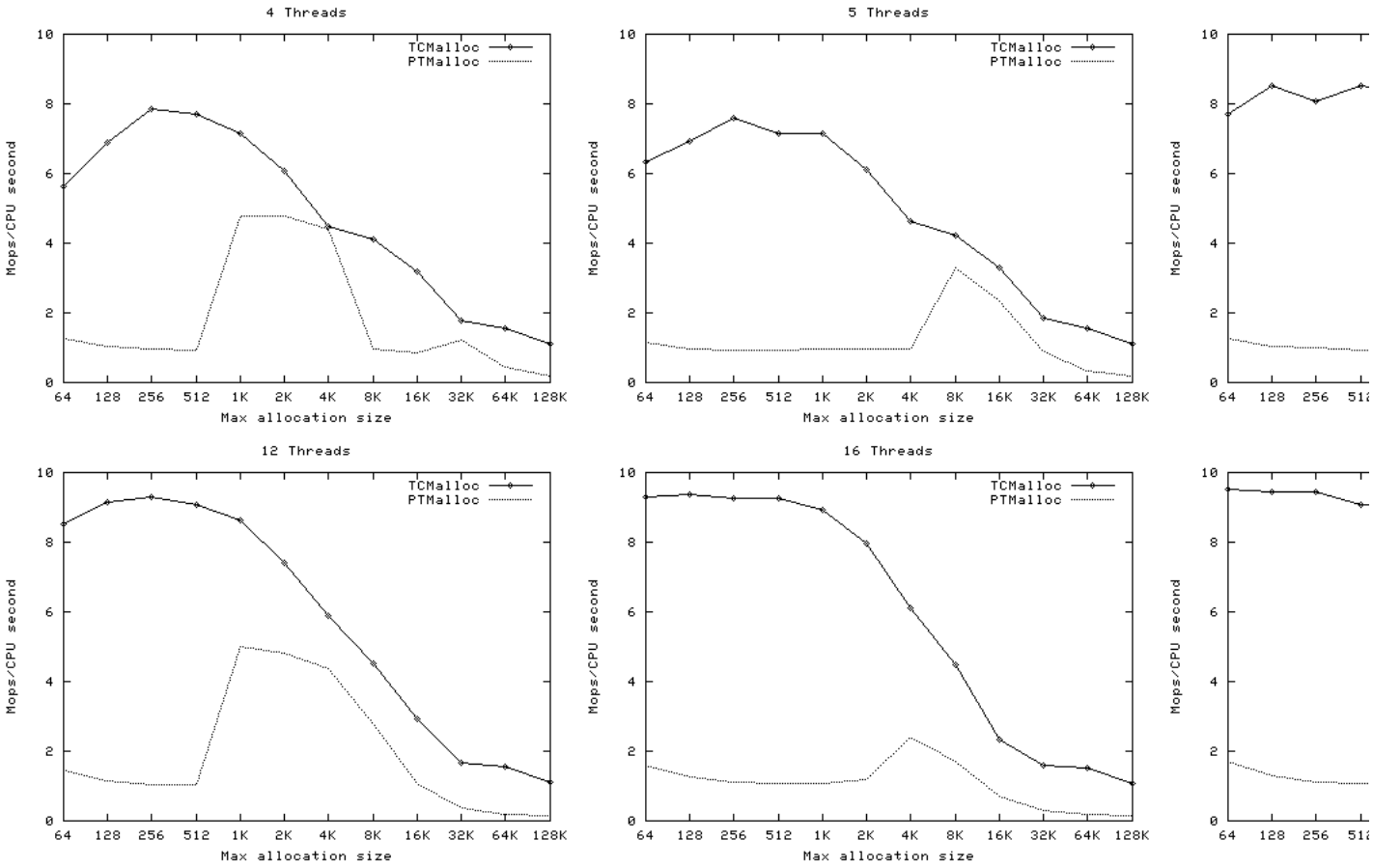
PTMalloc2 unittest

The PTMalloc2 package (now part of glibc) contains a unittest program t-test1.c. This forks a number of threads and performs a series of allocations and deallocations in each thread; the threads do not communicate other than by synchronization in the memory allocator.

t-test1 (included in google-perftools/tests/tcmalloc, and compiled as ptmalloc_unittest1) was run with a varying numbers of threads (1-20) and maximum allocation sizes (64 bytes - 32Kbytes). These tests were run on a 2.4GHz dual Xeon system with hyper-threading enabled, using Linux glibc-2.3.2 from RedHat 9, with one million operations per thread in each test. In each case, the test was run once normally, and once with LD_PRELOAD=libtcmalloc.so.

