EE309 Advanced Programming Techniques for EE

Lecture 9: Dynamic allocation (Real-world)

INSU YUN (윤인수)

School of Electrical Engineering, KAIST

Malloc implementation

- dlmalloc – General purpose allocator
  - Old glibc
- **ptmalloc2** – glibc
- jemalloc – FreeBSD and Firefox
- **tcmalloc** – Google
- libumem – Solaris

glibc malloc (ptmalloc2)

- A default allocator for Linux, which we are working with
  - Many similarities with others (e.g., a Windows allocator or others)
- Derived from ptmalloc, which is derived from dlmalloc (Doug Lea malloc)
- “Heap” style: A larger region of memory (heap) can contain chunks of various sizes
- Allows for multiple heaps for one application

ptmalloc2 vs dlmalloc

- Threading support: Performance improvement

- In dlmalloc,
  - when two threads call malloc at the same time ONLY one thread can enter the critical section, since freelist data structure is shared among all the available threads
  - Hence memory allocation takes time in multi threaded applications, resulting in performance degradation

- In ptmalloc2,
  - when two threads call malloc at the same time memory is allocated immediately since each thread maintains a separate heap segment and hence freelist data structures maintaining those heaps are also separate.
  - This act of maintaining separate heap and freelist data structures for each thread is called **per thread arena**.

glibc malloc (ptmalloc2)

- A default allocator for Linux, which we are working with
  - Many similarties with others (e.g., a Windows allocator or others)
- Derived from ptmalloc, which is derived from dlmalloc (Doug Lea malloc)
- “Heap” style: A larger region of memory (heap) can contain chunks of various sizes
- Allows for multiple heaps for one application

Terminologies

- Arena: A structure that is shared among one or more threads which contains references to **one or more heaps**, as well as linked lists of chunks within those heaps which are "free“. Threads assigned to each arena will allocate memory from that arena's **free list**s.

- Heap: **A contiguous region of memory** that is subdivided into chunks to be allocated. Each heap belongs to exactly one arena.

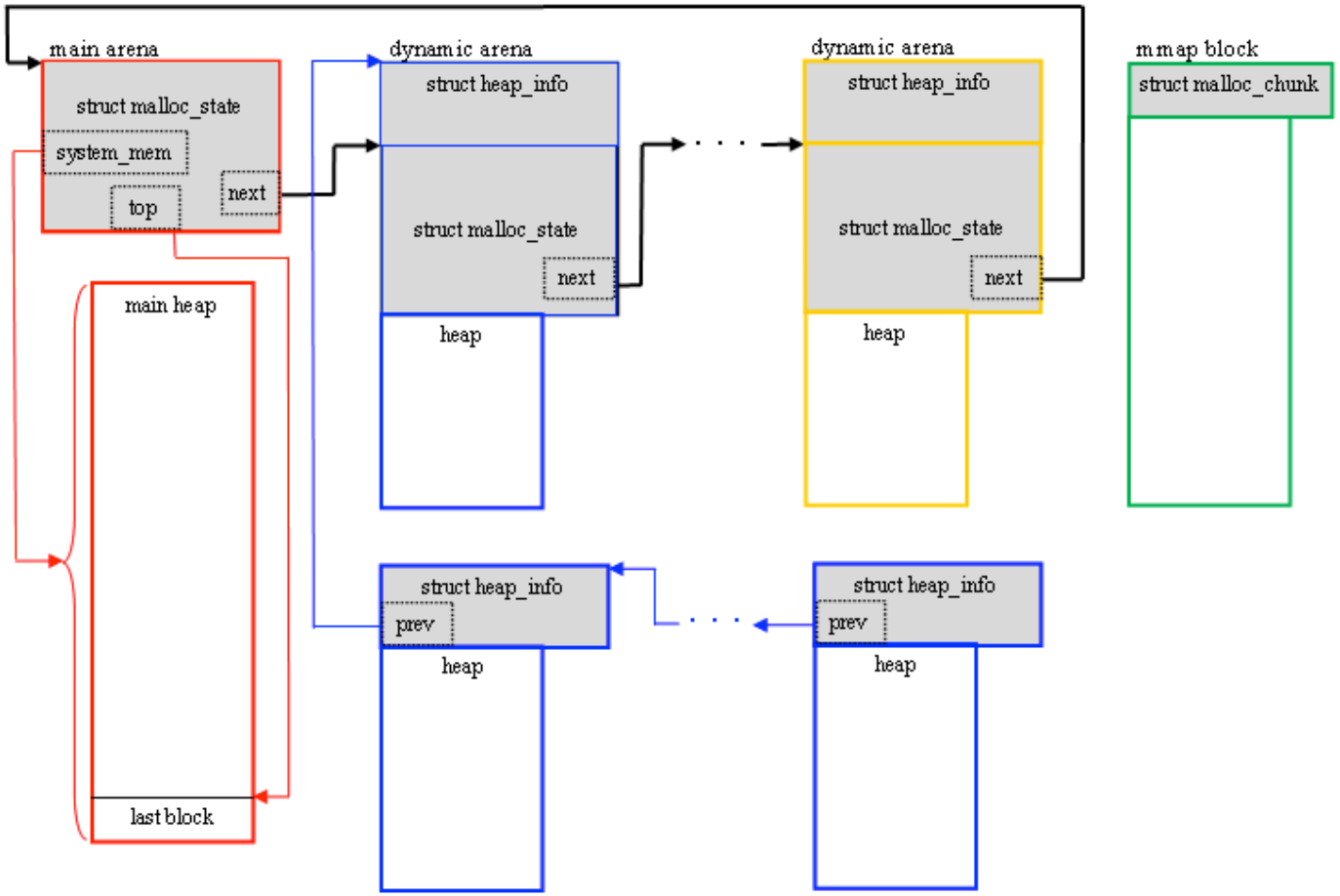
Terminologies

- Chunk: A small range of memory that can be allocated (owned by the application), freed (owned by glibc), or combined with adjacent chunks into larger ranges. Note that a chunk is a wrapper around **the block of memory that is given to the application**. Each chunk exists in one heap and belongs to one arena.

