Asst2-416
OS Assignment 2 (My Allocate & My Deallocate)

Authors: Bruno Lucarelli
Alex Marek
Joseph Gormley

Library: my_pthread_t.h

iLab machine: man.cs.rutgers.edu

========================
0. CORE FUNCTION
========================

<u>myallocate()</u>
This function provides the calling thread an address to the start of a block of a requested memory size. Using strategies discussed in class we give a thread the illusion of an increased memory capacity while keeping a continuous state of memory. To give this illusion the library uses the hard disk to deliver additional pages while constantly reordering pages in the background.
Note: The threads do not know they are using myallocate() vs. malloc()

<u>mydeallocate()</u>
Once the requested memory is no longer needed, it can be deallocated by passing the pointer to the allocated block through this function. Once passed through it sets the block to free and goes through various checks to see if it can prevent fragmentation by combining its free space with neighboring free space. Once combined, if the resulted space results in an entire free page (or pages), the *memory manager* updates the pages to free so the space can be used once again.

shalloc()
Very similar to LIBRARYREQ logic for myallocate(), myshalloc() provides the user with an address to the start of a block of a requested memory size. This space can be accessed by any thread and is used as a shared space between all created threads. This space is limited to four pages and is not involved in the *memory manager's* page swapping mechanism.
=======================
1. DESIGN / DATA STRUCTURES
=======================
'Metadata'
SegMetadata

The members contained in this metadata ultimately gives a segment its *block* like shape. After a page's memory is organized into such block(s), the *memory manager* can produce memory efficient decisions that are dependent on the size of the requested space.

PageMetadata

This is essentially a block of data that stores information relevant to a single page. Such information allows the *memory manager* to protect pages from unknown threads while also giving threads the illusion of continuous memory. It is important to note that this block of data is stored in an additional data structure while SegMetadata is stored as part of the threads memory.

ThreadMetadata

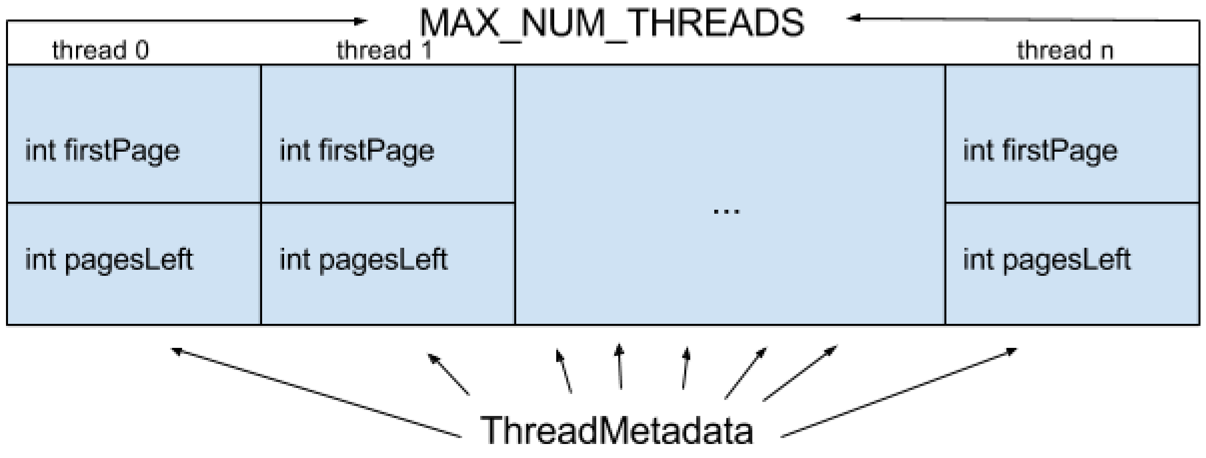
This struct is designed to do simple bookkeeping on the details of each individual thread. The main purpose of this is to allow easy iteration of the owned pages of a specific thread. With this in mind, we can use this information to track what memory a thread has requested and freed.

Unlike PageMetadata and ThreadMetadata, the number of structs used is dependent on a defined amount of threads (MAX_NUM_THREADS). Since this value is predetermined these structs are stored in an array inside the kernels space.

‘Thread Node List’
Shortly after the kernel space is configured and the kernel’s initialization has begun we iterate this structure and set each thread’s ThreadMetaData to its default values to indicate that the thread has not allocated any memory. As a thread allocates space its first page will be set and pagesLeft will be decremented accordingly.

Note: Each thread is referenced by an unique id which makes referencing threads metadata extremely efficient compared to your typical linked list.

