EE488A

Advanced Programming Techniques for Electrical Engineering, Fall 2020

HW2

Due date: 11/6/2020 (11:59:59pm)

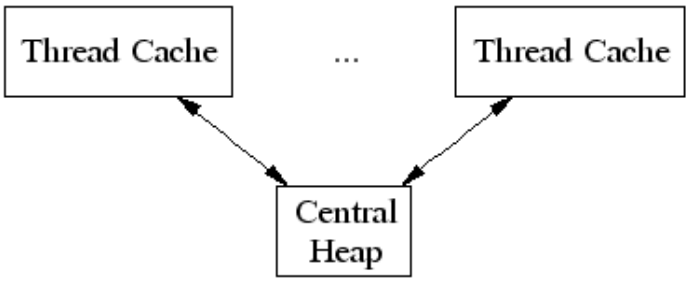
1. Introduction

In this homework, you will be implementing Thread-Caching Malloc (TCMalloc) that you have learned in lecture 8. Specifically, you need to implement *(i) Small Object Allocation, (ii) Large Object Allocation, (iii) Span Management, and (iv) Deallocation* functions of TCMalloc in a muti-threaded environment. Your program will be tested by TA's memory allocator program that performs simultaneous memory (de) allocation tasks using a number of threads. Through the test, we evaluate processing time of your program for the same operations to grade your scores.

2. TCMalloc

(1) Overview

Speed is very important for a memory allocation, and TCMalloc is the implementation to heavily consider the speed of allocation and freeing especially for muti-threaded programs. Traditional malloc suffers from performance degradation when a number of threads do memory allocation jobs, because a one heap-memory is simultaneously accessed by many threads, which results in severe lock contentions. To address this problem, TCMalloc assigns each thread a *thread-local cache*. It provides a clear benefit in terms of performance improvement; Small allocations are satisfied from the thread-local cache without accessing the central heap. 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.



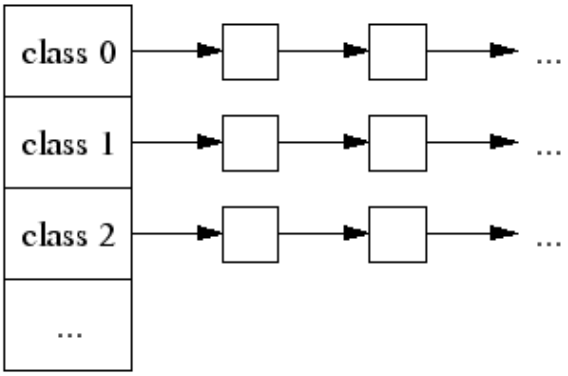
TCMalloc treats objects with size <= 32K ("small" objects) differently from larger objects. Large objects are allocated directly from the central heap using a page-level allocator (a page is a 4K aligned region of memory). i.e., a large object is always page-aligned and occupies an integral number of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

Note that you should use spin-lock for any locks in TCMalloc implementation.

(2) Small Object Allocation

Each thread cache maintains a list of size-classes Each small object size maps to one of approximately 158 (=8+ 31+ 119) allocatable size-classes. For example, all allocations in the ran

ge 961 to 1024 bytes are rounded up to 1024. The size-classes are spaced so that small size s are separated by 8 bytes (for sizes <= ~64bytes), larger sizes by 64 bytes (for sizes <= ~2 Kbytes), even larger sizes by 256 bytes. The spacing (for sizes >= ~2K) is 256 bytes.



Allocating a small object:

1) 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, remove the first object from the list and return it.

When following this fast path, TCMalloc acquires no locks at all.

If the free list is empty:

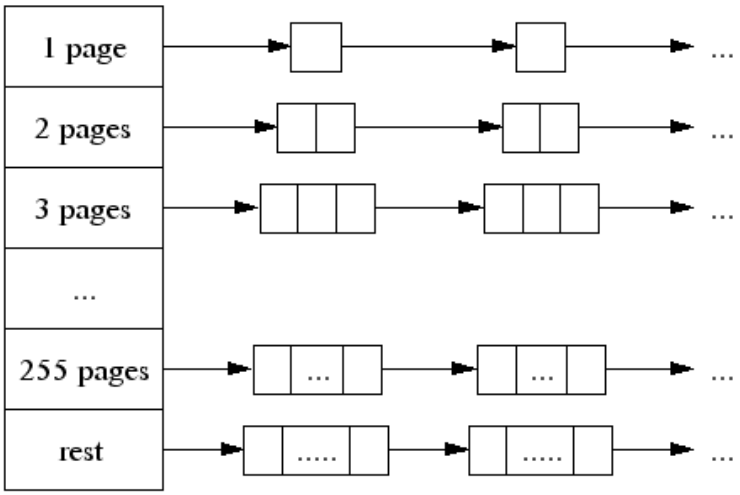
1) Fetch a bunch of objects from a central free list for the size-class (needs spin-lock)
2) Place them in the thread-local free list
3) Return one of the newly fetched objects to the applications

If the central free list is also empty:

1) 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) Move some of these objects to the thread-local free list.

You can freely design number of object to fetch and size of the large object with correspondin g small object. For example a run of one page (4K) can be carved up into 32 objects of size 1 28 bytes each.

(3) 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 kth entry is a free list of runs that consist of k pages. (1~4 pages lists are also required to make small object.) The 256th entry is a free list of runs that have length >= 256 pages. (Note that the entries in linked lists are spans.)