- Memory: A portion of the application's address space which is typically backed by RAM or swap.

Arenas and Heaps

- To efficiently handle multi-threaded applications, glibc's malloc allows for more than one region of memory to be active at a time
  - Different threads can access different regions of memory without interfering with each other
  - These regions of memory are collectively called "arenas“
- main arena: The application's initial heap
  - A static variable in the malloc code
  - Next to link additional arenas
  - For exploit, this is an address in libc
    (i.e., if we leak this memory, we can break ASLR)



http://core-analyzer.sourceforge.net/index_files/Page335.html

malloc_state (glibc 2.31)

```
struct malloc_state
{
  /* Serialize access.  */
  __libc_lock_define (, mutex);
  /* Flags (formerly in max_fast).  */
  int flags;

  /* Fastbins */
  mfastbinptr fastbinsY[NFASTBINS];
  /* Base of the topmost chunk -- not otherwise kept in a bin */
  mchunkptr top;
  /* The remainder from the most recent split of a small request */
  mchunkptr last_remainder;
  /* Normal bins packed as described above */
  mchunkptr bins[NBINS * 2 - 2];

  /* Bitmap of bins */
  unsigned int binmap[BINMAPSIZE];
```

malloc_state (glibc 2.31)

```
  /* Linked list */
  struct malloc_state *next;
  /* Linked list for free arenas.  Access to this field is serialized
     by free_list_lock in arena.c.  */
  struct malloc_state *next_free;
  /* Number of threads attached to this arena.  0 if the arena is on
     the free list.  Access to this field is serialized by
     free_list_lock in arena.c.  */

  INTERNAL_SIZE_T attached_threads;
  /* Memory allocated from the system in this arena.  */
  INTERNAL_SIZE_T system_mem;
  INTERNAL_SIZE_T max_system_mem;
};

typedef struct malloc_state *mstate;
```

Chunks

- Glibc malloc divides a large region of memory (a "heap") into chunks of various sizes.
- Each chunk includes meta-data about how big it is (via a size field in the chunk header), and thus where the adjacent chunks are.
- When a chunk is in use by the application, the only data that's "remembered" is the size of the chunk.
- When the chunk is free'd, the memory that used to be application data is re-purposed for additional arena-related information, such as pointers within linked lists, such that suitable chunks can quickly be found and re-used when needed.
- Also, the last word in a free'd chunk contains a copy of the chunk size (with the three LSBs set to zeros, vs the three LSBs of the size at the front of the chunk which are used for flags).

```
struct malloc_chunk {
  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previou
s chunk, if it is free. */
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes,
including overhead. */
  struct malloc_chunk* fd;                /* double links --
 used only if this chunk is free. */
  struct malloc_chunk* bk;
  /* Only used for large blocks: pointer to next larger size
.  */
  struct malloc_chunk* fd_nextsize; /* double links --
 used only if this chunk is free. */
  struct malloc_chunk* bk_nextsize;
};

typedef struct malloc_chunk* mchunkptr;
```

