ECE454, Fall 2017
Homework4: Multi-threaded Parallelization
Assigned: Nov 7th, Due:Nov 21st, 11:59PM

The TAs for lab assignment 4 are: Arnamoy Bhattacharyya (arnamoyb@ece.utoronto.ca),
Ali Jokar (sa.jokar@mail.utoronto.ca), and Yongle Zhang (yongle.zhang@mail.utoronto.ca).

1 Introduction

OptsRus is having an amazing year so far. After previous successful projects, the company is now well known for their excellent system optimization. In the last project, you were assigned to create a dynamic memory allocator for C programs for a new experimental operating system. One of the OptsRus industry partner (OS161 Ltd.) learned about the success of this project and wants you to improve their existing legacy dynamic memory allocator. Currently, the memory allocator is strictly single threaded, but OS161 Ltd. wants you to implement a high performance multi-threaded version for their upcoming product.

2 Motivation

The allocation and management of blocks of virtual memory is a key function required by nearly all modern programs, including the operating system itself. The use of a memory allocator designed for sequential programs can be a serious bottleneck limiting the performance and scalability of multi-threaded parallel programs. Fast operation and efficient use of memory are the typical criteria by which a sequential allocator is evaluated; parallel allocators must also take scalability into account. In OS161 Ltd's case, the company owns high end servers starting from ones with 8 core CPU 128GB RAM configuration. Therefore, a high performance multi-threaded parallel allocator is essential for the company to fully utilize the resource provided by the hardware.

The naive implementation OS161 Ltd had was a simple sequential allocator. The simplest way to adapt a sequential allocator for use by parallel programs – placing a single lock around the bodies of the malloc and free functions – serializes all memory allocation and deallocation requests, limiting scalability. Even if the allocator is designed to allow individual malloc and free requests to proceed (mostly) in parallel, scalability problems could remain due to cache effects. Your job is to create a fast memory allocator for use by parallel program while maintaining a certain level of efficiency.

3 Logistics

We are providing a package with a sample serial allocator (kheap) that has been made thread-safe by the use of a single lock that surrounds all malloc and free operations. The only purpose for this piece of code is to showcase how to make your previous lab 3 code work in multi-threaded environment by placing coarse grained locks. Secondary purpose include providing groups who did poorly some starter code. Be warned that the performance of included kheap memory allocator has poor performance against many group's optimized memory allocators from lab 3. There is also a thin wrapper around the libc malloc and free routines for comparison (libc). The package also contains some

sample benchmarks that you can use (and that we will use!) to evaluate your allocator, and some utility routines that you can and/or must use in your implementation.

Start by copying the `hw4.tar.gz` file from UG shared directory
`/cad2/ece454f/hw4/hw4.tar.gz`
into a protected directory within your **UG** home directory.

Then run the command:
`tar xzvf hw4.tar.gz`
This will cause a number of files to be unpacked into the directory.

The starter code contains the following directories:

- allocators: contains one subdirectory for the implementation of each allocator.
  - kheap - an allocator based on the OS/161 kernel heap allocator
  - libc_wrapper - thin wrappers around the standard libc implementations of malloc and free

  We have also set up a directory for your work named alloc with an alloc.c file for you to fill in.
- benchmarks: contains one subdirectory for each benchmark. They are:
  - threadtest - tests basic scalability of the allocator
  - larson - simulates a server by having each thread allocate and deallocate objects and then transfer some objects to other threads to be freed.
  - cache-scratch - tests for passive false sharing
  - cache-thrash - tests for active false sharing
  - autoTester.py - python 3 script which gives you a mark which you can use to compare relative speedup of your code between versions

  Each benchmark subdirectory contains scripts to run and graph the performance of the allocators provided, and a Makefile to create a version of the benchmark linked to each of the different allocators. The Makefiles and runscripts include the (currently a copy of the slow kheap) ”alloc” allocator such that the program will compile. You are expected to replace the placeholder implementation in alloc.c with your own.
- util: contains utility routines for use in the benchmarks and allocators. In mm_thread.c are operations related to threads (initializing a pthread_attr_t for thread creation, determining the number of processors on a system, and pinning a particular thread to a particular CPU). In tsc.c are the cycle counter routines used to measure performance Finally in memlib.c are the routines that your allocator must use to manage its heap.
- include: header files used by the benchmarks, allocators, and utility routines.

To build the sample allocators, you need to define an environment variable, TOPDIR, to be the top-level directory where you checked out the code.

E.g., if you are using bash:
`export TOPDIR=/nfs/ug/homes-5/m/mysteryUser/lab4`

E.g., if you are using csh:
`setenv TOPDIR /nfs/ug/homes-5/m/mysteryUser/lab4`

You may consider adding this line to your `.bashrc or other initialization configuration` file to automatically execute this command once you login.

4 Specifications

Your allocator consists of the following three functions, which are declared in `include/malloc.h`:

```
int   mm_init(void);
void *mm_malloc(size_t size);
void  mm_free(void *ptr);
```

In the `allocators/alloc` subdirectory, you must fill in these function bodies in the file named alloc.c. Feel free to experiment with different designs. However, the only file you will be submitting is alloc.c and as a result, we will only use `alloc/alloc.c` file for marking your implementation.