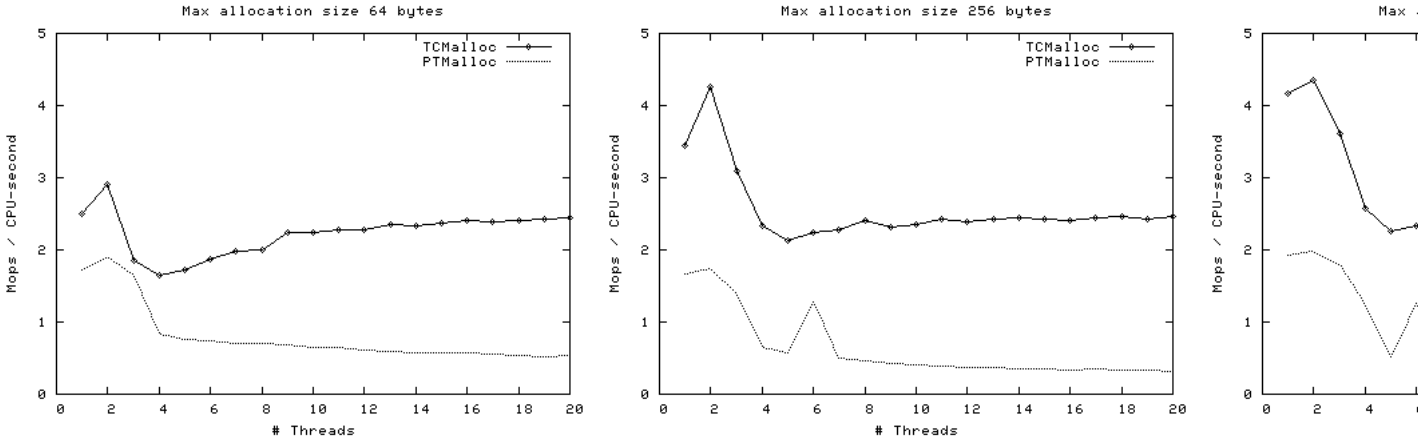
The graphs below show the performance of TCMalloc vs PTMalloc2 for several different metrics. Firstly, total operations (millions) per elapsed second vs max allocation size, for varying numbers of threads. The raw data used to generate these graphs (the output of the "time" utility) is available in t-test1.times.txt.