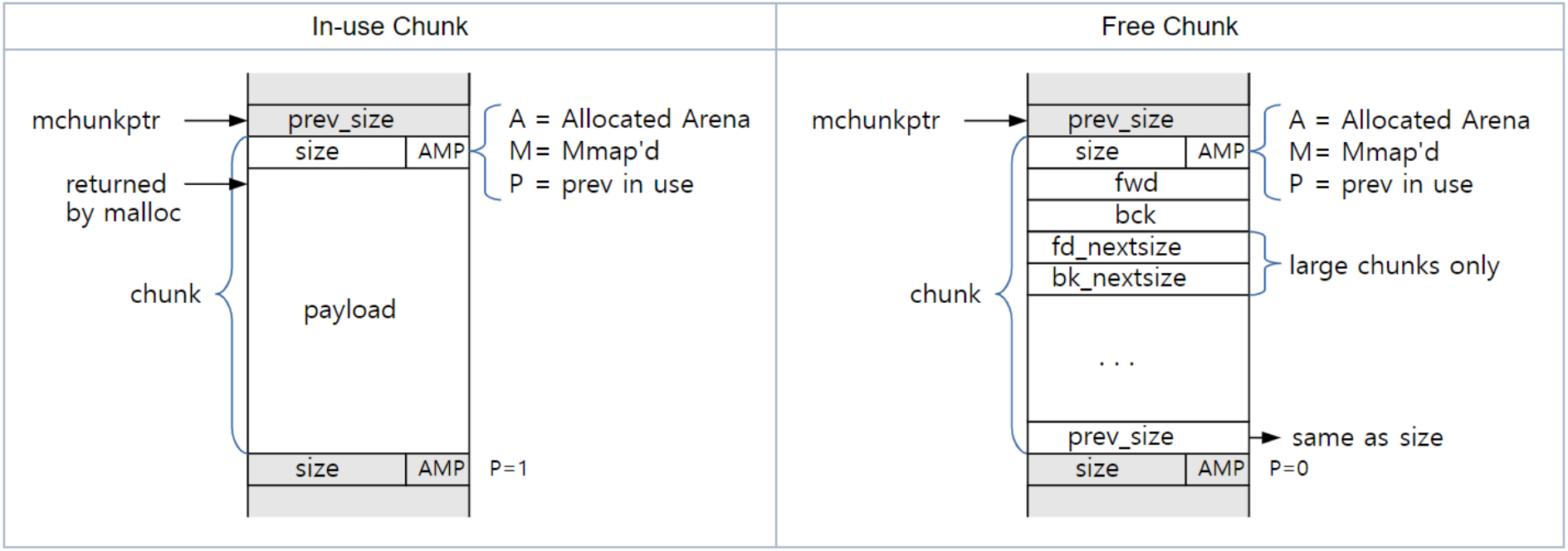
Chunk flags

- Since all chunks are multiples of 8 bytes, the 3 LSBs of the chunk size can be used for flags. These three flags are defined as follows:

- A (0x04): Allocated Arena - the main arena uses the application's heap. Other arenas use mmap'd heaps.

- M (0x02): MMap'd chunk - this chunk was allocated with a single call to mmap and is not part of a heap at all.

- P (0x01): Previous chunk is in use - if set, the previous chunk is still being used by the application, and thus the prev_size field is invalid.


In-use Chunk
Free Chunk
mchunkptr
prev_size
size
AMP
returned
by malloc
chunk
payload
A = Allocated Arena
M = Mmap'd
P = prev in use
size
AMP
P=1
mchunkptr
prev_size
size
AMP
fwd
bck
fd_nextsize
bk_nextsize
chunk
large chunks only
. . .
prev_size
same as size
size
AMP
P=0

Bins

- Within each arena, chunks are either in use by the application or they're free (available).
- In-use chunks are not tracked by the arena.
- Free chunks are stored in various lists based on size and history, so that the library can quickly find suitable chunks to satisfy allocation requests. The lists, called "bins"

Types of bins

- **Fast:** Small chunks are stored in size-specific bins. Chunks added to a fast bin ("fastbin") are not combined with adjacent chunks. Fastbin chunks are stored in an array of singly-linked lists, since they're all the same size and chunks in the middle of the list need never be accessed.

- **Unsorted**: When chunks are free'd they're initially stored in a single bin. They're sorted later, in malloc, in order to give them one chance to be quickly re-used.

Types of bins

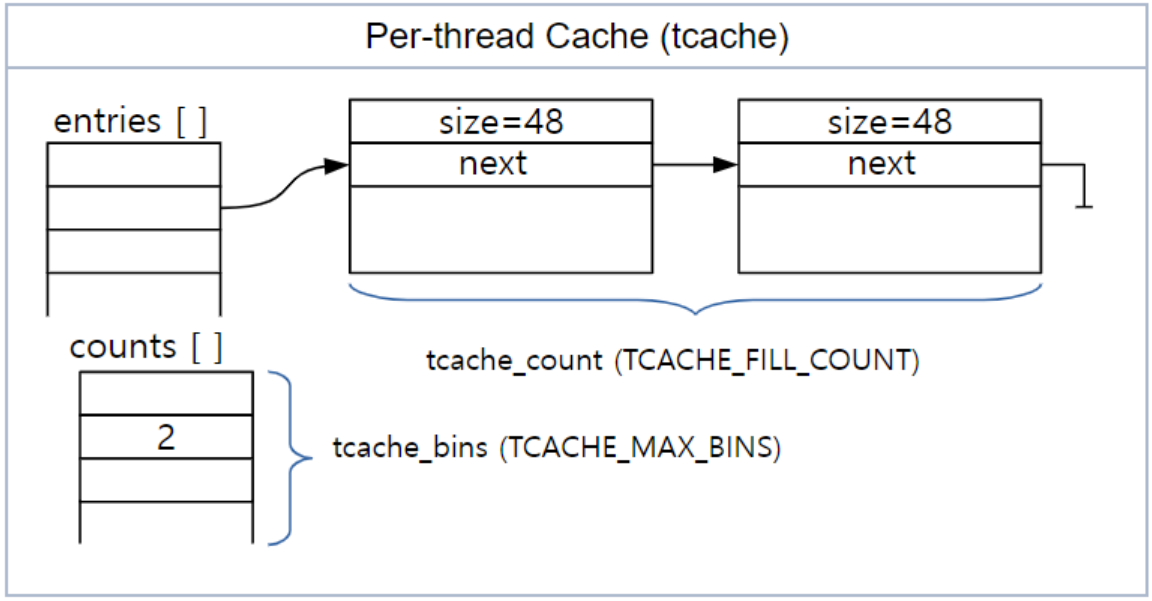
- **Small**: The normal bins are divided into "small" bins, where each chunk is the **same size**. When a chunk is added to these bins, they're first combined with adjacent chunks to **"coalesce"** them into larger chunks. Small and large chunks are **doubly-linked** so that chunks may be removed from the middle.

- **Large**: A chunk is "large" if its bin may contain more than one size. For small bins, you can pick the first chunk and just use it. For large bins, you have to find the "best" chunk, and possibly split it into two chunks (one the size you need, and one for the remainder).

Thread cache (tcache)

- For NUMA architectures, coordinate thread locality, sort threads by core, etc.
- Each thread has a per-thread cache (called the tcache) containing a small collection of chunks which can be accessed without needing to lock an arena.
- These chunks are stored as an array of singly-linked lists
- Each bin contains one size chunk, so the array is indexed (indirectly) by chunk size.
- Unlike fastbins, the tcache is limited in how many chunks are allowed in each bin (tcache_count).
- If the tcache bin is empty for a given requested size, the fallback is to use the normal malloc routines i.e. locking the thread's arena and working from there.