All your code must be contained in the file named `alloc.c`; you are not allowed to change any of the interfaces to the three functions above. Of course you can define as many helper routines as you like within this file.

Your allocator interacts with an arbitrary application program in the following way. As part of its initialization phase, the application calls your `mm_init` function to perform initialization of the heap. You must allocate the necessary initial heap area (starting with a call to mem_init to create the chunk of memory that you can use), and initialize all structures that you need. The application then makes a series of calls to `mm_malloc` and `mm_free`. You may assume that any programs we test with will always call `mm_init` before any calls to `mm_malloc/mm_free` and before creating any threads.

You are allowed to use any of the functions from memlib.h in your allocator:

- **mem_init**: Use this function to create the chunk of memory that forms your heap. This function uses the libc malloc to obtain a 256MB chunk. All of the benchmark programs we use fit easily within this space, but you may run into trouble if your allocator allows unbounded fragmentation.
- **mem_sbrk**: You use this function to expand your heap area (within the overall 256MB limit!). The lower and upper boundaries of the heap area are stored in the global variables dseg_lo and dseg_hi respectively. You are allowed to read these variables in your allocator, but you are not allowed to modify them in any way. You must call mem_sbrk to change the upper bound. This function accepts a positive integer argument, which is the number of bytes by which to increase the upper bound. The return value is the beginning of the newly added heap area, or NULL if there wasn't enough memory left. The interface of mem_sbrk is very similar to the sbrk system call, and the general effect is the same. However, you cannot decrease the heap area in size, only increase it, so be careful how you call mem_sbrk. In effect, each time you call mem_sbrk the value of dseg_hi is incremented by the amount you request. It may be convenient for your allocator to work with blocks of memory that are guaranteed to be pagesize-aligned (or some other size that you choose). To help with this, the initial heap is aligned on a page boundary, and you can call mem_pagesize to find out the system page size.
- **mem_usage**: This is simply a shorthand that returns the current size of the heap in bytes.

The functions in memlib.c are not generally thread-safe. They are safe in the current allocators because they are only called from within routines that are themselves protected by a lock. If you use finer-grained locking, you must ensure that calls to mem_sbrk are serialized by grabbing a suitable lock before the call and releasing it after the return. You are not allowed to modify the functions in memlib.c themselves.

The functions you are to implement are the following:

- **mm_init**: Before calling mm_malloc or mm_free, the application program calls mm_init to perform any necessary initializations, including the allocation of the initial heap area. The return value should be -1 if there was a problem with the initialization, 0 otherwise.
- **mm_malloc**: The mm_malloc routine returns a pointer to an allocated region of at least size bytes. The pointer must be aligned to 8 bytes, and the entire allocated region should lie within the memory region from dseg_lo to dseg_hi.
- **mm_free**: The mm_free routine is only guaranteed to work when it is passed pointers to allocated blocks that were returned by previous calls to mm_malloc. The mm_free routine should add the block to the pool of unallocated blocks, making the memory available to future mm_malloc calls.

The allocator provided, kheap, meets the specification provided above, but it is deficient as a parallel allocator. It will actively induce false sharing and is not scalable. Its baseline sequential performance and memory utilization could also be improved.

5 Coding Rules

You may write any code you want, as long as it satisfies the following:

- You submission have only the modified alloc.c file and not any other file.
- You may not statically allocate large data structures for your heap. (For instance, the original OS/161 kheap.c created a page-sized array of references to pages in the heap as a global variable. In the implementation provided with this assignment, the space for these references is also allocated from the heap.) In the extreme, all state could be stored in the heap itself, at the beginning, with dseg_lo providing the pointer to the heap's own data structures. For this assignment, some globals variables are acceptable, but no more than 1KB worth of global variables can be used.
- You may assume that each thread making requests of your allocator will execute exclusively on a single CPU. Our test programs ensure this using the sched_setaffinity system calls. You may also assume that we will not run more than one thread per CPU.
- You may read any and all publications relating to parallel (or sequential) memory allocation to help you understand the problem and your design. You may choose to implement a design described in the literature.
- You must provide a citation with your report citing any resources that you used. These should be discussed in your short report.
- You may not copy the source code for any memory allocators other than the ones provided in the assignment distribution. You must implement your own allocator; using ideas from the literature is fine, using other people's code is not.

6 Evaluation

Your code will be run on 4 of the benchmarks provided (cache-scratch, cache-thrash, threadtest, and larson). We suggest that you do not submit code that does not compile, code with infinite loops, etc. The script will nevertheless detect such special cases and kill your implementation due to taking too long. The default timeout is 30 minutes for all tests to complete.

A report in plain text file report.txt needs to be submitted as well. Please describe your implementation in the report in 150 words or less. If the implementation detail submitted in the report does not match your actual implementation, you will be be penalized at the discretion of the TA.