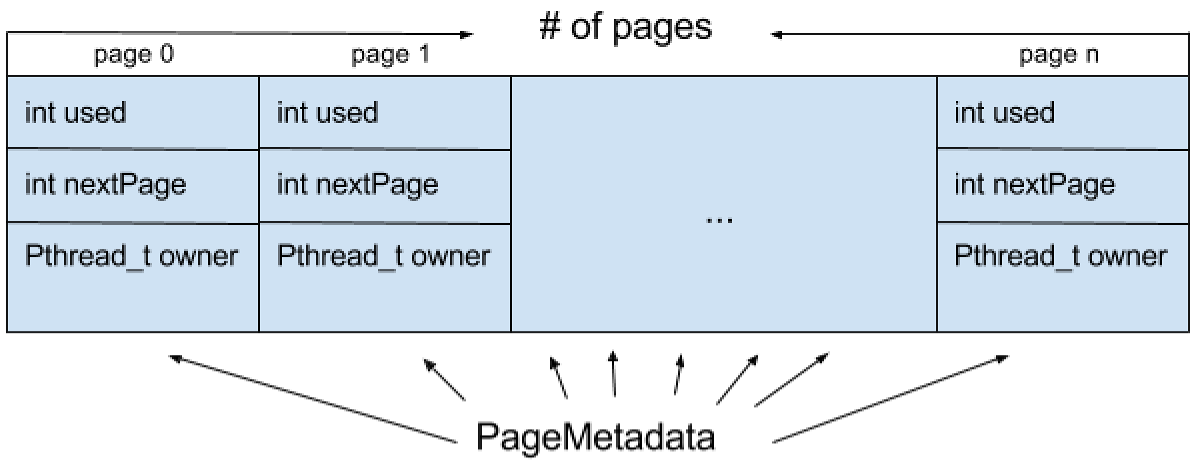
Ex. threadNodeList[0] will reference thread 0’s ThreadMetadata
threadNodeList[1] will reference thread 1’s ThreadMetadata
threadNodeList[x] will reference thread x’s ThreadMetadata



‘Page Table’
As mentioned previously, this data structure plays a large part in the iteration of a thread’s pages. Similar to ThreadMetadata containing a firstPage member, the pageTable has a nextPage member that holds the number of the following page. Keep in mind that this will not always be in a numerical order due to the abstraction of continuous space.

This design also allows the library to handle complex allocations such as an allocation that is larger than one page (4096 bytes). Cases as such will

leak onto as many pages as the allocation needs (within the limits of RAM). Any requests greater than the physical free memory will evict a page in accordance with a left to right linear algorithm (starting at page 0)
as long as the page is not owned by the active thread. The evicted page is written to the hard disk and will be retrieved once needed while the newly freed space will be given to the requesting thread.



Ex. PageTable[0] will reference page 0’s PageMetadata
PageTable[1] will reference page 1’s PageMetadata
PageTable[x] will reference page x’s PageMetadata

‘Swap File’

The swap file is in a format much simpler than that of the other structures. It has space in the PageTable (over 2/3’s of it to be exact) but essentially emulates extra “user space”. Thus, it only stores SegMetadata structs and the segments they describe. It is read into a globally-accessible character array of size 16MB called swapFile.

=================
2. PAGE SWAP DETAILS
=================
The default convention for a page that is not-currently owned, is that the page will have NO permissions (PROT_NONE). Pages swap locations in one of two situations:

1. A thread attempts to read a page that is not its own.
2. A thread needs to allocation a pointer at a certain location in virtual memory, but the location is in a page currently-owned by another thread.

Our function, swapPages() in my_pthread.c, handles the formal logic for swapping the metadata and actual data of two pages, while also accepting a parameter that tells which thread is the current thread. Protections are also unlocked at the beginning of the function/locked accordingly at the end of it. Cases specific to our convention (such as a page being free) are also considered in the process of swapping metadata and setting protections.

Upon entering Phase C, we modified swapPages() to interact with the swap file accordingly. Instead of performing I/O every time a page is in the swap file, we perform a check before swap-critical sections to see if a thread has any pages in the swap file *or* if it will need to access the swap file. If either of these is true, the file's contents are read into a 16MB character array. The array, modified by the swap operations, is written back into the file at the end of these critical sections.

==============
TESTS USED
==============

To use our test files:

1. Type “make clean” in the head directory.
2. Type “make” in the head directory.
3. Type “make clean” in the Benchmark director.
4. 4. Type “make” in the Benchmark directory.
5. Type “./executableName” to test out your choice of executable listed below.

<u>Test1.c</u>
This is a basic test that tests one multiple page request from the main thread and is shortly freed after. This space is not written to keep complexity of the test at a minimum.

<u>Test2.c</u>
This is a basic test that tests the ability of handling multiple requests from the main thread. Each requested block is written into and the data in the block is verified to ensure correct decisions are being made by the *memory manager*. After verification each block is freed.

<u>Test3.c</u>
Similar to Test2.c, Test3.c runs allocation, verification and free followed by a second allocation to verify the state of memory is properly booked after mydeallocate() is called.

<u>Test4.c</u>
This test is designed to make consecutive allocations of varying size (four, to be exact). Ideally, one would allocate a very large chunk of space, then a very small one, before freeing the two. Then, one would repeat the process.

Test5.c

This test is designed to see how the library handles a very large amount of small, variable-sized allocations. This stresses the members of PageTable and tests for accuracy among the data structure. Any slight discrepancies could offset the SegMetadata by unforeseeable amounts.

MultiTest1.c

This is a basic test that tests the ability to handle requests of varying size from multiple threads, the threads running in succession. One very large set of allocations can be made and then freed by the same thread in the same cycle. It also helps gauge the "sanitation" of our freeing procedures, as a healthy library should handle an enormous number of allocations made on top of space that is being constantly-freed. This puts a large amount of stress on our system's ability to handle large allocations (try 100,000 as the arrSize!).

MultiTest2.c

This is a higher-stress test that verifies that the memory manager and scheduler can handle many read requests from many different threads, over shared space. Threads allocate arrays of a given size, storing their thread ID's in the space, and then infinitely verify their space through read operations until they get interrupted. This means that pages are constantly being swapped into position on user-induced, "expected" segmentation faults. The size of the allocations made can be changed to only test Phase B functionality (up to size 50,000) or Phase C functionality.

vectorMultiply.c

This is a slightly-modified version of the original benchmark. It provides a case to test shalloc(). The global "counter" that stores arguments for the threads, is stored in the shared space and accessed by all of the involved threads for the vector calculations.