Thread cache (tcache)



Malloc algorithm

- If there is a suitable (exact match only) chunk in the tcache, it is returned to the caller.
- If the request is large enough, mmap() is used to request memory directly from the operating system.
- If the appropriate fastbin has a chunk in it, use that.
- If the appropriate smallbin has a chunk in it, use that.
- … (large)
- Use chunks in unsorted bin (if not, move them to regular bins)
- Split off part of the “top” chunk, possibly enlarging "top" beforehand.

Free algorithm

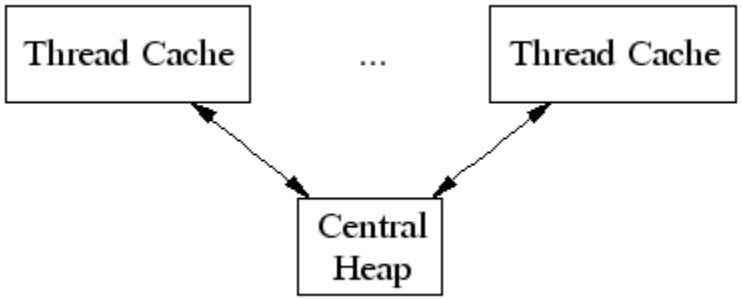
- If there is room in the tcache, store the chunk there and return.
- If the chunk is small enough, place it in the appropriate fastbin.
- If the chunk was mmap'd, munmap it.
- See if this chunk is adjacent to another free chunk and coalesce if it is.
- Place the chunk in the unsorted list, unless it's now the "top" chunk.
- …

Platform-specific Thresholds and Constants

Parameter	32 bit	i386	64 bit
MALLOC_ALIGNMENT	8	16	16
MIN_CHUNK_SIZE	16	16	32
MAX_FAST_SIZE	80	80	160
MAX_TCACHE_SIZE	516	1,020	1,032
MIN_LARGE_SIZE	512	1,008	1,024
DEFAULT_MMAP_THRESHOLD	131,072	131,072	131,072
DEFAULT_MMAP_THRESHOLD_MAX	524,288	524,288	33,554,432
HEAP_MIN_SIZE	32,768	32,768	32,768
HEAP_MAX_SIZE	1,048,576	1,048,576	67,108,864

tcmalloc

- 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

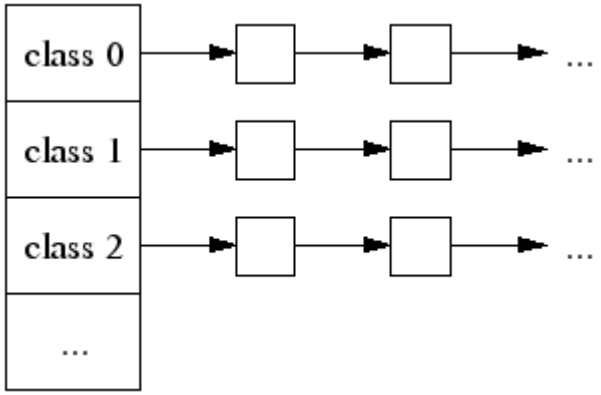
- TCMalloc treats objects with size <= 256K ("small" objects) differently from larger objects.

- Large objects are allocated directly from the central heap using a page-level allocator (a page is a 8K 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 (8K) can be carved up into 64 objects of size 128 bytes each.

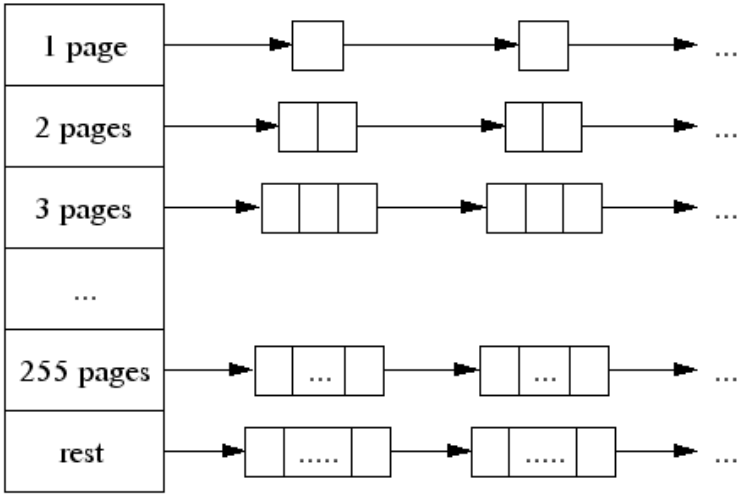
Small object (< 256K)

- Each small object size maps to one of approximately 88 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.
  - A thread cache contains a singly linked list of free objects per size-class.