6.1 Marking Scheme

The total available marks are divided into 2 portions (60% + 40%). The first portion is for correctness. You will receive full marks if your solution compiles and runs successfully within 30 minutes (very generous time). The second portion is performance which is described in detail below.

Correctness Portion - 60%

The correctness portion should be fairly easy and serves the minimal requirement. If you submit existing kheap code without any modification, you not be penalized. However, by submitting the existing code, means you will receive a very low mark for the performance portion. You will be heavily penalized at TA's discretion if you submit a version of memory allocator slower than the one in existing code.

Performance Portion - 40%

This lab is designed to have very high potential for performance optimization. For this portion, we will be using an automated scoring system. Once you submit your work using the usual `submitece` command, your submission will be placed onto a queue for auto-grading.

Performance is measured in following categories: sequential speed, scalability, false sharing avoidance and fragmentation.

Marks will only be assigned if all threads run successfully!

Speedup pts (accounts for 80% of performance portion):
15% - single threaded performancew.r.t libc malloc (average sequential speed score from all benchmarks)
65% - multi-threaded performance w.r.t to libc malloc (average scalability score from threadtest and larson benchmarks)
12% - false sharing avoidance w.r.t libc malloc (average scalability score from cache-scratch and cache-thrash benchmarks)

Memory fragmentation pts (accounts fo 20% of the performance portion):
8% - calculated by obtaining the min and max memory used during each benchmark with various paralellization thread count

Tips for testing locally:

- Use the autoTester script found in /cad2/ece454f/hw4/autoTester.py to get a lowerbound mark
- Optimize your malloc and free for individual benchmarks separately instead of everything at once
- Run only the benchmark test inside benchmark/runTests.pl you are trying to optimize (expected workflow)

Notes for testing on the test server:

- Submitting your solution using `submitece454f alloc.c command` will trigger a remark of solution
- WEB access URL: http://ece454-lab4.dsrg.utoronto.ca/
- Notable Features: performance mark, performance mark breakdown, benchmark output download
- The web access allows you to see your performance marks. However, threshold which allows you to achieve 100% on the lab will be announced on Piazza shortly. As a result, the marks you see might fluctuate for the for couple days.

7 Submission

When you have completed the lab, you will hand in exactly files, `alloc.c` that contains your solution. For teams of 2, **only submit from 1 partner's ug account**. If there are two submissions, a **random partner's submission will be selected**.

The standard procedure to submit your assignment is by typing
`submitece454f 4 alloc.c`
on one of the UG machines.

- Make sure you have included your identifying information in the `name_t myname` structure in tt alloc.c.
- Remove any extraneous print statements.

8 Performance Testing

Obtaining good and accurate measurement of your allocator's performance on a shared machine is very difficult. We have configured an 8 core 16 thread server specifically for this assignment. The server's configurations matches typical machines found in the industry and is a good chance to practice writing programs for realistic server environment.

Details of the server hardware and software setup on the server can be found below:

- Output of "uname -srvmo": Linux 4.4.0-57-generic #78-Ubuntu SMP Fri Dec 9 23:50:32 UTC 2016 x86_64 GNU/Linux
- GCC version: 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1 16.04.4)
- Output of lscpu:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                16
On-line CPU(s) list:   0-15
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             1
NUMA node(s):          1
```

```
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
Stepping:              2
CPU MHz:               1212.000
CPU max MHz:           2400.0000
CPU min MHz:           1200.0000
BogoMIPS:              4794.17
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              20480K
NUMA node0 CPU(s):     0-15
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx
pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology
nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est
tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt
tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm epb tpr_shadow vnmi
flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid cqm
xsaveopt cqm_llc cqm_occup_llc dtherm ida arat pln pts
```

- Snapshot of /proc/meminfo

```
MemTotal:       131905472 kB
MemFree:         9648780 kB
MemAvailable:   49663756 kB
Buffers:          577220 kB
Cached:         116623100 kB
SwapCached:        14552 kB
Active:         57359000 kB
Inactive:       61296008 kB
Active(anon):   30395084 kB
Inactive(anon): 49642964 kB
Active(file):   26963916 kB
Inactive(file): 11653044 kB
Unevictable:           20 kB
Mlocked:               20 kB
SwapTotal:      134086652 kB
SwapFree:       128440404 kB
Dirty:              1040 kB
Writeback:             0 kB
AnonPages:       1440544 kB
Mapped:         47445516 kB
Shmem:          78583360 kB
Slab:            2267484 kB
SReclaimable:    2192100 kB
SUnreclaim:        75384 kB
KernelStack:       13312 kB
PageTables:       200376 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:    199917532 kB
Committed_AS:   89003624 kB
VmallocTotal:   34359738367 kB
VmallocUsed:           0 kB
```

```
VmallocChunk:          0 kB
HardwareCorrupted:     0 kB
AnonHugePages:   1052672 kB
CmaTotal:              0 kB
CmaFree:               0 kB
HugePages_Total:     119
HugePages_Free:      111
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:      325628 kB
DirectMap2M:    42534912 kB
DirectMap1G:    93323264 kB
```