An allocation for k pages is satisfied by looking in the kth free list. If that free list is empty, TCMalloc should look in the next free list, and so forth. Eventually, the program look in the last free list if necessary. If that fails, the program should fetch memory from the system.

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.

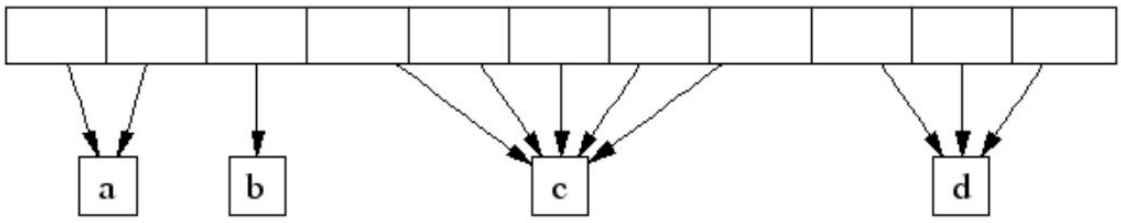
(4) Span Management

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 Span is free: The span is one of the entries in a page heap linked-list.

If Span is allocated: The span is either following two conditions.

1) It is a large object that has been handed off to the application,
2) 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.

(5) Deallocation

When an object is deallocated, you compute its page number and look it up in the central array to find the corresponding span object. The span tells you whether or not the object is small.

If object is small:

1) The span tells you its size-class. Then, you insert it into the appropriate free list in the current thread's thread cache. .

If object is large:

1) The span tells you the range of pages covered by the object.
2) If range of pages is [p,q], then you lookup the spans for pages p-1 and q+1. If either of these neighboring spans are free, you coalesce them with the [p,q] span. The resulting span is inserted into the appropriate free list in the page heap.

(5) Central Free lists for Small objects

As mentioned before, you should 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.

Please check the lecture 8 slide page 38.

3. Implementation

(1) APIs

Your TCMalloc will consists of the following functions which are defined in tc_malloc.c:

```
void *tc_central_init();
void *tc_thread_init();
void *tc_malloc(size_t size);
void tc_free(void *ptr);
```

- tc_central_init: Performs initialization of the central heap when called in a main thread and returns a pointer to the starting address of list. TAs do not explicitly define a specific size of the initial central heap, so find your optimal size through doing iterative tests.
- tc_thread_init: Creates a thread-local free list when invoked by a child thread and returns a pointer to the starting address of list.

- tc_malloc: Allocates an object desired by a child thread with size bytes and returns a p
  ointer that indicates the allocated object. Note that when tc_malloc is called, you need
  to consider the above mentioned allocation techniques according to the requested size
  (i.e., the small/large allocation policies) and manages the spans. Care must be taken w
  hen allocating large objects, since a child thread requests to access the central heap.
- tc_free: Frees the object pointed by *ptr and returns nothing.

(2) Interfaces

Your program must consist of two files, tc_malloc.c and tc_malloc.h. Your program must be c
ompiled with TAs makefile and be executed when the given APIs are called. Therefore, additi
onal and customized .c files and APIs are not allowed since they are not compatible with TA's
program.

(3) Testing Scenario

We will test your program how fast it processes heavy memory (de) allocation tasks when a
bunch of threads simultaneously invoke the APIs. Consider the following example scenario (it
might not be the same with the final testing program).

1) Create 1,000 threads and initialize the thread-local free list with defined size.
2) Each thread performs following steps.
   a) allocates 100 small objects which size is 8 bytes.
   b) allocates 100 small objects which size is 128 bytes.
   c) allocates 100 small objects which size is 1K bytes.
   d) allocates 10 large objects which size is 64K.
   e) frees the 200 small objects allocated in the step b and c.
   f) frees the large objects.

Note that the performance will be different depending on the individuals' laptop/desktop spe
cs, because the multi-thread program is highly affected by a number of available cores in you
r CPUs. Therefore, we highly recommend that you test your program in Haedong lounge serv
ers before submitting your program.

4. Submission instructions

Use KAIST KLMS to submit your homework. Your submission should be one gzipped tar file
whose name is YourStudentID_hw2.tar.gz. For example, if your student ID is 20191234, and it
is for homework #2, please name the file as 20191234_hw2.tar.gz. If you do not follow this zi
p file name, you will lose some points.

Your zip file should contain **three things**:

1. *One PDF* file for document which explains your codes (hw2.pdf). This PDF file should s
   how explanations for code in detail.
2. *One C file, tc_malloc.c. (only one c file is allowed)*
3. *One header file, tc_malloc.h.*

*Do not include Korean letters in any file name or directory name when you submit.
Important note: Files not following the above format will be heavily punished! (You will get 0p
)

Test environment : We will test your codes in ubuntu 18.04 build-essential.

Late submission policy : If you submit your homework after due, you will lose 20% for each late day(i.e. 80% of the full credit up to 24 hours late, 20% of the full credit up to 96 hours late)

Plagiarism

Discussions with other people are permitted and encouraged. However, when the time comes to write your solution, such discussions (except with course staff members) are no longer appropriate: you must write down your own solutions independently. If you received any help, you must specify on the top of your written homework any individuals from whom you received help, and the nature of the help that you received. Do not, under any circumstances, copy another person's solution. We check all submissions for plagiarism and take any violations seriously.

*Plagiarism will get severely penalized If detected, 0 points for all assignments (both providers and consumers)