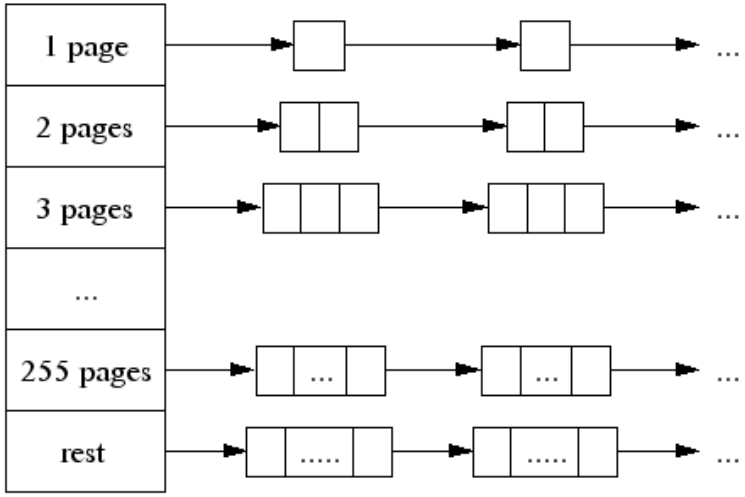
Medium object (256K <= size < 1MB)

- A large object size (> 256K) is rounded up to a page size (8K) and is handled by a central page heap.
- The central page heap is again an array of free lists. For i < 256, the kth entry is a free list of runs that consist of k pages.
- The 256th entry is a free list of runs that have length >= 256 pages:
- An allocation for k pages is satisfied by looking in the kth free list. If that free list is empty, we look in the next free list, and so forth.



Large object (size >= 1MB)

- 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 (use best-fit + red-black tree)

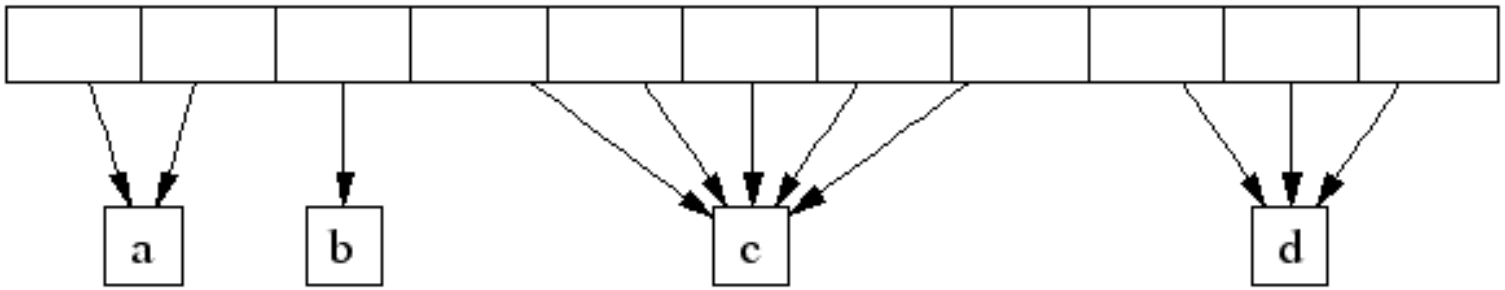


span

- 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.