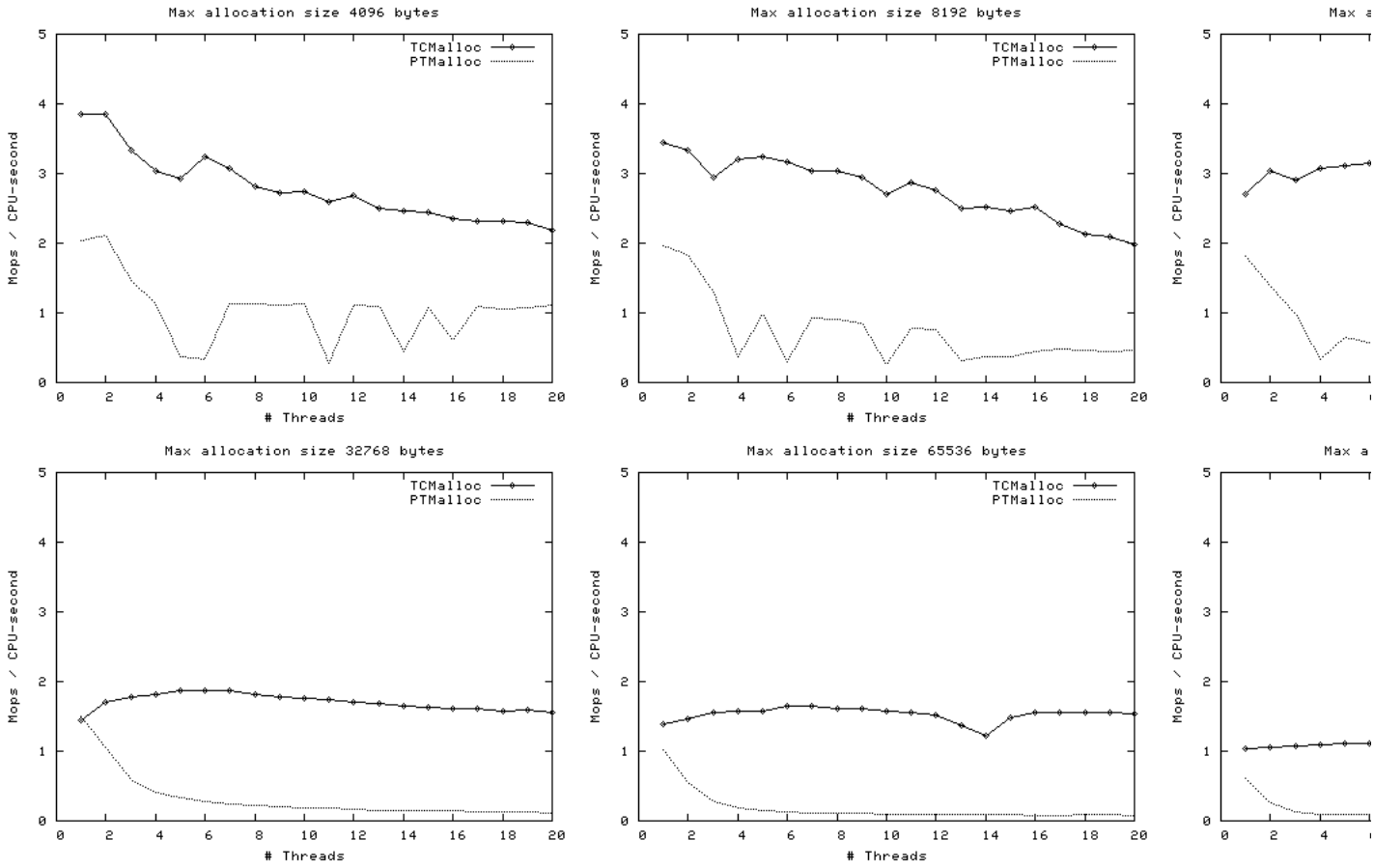




- TCMalloc is much more consistently scalable than PTMalloc2 - for all thread counts >1 it achieves ~7-9 million ops/sec for small allocations, falling to ~2 million ops/sec for larger allocations. The single-thread case is an obvious outlier, since it is only able to keep a single processor busy and hence can achieve fewer ops/sec. PTMalloc2 has a much higher variance on operations/sec - peaking somewhere around 4 million ops/sec for small allocations and falling to <1 million ops/sec for larger allocations.
- TCMalloc is faster than PTMalloc2 in the vast majority of cases, and particularly for small allocations. Contention between threads is less of a problem in TCMalloc.
- TCMalloc's performance drops off as the allocation size increases. This is because the per-thread cache is garbage-collected when it hits a threshold (defaulting to 2MB). With larger allocation sizes, fewer objects can be stored in the cache before it is garbage-collected.
- There is a noticeably drop in the TCMalloc performance at ~32K maximum allocation size; at larger sizes performance drops less quickly. This is due to the 32K maximum size of objects in the per-thread caches; for objects larger than this tcmalloc allocates from the central page heap.

Next, operations (millions) per second of CPU time vs number of threads, for max allocation size 64 bytes - 128 Kbytes.





Here we see again that TCMalloc is both more consistent and more efficient than PTMalloc2. For max allocation sizes <32K, TCMalloc typically achieves ~2-2.5 million ops per second of CPU time with a large number of threads, whereas PTMalloc achieves generally 0.5-1 million ops per second of CPU time, with a lot of cases achieving much less than this figure. Above 32K max allocation size, TCMalloc drops to 1-1.5 million ops per second of CPU time, and PTMalloc drops almost to zero for large numbers of threads (i.e. with PTMalloc, lots of CPU time is being burned spinning waiting for locks in the heavily multi-threaded case).

Caveats

For some systems, TCMalloc may not work correctly on with applications that aren't linked against libpthread.so (or the equivalent on your OS). It should work on Linux using glibc 2.3, but other OS/libc combinations have not been tested.

TCMalloc may be somewhat more memory hungry than other mallocs, though it tends not to have the huge blowups that can happen with other mallocs. In particular, at startup TCMalloc allocates approximately 6 MB of memory. It would be easy to roll a specialized version that trades a little bit of speed for more space efficiency.

TCMalloc currently does not return any memory to the system.

Don't try to load TCMalloc into a running binary (e.g., using JNI in Java programs). The binary will have allocated some objects using the system malloc, and may try to pass them to TCMalloc for deallocation. TCMalloc will not be able to handle such objects.