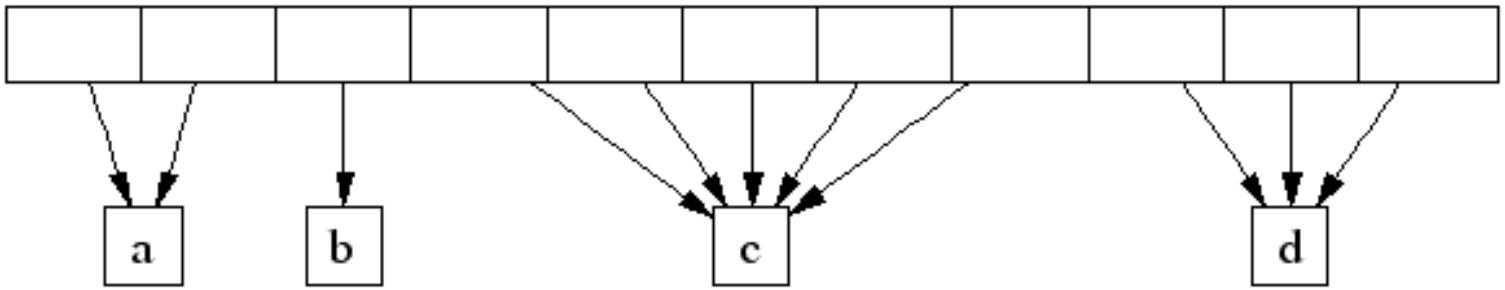
span

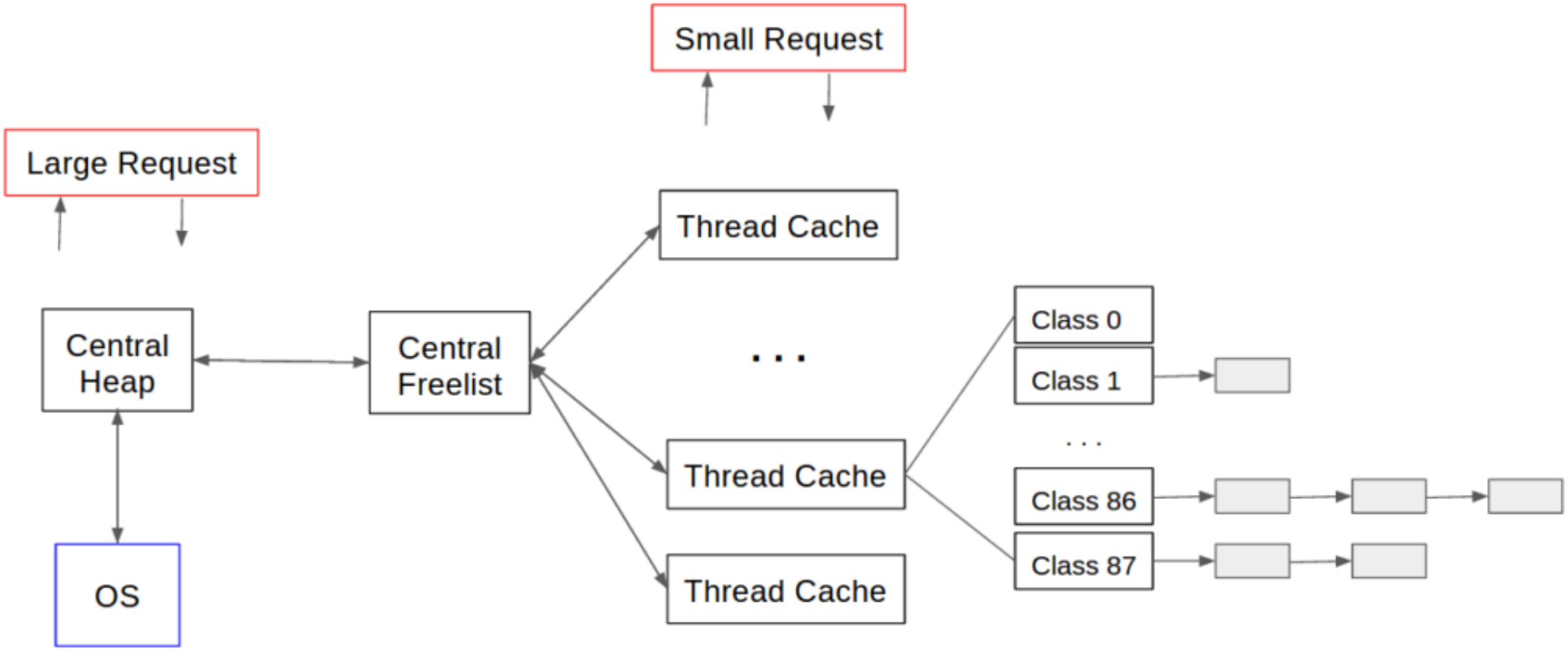
- A central array indexed by page number can be used to find the span to which a page belongs. For example, span a below occupies 2 pages, span b occupies 1 page, span c occupies 5 pages and span d occupies 3 pages.

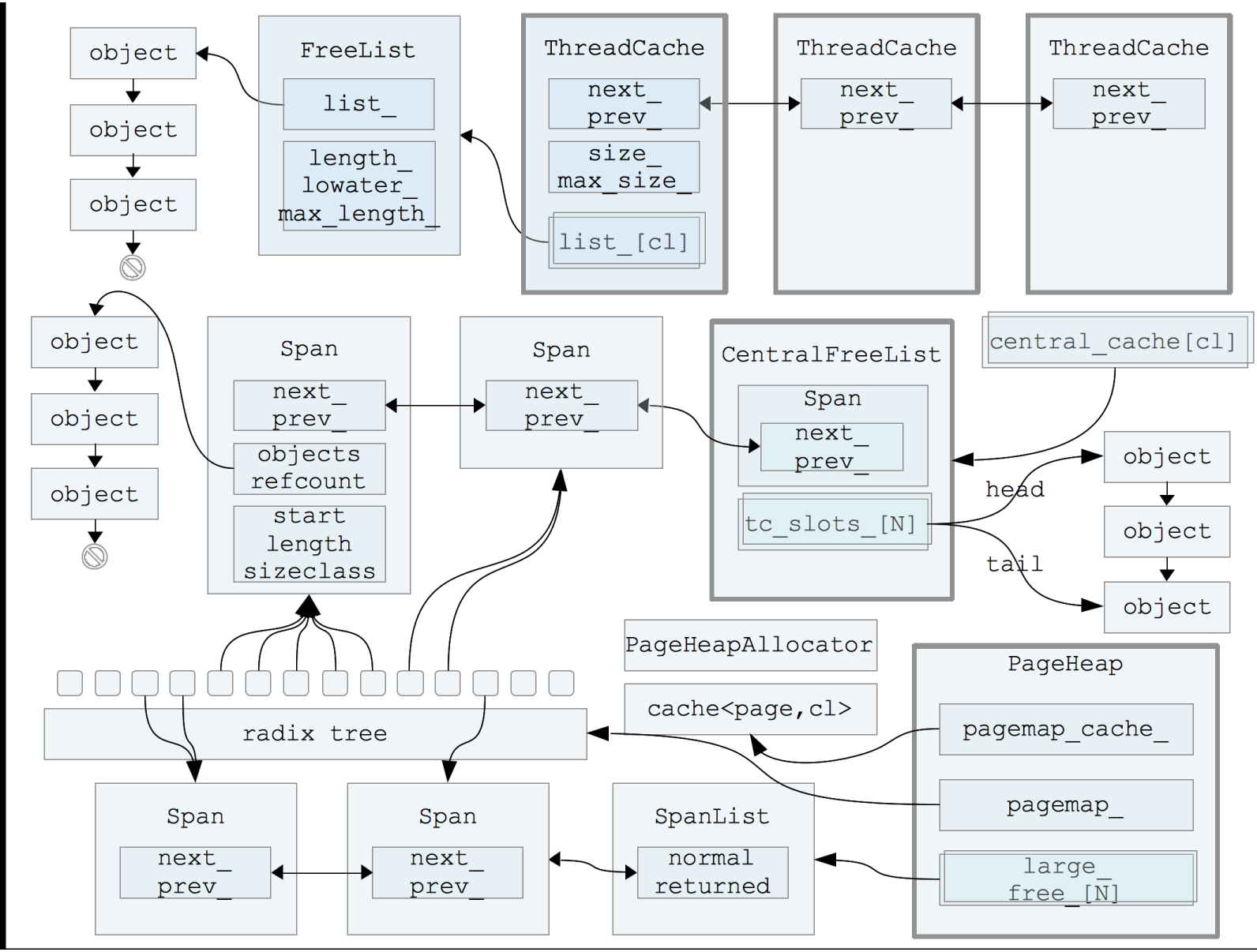


span

- 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.

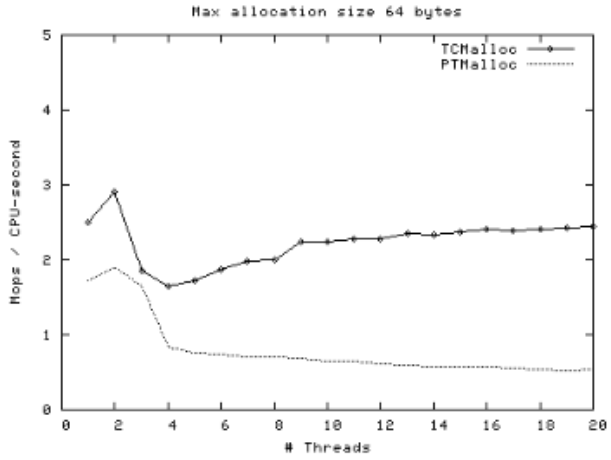


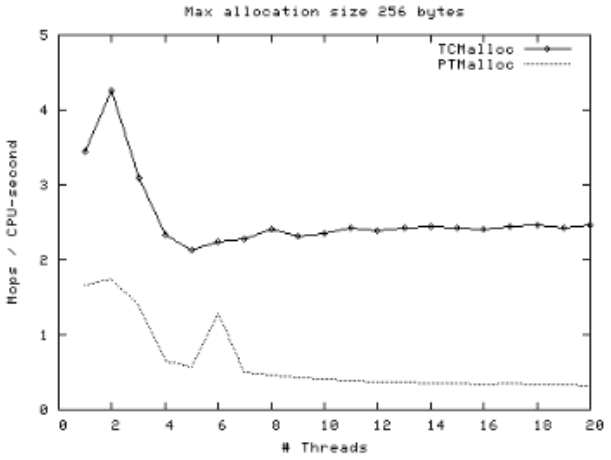

Small Request
Large Request
Thread Cache
Central Heap
Central Freelist
. . .
Class 0
Class 1
. . .
Thread Cache
Class 86
Class 87
OS
Thread Cache

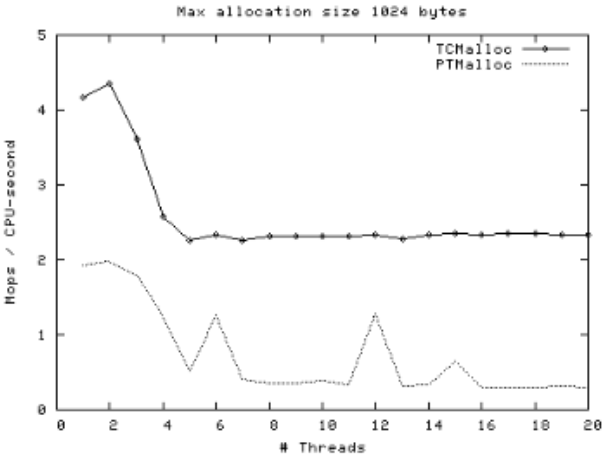

object
object
object
FreeList
list_
length_
lowater_
max_length_
ThreadCache
next_
prev_
size_
max_size_
list_[cl]
ThreadCache
next_
prev_
ThreadCache
next_
prev_
object
object
object
Span
next_
prev_
objects
refcount
start
length
sizeclass
Span
next_
prev_
CentralFreeList
Span
next_
prev_
tc_slots_[N]
central_cache[cl]
head
tail
object
object
object
PageHeapAllocator
cache<page,cl>
radix tree
PageHeap
pagemap_cache_
pagemap_
large_
free_[N]
Span
next_
prev_
Span
next_
prev_
SpanList
normal
returned

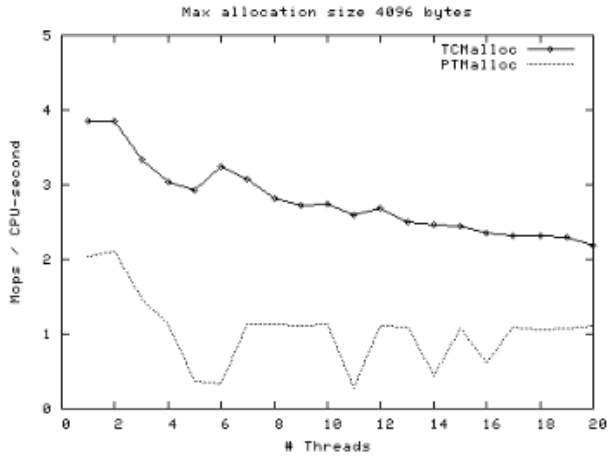
Deallocation

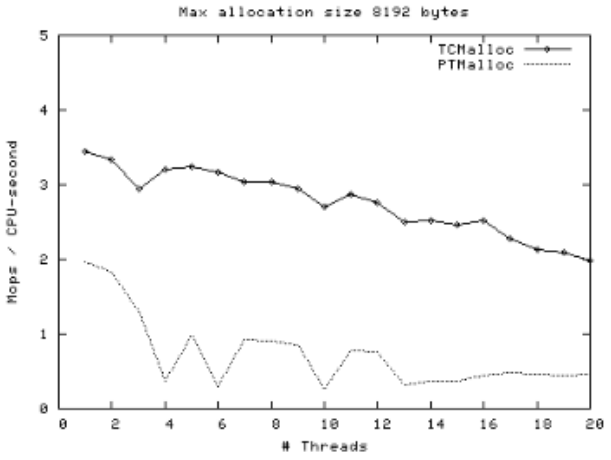
- When an object is deallocated, we compute its page number and look it up in the central array to find the corresponding span object.
- The span tells us whether or not the object is small, and its size-class if it is small.
  - If the object is small, we insert it into the appropriate free list in 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.

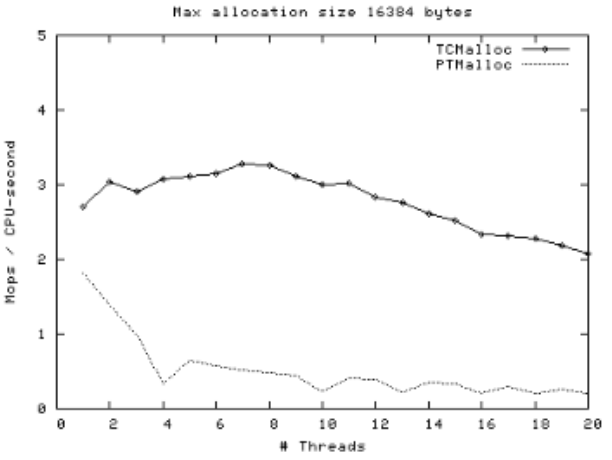

Max allocation size 64 bytes
TCMalloc
PTMalloc
Mops / CPU-second
Threads

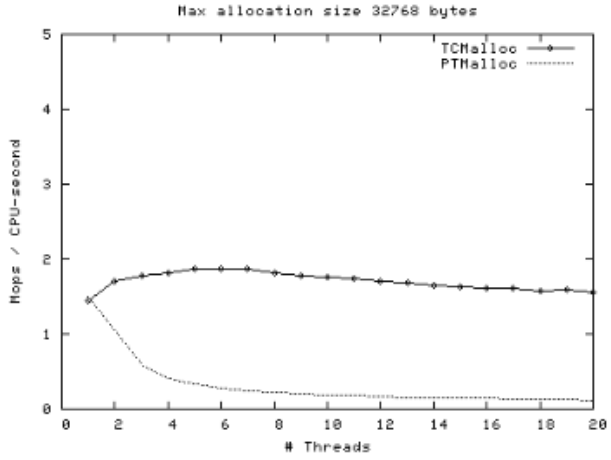

Max allocation size 256 bytes
TCMalloc
PTMalloc
Mops / CPU-second
Threads

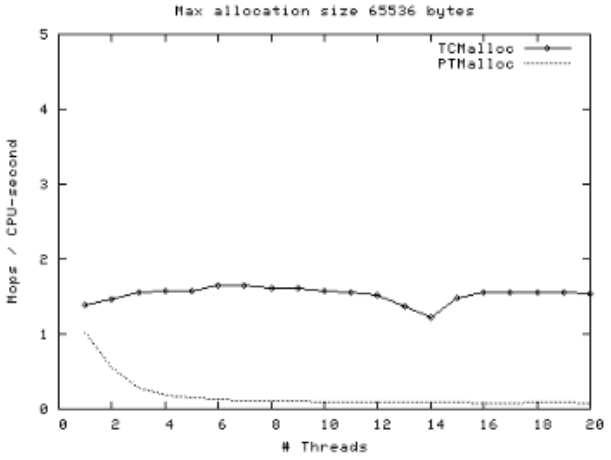

Max allocation size 1024 bytes
TCMalloc
PTMalloc
Mops / CPU-second
Threads

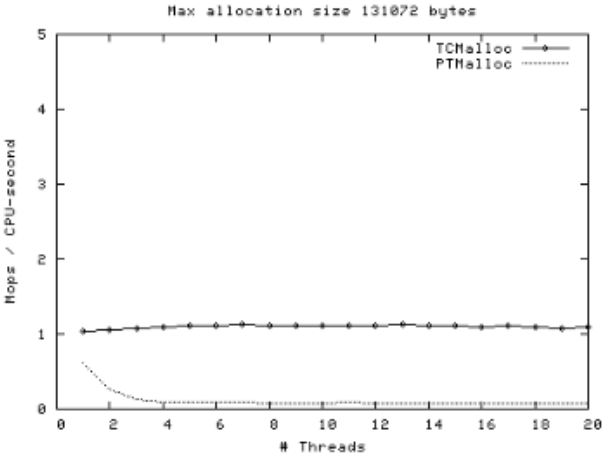

Max allocation size 4096 bytes
TCMalloc
PTMalloc
Mops / CPU-second
Threads


Max allocation size 8192 bytes
TCMalloc
PTMalloc
Mops / CPU-second
Threads


Max allocation size 16384 bytes
TCMalloc
PTMalloc
Mops / CPU-second
Threads


Max allocation size 32768 bytes
TCMalloc
PTMalloc
Mops / CPU-second
Threads


Max allocation size 65536 bytes
TCMalloc
PTMalloc
Mops / CPU-second
Threads


Max allocation size 131072 bytes
TCMalloc
PTMalloc
Mops / CPU-second
Threads